Министерство образования Республики Беларусь
Учреждение образования
«Полоцкий государственный университет»

П. В. Павловец

МИКРОПРОЦЕССОРНЫЕ СРЕДСТВА И СИСТЕМЫ

Учебно-методический комплекс
для студентов специальности 1-40 02 01
«Вычислительные машины, системы и сети»

Новополоцк
ПГУ
2011

УДК 004.31:621.382.049.77(075.8)
ББК 32.973.2я73
П12

Рекомендовано к изданию
методической комиссией факультета информационных технологий
в качестве учебно-методического комплекса (протокол № 2 от 18.02.2010)

РЕЦЕНЗЕНТЫ:
зам. нач. цеха ТАИ по АСУТП Новополоцкой ТЭЦ В. Е. ПИТОЛИН;
канд. техн. наук, доц., зав. каф. вычислительных систем и сетей
УО «ПГУ» Р. П. БОГУШ

Павловец, П. В.

П12 Микропроцессорные средства и системы : учеб.-метод. комплекс для студентов специальности 1-40 02 01 «Вычислительные машины, системы и сети» / П. В. Павловец. – Новополоцк : ПГУ, 2011. – 364 с.
ISBN 978-985-531-199-8.

Сформулированы цели и задачи курса, приведена основная терминология, дана классификация существующих микропроцессоров и рассмотрена эволюция микропроцессорной техники. Описаны основные архитектурные аспекты организации микропроцессорных систем, способы организации работы универсальных микропроцессоров, методы оптимизации быстродействия микропроцессорной техники, приведен обзор архитектуры и методов программирования микроконтроллеров и специализированных микропроцессоров обработки сигналов, рассмотрены перспективы развития микропроцессорной техники. Даны методические рекомендации по выполнению лабораторных работ, и описана система выставления оценок.

Предназначен для студентов специальности 1-40 02 01«Вычислительные машины, системы и сети».

УДК 004.31:621.382.049.77(075.8)
ББК 32.973.2я73

ISBN 978-985-531-199-8

© Павловец П. В., 2011
© УО «Полоцкий государственный
университет», 2011

СОДЕРЖАНИЕ

РАБОЧАЯ ПРОГРАММА 5

ЛЕКЦИОННЫЙ КУРС 7

МОДУЛЬ 0. ВВЕДЕНИЕ 7

1. Цели и задачи курса 7
2. Микропроцессор и микропроцессорная система 8
3. Основные понятия и определения 10
4. Характеристики микропроцессоров 10
5. Классификация микропроцессоров 12
6. Эволюция микропроцессоров 15

МОДУЛЬ I. ОРГАНИЗАЦИЯ МИКРОПРОЦЕССОРНОЙ СИСТЕМЫ 20

1. Основные типы архитектур микропроцессорных систем. Фон-неймановская (принстонская) и гарвардская архитектуры. Организация пространств памяти и ввода/вывода 20
2. Магистраль микропроцессорной системы. Стандартная структура шины. Трехшинная магистраль с раздельными шинами передачи адреса и данных. Совмещение шины адреса и шины данных. Циклы обращения к магистрали. Временные диаграммы работы шины. Организация обращения к магистрали с синхронным и асинхронным доступом. 27
3. Архитектура системы ввода/вывода. Способы организации передачи данных. Система непосредственного ввода/вывода. Система канального ввода/вывода. Программно-управляемый ввод/вывод. Прямой ввод/вывод. Условный ввод/вывод. Ввод/вывод с программным квитированием. Ввод/вывод по прерываниям 41
4. Понятие прерывания процессора. Организация подсистемы прерываний в МПС. Контекстное переключение. Организация радиальной системы прерываний. Метод поллинга. Организация векторной системы прерываний. Вектор прерывания 51
5. Прямой доступ к памяти. Организация прямого доступа к памяти. Контроллер ПДП 64
6. Память микропроцессорной системы. Функции памяти. Архитектура и иерархия памяти. Организация кэш-памяти. Виртуальная память 70

МОДУЛЬ II. УНИВЕРСАЛЬНЫЕ МИКРОПРОЦЕССОРЫ 85

1. Определение понятия «архитектура». Архитектура системы команд. Классификация процессоров CISC и RISC 85
2. Методы адресации и типы данных. Типы команд. Команды управления потоком команд 88
3. Конвейеризация и параллелизм. Конвейерная организация обработки данных. Простейшая организация конвейера и оценка его производительности 94
4. Структурные конфликты и способы их минимизации. Конфликты по данным, остановы конвейера и реализация механизма обходов. Сокращение потерь на выполнение команд перехода и минимизация конфликтов по управлению 97

5. Проблемы реализации точного прерывания в конвейере. Обработка многотактных операций и механизмы обходов в длинных конвейерах 111
6. Параллелизм на уровне выполнения команд, планирование загрузки конвейера и методика разворачивания циклов. Конвейерная суперскалярная обработка 120
7. Зависимости. Классификация зависимостей и их применение. Устранение зависимостей по данным и механизмы динамического планирования. 135
8. Аппаратное прогнозирование направления переходов и снижение потерь на организацию переходов. Буфер прогнозирования переходов. Корреляционая схема. Другие методы сокращения приостановок по управлению. 154
9. Одновременная выдача нескольких команд для выполнения и динамическое планирование 163
10. Архитектура машин с длинным командным словом (VLIW). Средства поддержки большой степени распараллеливания 169
11. Архитектура EPIC 201
МОДУЛЬ III. МИКРОКОНТРОЛЛЕРЫ И СПЕЦИАЛИЗИРОВАННЫЕ МИКРОПРОЦЕССОРЫ 208
1. Микроконтроллеры. 8-разрядные микроконтроллеры. 16- и 32-разрядные микроконтроллеры. Структурная организация. Построение микропроцессорной системы на базе микроконтроллеров. Особенности программирования 208
2. Специализированные микропроцессоры. Цифровые процессоры обработки сигналов 225
МОДУЛЬ IV. ПЕРСПЕКТИВЫ РАЗВИТИЯ МИКРОПРОЦЕССОРНОЙ ТЕХНИКИ 234
ЛАБОРАТОРНЫЙ КУРС 243
7 СЕМЕСТР 243
Лабораторная работа 1 243
Лабораторная работа 2 247
Лабораторная работа 3 249
Лабораторная работа 4 255
8 СЕМЕСТР 261
Лабораторная работа 1 261
Лабораторная работа 2 294
Лабораторная работа 3 316
Лабораторная работа 4 338
ОЦЕНКА РАБОТЫ СТУДЕНТОВ. РЕЙТИНГОВАЯ СИСТЕМА 360
ЛИТЕРАТУРА 363

РАБОЧАЯ ПРОГРАММА

№ п/п	Наименования разделов и тем лекций и их содержание [1]	Количество часов	
		Д	З
		П	П
1	2	3	4
	Лекционный курс		
	Модуль 0. Введение		
1.	Цели и задачи курса. Основные понятия и определения. Микропроцессор и микропроцессорная система (МПС). МикроЭВМ. Структура микроЭВМ. Однокристальная микроЭВМ. Программируемый микроконтроллер. Микропроцессорный комплект (семейство) БИС. Микропроцессорные средства. Характеристики и классификация микропроцессоров. Универсальные микропроцессоры. Микроконтроллеры. Специализированные микропроцессоры. Эволюция микропроцессоров.	4	-
	Модуль I. Организация микропроцессорной системы[2]		
2.	Основные типы архитектур микропроцессорных систем. Фоннеймановская (принстонская) и гарвардская архитектуры. Организация пространств памяти и ввода/вывода.	2	-
3.	Магистраль микропроцессорной системы. Стандартная структура шины. Шина данных. Шина адреса. Шина управления. Трехшинная магистраль с раздельными шинами передачи адреса и данных. Совмещение шины адреса и шины данных. Шина адреса/данных. Двухшинная магистраль с совмещенными шинами передачи адреса и данных. Циклы обращения к магистрали (циклы шины). Цикл чтения из памяти (ввода из порта). Цикл записи в память (вывода в порт). Временные диаграммы работы шины. Организация обращения к магистрали с синхронным и асинхронным доступом.	6	1
4.	Структурная организация микропроцессорной системы. Архитектура системы ввода/вывода. Способы организации передачи данных. Система непосредственного ввода/вывода. Система канального ввода/вывода. Программно-управляемый ввод/вывод. Прямой ввод/вывод. Условный ввод/вывод. Ввод/вывод с программным квитированием. Ввод/вывод по прерываниям.	3	1
5.	Понятие прерывания процессора. Организация подсистемы прерываний в МПС. Контекстное переключение. Организация радиальной системы прерываний. Метод поллинга. Организация векторной системы прерываний. Вектор прерывания.	3	1
6.	Прямой доступ к памяти. Организация прямого доступа к памяти. Контроллер ПДП.	1	1
7.	Память микропроцессорной системы. Функции памяти. Архитектура и иерархия памяти. Организация кэш-памяти. Виртуальная память.	6	1
	Модуль II. Универсальные микропроцессоры		
8.	Определение понятия «архитектура». Архитектура системы команд. Классификация процессоров CISC и RISC.	1	-
9.	Методы адресации и типы данных. Типы команд. Команды управления потоком команд.	1	-
10.	Конвейеризация и параллелизм. Конвейерная организация обработки данных. Простейшая организация конвейера и оценка его производительности.	1	-
11.	Структурные конфликты и способы их минимизации. Конфликты по данным, остановы конвейера и реализация механизма обходов. Сокращение потерь на выполнение команд перехода и минимиза-	3	1

	ция конфликтов по управлению.		
12.	Проблемы реализации точного прерывания в конвейере. Обработка многотактных операций и механизмы обходов в длинных конвейерах.	2	1
13.	Параллелизм на уровне выполнения команд, планирование загрузки конвейера и методика разворачивания циклов. Конвейерная суперскалярная обработка.	2	-
14.	Зависимости. Классификация зависимостей и их применение. Устранение зависимостей по данным и механизмы динамического планирования.	6	1
15.	Аппаратное прогнозирование направления переходов и снижение потерь на организацию переходов. Буфер прогнозирования переходов. Корреляционая схема. Другие методы сокращения приостановок по управлению.	4	1
16.	Одновременная выдача нескольких команд для выполнения и динамическое планирование.	4	1
17.	Архитектура машин с длинным командным словом (VLIW). Средства поддержки большой степени распараллеливания.	6	1
18.	Архитектура EPIC.	2	1
	Модуль III. Микроконтроллеры и специализированные микропроцессоры		
19.	Микроконтроллеры. 8-разрядные микроконтроллеры. 16- и 32-разрядные микроконтроллеры. Структурная организация. Построение микропроцессорной системы на базе микроконтроллеров. Особенности программирования.	4	1
20.	Специализированные микропроцессоры. Цифровые процессоры обработки сигналов.	2	1
	Модуль IV. Заключение		
21.	Перспективы развития микропроцессорной техники.	1	-
ВСЕГО:		**64**	**14**

[1] Студенты заочной формы обучения темы №№ 1, 2, 8, 9, 10, 13 изучают самостоятельно.
[2] Лекции первого модуля составлены на основании лекций доц. Качинского М. В., каф. ЭВС, БГУИР.

№ п/п	Наименования тем лабораторных работ и их содержание	Количество часов	
		ДП	ЗП
1	**2**	**3**	**4**
	Лабораторный курс		
	7 семестр		
1.	Суперскалярные вычисления. Параллельное выполнение вычислений.	4	-
2.	Оценка времени выполнения арифметических операций.	4	-
3.	Вычисления с использованием MMX регистров.	4	-
4.	Определение размеров кэш-памяти на основании вычисленного времени доступа.	4	-
	8 семестр		
5.	Основы работы с DSP процессором TMS320VC5510.	4	-
6.	Работа с DSP/BIOS для генерации звукового сигнала платой DSK5510.	4	-
7.	Цифровая фильтрация. Реализация фильтра с конечной импульсной характеристикой (КИХ).	4	-
8	Цифровая фильтрация. Реализация фильтра с бесконечной импульсной характеристикой (БИХ).	4	-
ВСЕГО:		**32**	**14**

ЛЕКЦИОННЫЙ КУРС

МОДУЛЬ 0. ВВЕДЕНИЕ

1. Цели и задачи курса

Предметом дисциплины являются микропроцессоры (МП) и микропроцессорные семейства БИС (аппаратные средства).

За 30 лет своего существования МП из скромных приборов превратились в поражающие своим многообразием кристаллы, применяемые практически повсеместно – от бытовых приборов до суперкомпьютеров. Сегодня всеобщее внимание приковано к МП, предназначенным для использования в персональных компьютерах (ПК), ведь именно благодаря ним фирма Intel, некогда небольшая компания, стала крупнейшим в мире производителем полупроводниковых приборов.

Расширение сферы применения МП неизбежно привело к специализации кристаллов в зависимости от решаемых задач. Как правило, в универсальные центральные процессоры (ЦП), помимо кэш-памяти и шинных интерфейсов, интегрируется не так много специальных блоков. Но устройства для выполнения операций с плавающей точкой и управления памятью стали для них стандартными компонентами.

В микроконтроллерах (МК), напротив, последние два модуля обычно отсутствуют, зато достаточно полно реализованы некоторые периферийные средства. Такое техническое решение преследует одну цель – свести к минимуму стоимость системы. Это приводит к еще большей специализации кристаллов и росту числа их классов, отличающихся не только набором команд и производительностью, но и составом периферийного оборудования, размещенного на чипе.

Наибольшей специализацией отличаются процессоры цифровой обработки сигналов (Digital Signal Processor, DSP). Рассчитанные на обработку в реальном времени аналоговых сигналов, преобразованных в цифровую форму, они имеют уникальную систему команд и ряд особенностей архитектуры, которые обеспечивают очень высокую производительность применительно к узкому кругу задач. Помимо классических DSP, в последнее время выпускаются специализированные микропроцессоры нового типа, получившие название медиа-процессоров. Они предназначены для обработки аудиосигналов, графики, видеоизображений, а также для решения ряда коммуникационных задач в мультимедиа ПК.

В настоящее время продается гораздо больше типов МП, чем можно было бы рассмотреть, хотя бы кратко, в одном курсе. Рынок предлагает порядка 50 типов кристаллов с различными системами команд, подтвердившими свою жизнеспособность, сотни различных реализаций и тысячи модификаций, незначительно отличающихся друг от друга.

Целью дисциплины является изучение основ организации и функционирования различных типов микропроцессоров и микропроцессорных БИС (МП БИС) и их программирования (программирование на уровне машинных команд, а не изучение языка программирования низкого уровня – Ассемблера – как такового).

Несмотря на то, что рынок ПК характеризуется ярко выраженной тенденцией к быстрой смене поколений МП (кристаллы устаревают в течение буквально двух-трех лет), многие ЦП оказались удивительными долгожителями. В свое время ПК совершили переход от 8-разрядных систем к 16-разрядным. Сегодня на рынке господствуют модели с длиной информационного слова в 32 разряда, а многие современные рабочие станции и серверы собраны на 64-разрядных кристаллах. Однако сегодня даже 4-разрядные кристаллы еще далеко не сошли со сцены, не говоря о 8-разрядных, которые по объемам продаж (в натуральном выражении) лидируют на рынке встраиваемых контроллеров. Благодаря успеху ПК доминирующее положение (по объему продаж) занимают процессоры с системой команд x86 фирмы Intel. Среди микроконтроллеров прочно разместились МК фирм Motorola, Hitachi и некоторых других. Среди DSP процессоров лидирующее положение занимают приборы фирмы Texas Instruments.

В курсе рассматриваются универсальные 8- и 16-разрядные МП и МК фирм Intel, Motorola и Zilog, их архитектура, функционирование и взаимодействие с внешними устройствами.

В результате изучения дисциплины студенты должны знать:

- принципы организации, основы построения и функционирования различных типов микропроцессоров и микропроцессорных БИС;
- принципы организации микропроцессорных систем;
- основы программирования микропроцессоров.

ЗАМЕЧАНИЕ. Несмотря на то, что в курсе рассматриваются конкретные МП БИС, в то же время не стоит задача дать технически исчерпывающие описания отдельных микросхем. Изучение данного курса не освобождает от необходимости использования более подробной технической документации при проектировании систем на базе этих микросхем.

2. Основные понятия и определения

Микропроцессор – функционально законченное программно-управляемое устройство, предназначенное для обработки данных и управления процессом этой обработки, выполненное в виде одной или нескольких БИС.

Микропроцессорная система, или микросистема (МС) – цифровое устройство или цифровая система (система обработки данных, контроля и управления), построенная на базе одного или нескольких МП. Программно-аппаратный принцип построения МС – один из основных принципов их

организации. Этот принцип заключается в том, что реализация целевого назначения МС достигается не только аппаратными средствами, но и с помощью программного обеспечения.

Микропроцессорная БИС – интегральная микросхема, выполняющая функцию МП или его части. По существу – это БИС с процессорной организацией, разработанная для построения микросистем. МП БИС относятся к особому классу микросхем, одной из особенностей которого является возможность программного управления БИС с помощью определенного набора команд.

Кроме МП БИС существуют также интегральные схемы, выполняющие функции памяти и интерфейсов периферийных устройств. Следовательно, микроЭВМ можно представить в виде БИС микропроцессора, памяти и периферийных устройств (рис. 2). Такая ЭВМ называется микроЭВМ.

Рис. 1. Структура микроЭВМ

МикроЭВМ – система обработки данных, содержащая одну или несколько МП БИС, кристаллы памяти (ПЗУ и ОЗУ), интерфейсов периферийных устройств, а также некоторые другие схемы.

Микропроцессорный комплект (семейство, набор) (МПК) – совокупность МП и других микросхем, совместимых по конструктивно-технологическому исполнению и предназначенных для совместного применения при построении МП, микроЭВМ и микросистем.

Все устройства микроЭВМ (ЦП, память, интерфейсы) можно объединить в один чип. Такая микроЭВМ называется однокристальной.

Однокристальная микроЭВМ – микроЭВМ, выполненная в виде одной БИС.

Однокристальные микроЭВМ широко используются для управления различной аппаратурой и оборудованием, например в бытовых приборах. Такие микроЭВМ называются микроконтроллерами.

Микроконтроллер – однокристальная микроЭВМ с небольшими вычислительными ресурсами и упрощенной системой команд, ориентированная на выполнение процедур логического управления различным оборудованием

(а не на производство вычислений). Особенностью микроконтроллеров является расширенная реализация периферийных средств на кристалле.

Размещая на одной плате несколько кристаллов, можно получить микроЭВМ с достаточно большими вычислительными ресурсами. Такая микроЭВМ называется одноплатной микроЭВМ.

Одноплатная микроЭВМ – микроЭВМ, выполненная в виде одной печатной платы и предназначенная для встраивания в различную радиоэлектронную аппаратуру.

Микропроцессорные средства – МПК БИС, однокристальные и одноплатные микроЭВМ.

3. Микропроцессор и микропроцессорная система

ЭВМ состоит из ЦП, памяти и периферийных устройств. Эти составные части ЭВМ соединяются между собой множеством сигнальных проводов, называемым шиной. ЦП представляет собой устройство обработки данных, а память – устройство для их хранения. Периферийные устройства можно разделить на устройства ввода/вывода и внешнюю память. Эти устройства подсоединяются к шине с помощью соответствующих интерфейсов (контроллеров, устройств сопряжения) (рис. 1). Поэтому ЭВМ можно представить в виде собственно вычислительной машины и периферийных (внешних) устройств. Часто под ЭВМ понимают именно саму вычислительную машину (в данном курсе ЭВМ рассматривается именно в таком смысле).

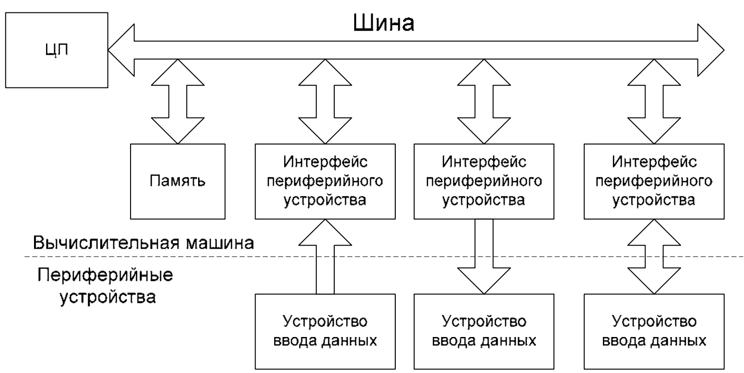


Рис. 2. Общая структура ЭВМ

Основным устройством ЭВМ является ЦП, имеющий наиболее сложную структуру. ЦП, реализованный в виде одной или нескольких БИС, называется микропроцессором.

4. Характеристики микропроцессоров

Архитектура МП ориентирована на достижение универсальности применения, высокой производительности и технологичности.

Универсальность (возможность разнообразного применения) МП определяется их широкими функциональными возможностями и обеспечивается:

– программным управлением, позволяющим производить программную настройку МП на выполнение определенных функций;
– гибкой системой команд и разнообразием способов адресации;
– магистрально-модульным принципом построения;
– специальными аппаратно-логическими средствами: регистровая память, виртуальная память, многоуровневая система прерываний, прямой доступ к памяти и т. п.

Относительно высокая производительность МП достигается использованием для их построения быстродействующих БИС и СБИС и специальных архитектурных решений, таких как регистровая память, кэш-память, конвейерная обработка, суперскалярная архитектура, предсказание переходов, механизм динамического выполнения команд и т. п.

Технологичность микропроцессорных средств обеспечивается модульным принципом конструирования, который предполагает реализацию этих средств в виде набора функционально законченных БИС, просто объединяемых в соответствующие вычислительные устройства, машины и системы.

Кроме перечисленных выше трех основных особенностей, исключительно широкое применение МП в различных цифровых устройствах и системах обеспечивается также:

– низкой стоимостью;
– небольшими размерами;
– малой мощностью потребления;
– высокой надежностью и большой устойчивостью к неблагоприятным внешним воздействиям.

МП характеризуется очень большим числом параметров и качеств, поскольку он, с одной стороны функционально является сложным программно-управляемым цифровым процессором, т. е. устройством ЭВМ, а с другой – БИС, т. е. электронным прибором. Поэтому для МП важны такие качества и параметры, как:

– тип микроэлектронной технологии, проектные нормы и число слоев металлизации (проектные нормы определяют минимальные топологические размеры элементов, что, в свою очередь, определяет количество транзисторов, размещаемых на кристалле МП, и максимальную рабочую частоту МП);
– количество кристаллов, образующих МП;
– площадь кристалла и количество транзисторов на кристалле;
– тип корпуса;
– разрядность МП;
– быстродействие МП (рабочая частота; число одновременно декодируемых инструкций; число команд, запускаемых на выполнение за один такт; время выполнения команд);

– размер адресуемой памяти;
– наличие и размер кэш-памяти;
– наличие арифметического сопроцессора;
– число входящих в микропроцессорный набор дополнительных БИС и выполняемые ими функции;
– система команд (количество команд, выполняемые операции, способы адресации, наличие команд обработки бит, чисел с плавающей запятой, десятичной арифметики);
– форматы данных;
– типы и число уровней прерывания;
– возможность прямого доступа к памяти;
– пропускная способность интерфейса ввода/вывода (частота и разрядность системной шины);
– количество и уровни питающих напряжений;
– требования к синхронизации;
– параметры используемых сигналов;
– потребляемая мощность;
– помехоустойчивость;
– нагрузочная способность;
– надежность и т. д.

5. Классификация микропроцессоров

По числу кристаллов, образующих МП, различают МП:
– однокристальные с фиксированной разрядностью и системой команд;
– многокристальные с фиксированной разрядностью и системой команд;
– многокристальные с разрядно-модульной организацией (секционные микропрограммируемые).

Однокристальные МП получаются при реализации всех аппаратных средств процессора в виде одной БИС или СБИС (на одном кристалле).

Многокристальный МП получается путем разбиения структуры процессора на функционально законченные части и реализации их в виде отдельных БИС или СБИС (нескольких кристаллов). Функциональная законченность БИС многокристального МП означает, что его части выполняют заранее определенные функции и могут работать автономно, а для построения полного процессора не требуется организации большого количества новых связей и каких-либо других микросхем. Например, все аппаратные блоки процессора ЭВМ можно распределить между тремя основными функциональными частями: операционной, управляющей и интерфейсной и реализовать МП в виде трех кристаллов (рис. 3, а).

Многокристальные разрядно-модульные МП получаются в том случае, когда в виде БИС реализуются части (секции) логической структуры процессора при функциональном разбиении ее вертикальными плоскостями (рис. 3, б). Многоразрядный МП реализуется параллельным включением микропроцессорных секций с помощью дополнительных средств стыковки. При этом требуется большое количество дополнительных аппаратных средств, не реализуемых в доступных БИС. Поэтому, как правило, логическую структуру МП разбивают еще горизонтальными плоскостями. В результате микропроцессорная секция – это БИС, предназначенная для обработки нескольких разрядов данных (как правило 2, 4, 8) или выполнения определенных управляющих операций. Секционность МП определяет возможность наращивания разрядности обрабатываемых данных или усложнения устройства управления МП при параллельном включении большого числа БИС. Для обеспечения заданной разрядности обрабатываемых слов микропроцессор составляется из соответствующего количества одинаковых кристаллов МП секций, объединенных микропрограммным управляющим блоком, реализованным на отдельных БИС. Микропрограммируемые многокристальные МП обеспечивают большую гибкость в достижении нужных пользователю характеристик проектируемого МП устройства или системы, предоставляя возможность задавать специализированную систему команд, ориентированную на определенное применение, даже на определенные процедуры обработки данных. Однако при этом пользователь должен разработать микропрограммы, реализующие эти команды, и занести их в управляющую память МП. Использование микропрограммируемых МП связано с определенными трудностями, требует от разработчика довольно высокой квалификации в вопросах проектирования вычислительных средств. Поэтому наиболее широко распространенными микропроцессорами являются различные варианты МП с фиксированной разрядностью и системой команд.

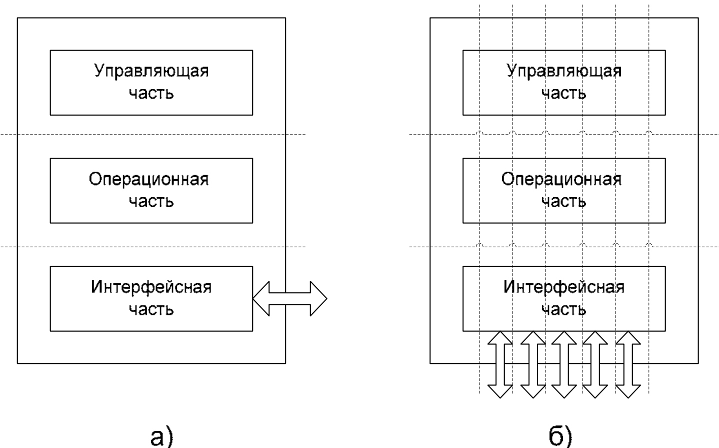


Рис. 3. Многокристальный микропроцессор:
а) МП с фиксированной разрядностью б) разрядно-модульный МП

По назначению различают МП:

– универсальные;

– специализированные.

Универсальные МП – такие МП, в архитектуре набора команд которых заложена алгоритмическая универсальность. Алгоритмическая универсальность означает, что выполняемый МП набор команд позволяет получить преобразование информации в соответствии с любым заданным алгоритмом. Универсальные МП могут быть применены для решения широкого круга разнообразных задач. При этом их эффективная производительность слабо зависит от проблемной ориентации решаемых задач.

Специализированные МП предназначены для решения определенного класса задач или одной конкретной задачи. Проблемная ориентация таких МП направлена на ускоренное выполнение определенных функций и позволяет резко увеличить эффективную производительность при решении только определенных задач. Примером таких МП являются процессоры для цифровой обработки сигналов – DSP процессоры (Digital Signal Processor) или ЦСП (цифровые сигнальные процессоры).

По типу набора команд (по типу архитектуры набора команд) различают МП:

– реализованные на базе архитектуры, называемой компьютером со сложным (полным, расширенным) набором команд (CISC – Complex Instruction Set Computer) – МП с CISC-архитектурой или CISC-микропроцессоры;

– реализованные на базе архитектуры, называемой компьютером с сокращенным набором команд (RISC – Reduced Instruction Set Computer) – МП с RISC-архитектурой или RISC-микропроцесссоры.

CISC-процессоры поддерживают широкий набор реализованных аппаратно (жестко «зашитых») инструкций, включая сложные команды. Многие из них, например команду умножения, принципиально трудно «прошивать»: для их реализации требуется большое количество транзисторов, а для выполнения – значительное число тактов. Состав и назначение регистров таких МП существенно неоднородны, широкий набор команд усложняет декодирование инструкций, на что расходуются дополнительные аппаратные ресурсы и такты. В результате в CISC-процессорах число тактов, необходимое для выполнения команд, относительно велико.

RISC-процессоры – это МП, имеющие меньший и более простой набор команд. В архитектуре данного типа отсутствуют сложные инструкции. Сложные инструкции создаются из более простых, реализуются не в аппаратном, а в программном обеспечении с помощью компилятора, который преобразует программу на языке высокого уровня в программу, состоящую из простых команд. В результате уменьшается количество аппа-

ратных средств, необходимых для декодирования и реализации сложных команд, что позволяет снизить сложность процессора и его стоимость, повысить эффективность его работы, скорость выполнения программы.

Отличительными чертами RISC-архитектуры являются:

– сокращенный набор относительно простых команд;

– однородный набор регистров универсального назначения;

– уменьшенный фиксированный формат команды;

– большинство операций имеет характер регистр-регистр, а обращения к памяти происходят только для выполнения простых операций загрузки в регистры и занесения в память;

– относительная простота процессора делает возможным размещение на кристалле большего числа регистров;

– высокая степень конвейеризации вычислений, что позволяет выполнять большинство команд набора за один такт.

По виду обрабатываемых входных сигналов различают МП:

– цифровые;

– аналоговые.

Сами МП – цифровые устройства, однако они могут иметь встроенные аналого-цифровые (АЦП) и цифро-аналоговые (ЦАП) преобразователи. В этом случае входные аналоговые сигналы передаются в МП через АЦП в цифровой форме, обрабатываются и после обратного преобразования ЦАП в аналоговую форму поступают на выход. С архитектурной точки зрения такие МП представляют собой аналоговые функциональные преобразователи сигналов и называются аналоговыми МП.

По характеру временной организации работы различают МП:

– синхронные;

– асинхронные.

В синхронных МП начало и конец выполнения операций задаются устройством управления с помощью сигналов тактовой частоты. В асинхронных МП начало выполнения каждой следующей операции определяется по сигналу фактического окончания выполнения предыдущей операции.

6. Эволюция микропроцессоров

Своему появлению микропроцессор обязан фирме Intel. Фирма Intel (от Integrated Electronics – интегральная электроника) основана 18 июля 1968 г. Гордоном Муром (Gordon Moore) и Бобом Нойсом (Bob Noyce). Первой идеей нового предприятия было создание полупроводниковых ЗУ, призванных заменить ЗУ на магнитных сердечниках. Поскольку к концу 60-х годов память этого типа практически исчерпала весь свой потенциал

развития, проблема была весьма актуальной, а ее решение сулила немалые прибыли. И хотя в данной области Intel добилась заметных успехов, тем не менее мировую славу ей принесли совсем другие изделия.

Переломным моментом в истории фирмы Intel стал 1969 г., когда ныне уже не существующая японская фирма Busicom предложила Intel разработать серию специализированных микросхем для калькуляторов нового образца. В те времена логические микросхемы, предназначенные для построения процессоров ЭВМ, разрабатывались исключительно на заказ, под конкретную продукцию конкретного производителя, что чрезвычайно ограничивало сферу применения любой такой микросхемы. Инженеры, работающие в фирме Busicom, создали проект (руководитель проекта Масатоси Шима), в котором предполагалось использовать 12 подобных уникальных микросхем с числом транзисторов в каждой от 3 до 5 тысяч. Перед этим фирма Intel освоила технологию производства микросхем, содержащих 2000 транзисторов. Работая над проектом для Busicom, инженеры фирмы Intel (ведущий специалист Тед Хофф) отвергли столь неуклюжую конструкцию, предложив революционную концепцию проектирования логических микросхем: расположить на одном-единственном кристалле логическое устройство общего назначения, считывающее прикладные команды из полупроводникового ЗУ. Если заказчикам требуется изделие с конкретными свойствами, они могут его запрограммировать, а фирма Intel запишет эту программу в ПЗУ. В этом случае фирме уже не придется разрабатывать для каждого заказчика новый набор логических схем. Разработанное устройство, явившееся ядром набора из четырех микросхем и получившее название центрального процессора, не только отвечало всем требованиям, предъявляемым к калькулятору Busicom, но и без каких-либо особых модификаций могло найти применение в других аналогичных приборах. Осознав громадный потенциал своей новой разработки, Intel предложила вернуть компании Busicom ее первоначальные инвестиции в размере 60 тысяч долларов в обмен на все права на данную продукцию. Японская фирма, столкнувшаяся к тому времени с финансовыми затруднениями, предложение приняла.

В конце 1970 г. состоялась официальная презентация вычислительного микроустройства 4004 (термин «микропроцессор» появился позднее). 15 ноября 1970 г. фирма Intel приступила к поставкам первого в мире микропроцессора – Intel 4004. Кристалл представлял собой 4-разрядный процессор с классической архитектурой ЭВМ гарвардского типа и изготавливался по передовой в те года р-канальной МОП-технологии с проектными нормами 10 мкм. МП 4004 содержал 2300 транзисторов, размещенных на кристалле площадью 3,2Ч4,3 мм, имел весьма ограниченные возможности

адресации памяти (1280 полубайтов данных и 4 Кбайт команд) и невысокое быстродействие (30000 операций в секунду). Микропроцессор работал на тактовой частоте 750 кГц при длительности цикла команды 10,8 мкс. МП 4004 слишком маломощен, чтобы выполнять функции процессора, однако вполне может служить основой калькулятора. Одна БИС МП 4004 заменила сотни схем, составляющих в то время элементную базу промышленных ЭВМ. Система из 46 команд кажется скромной по сегодняшним меркам, однако она вполне адекватна для решения задач управления с принятием решения, которые не помещаются в программируемые логические матрицы. Поэтому МП 4004 получил применение в простых системах управления и играх.

Разрабатывая первый в мире микропроцессор, специалисты Intel с самого начала позаботились о простоте и удобстве построения систем на базе i4004. Компанией был разработан и выпущен не один кристалл центрального процессора, а целое семейство БИС, в которое вошли ПЗУ 4001, ОЗУ 4002, регистр сдвига 4003 и ряд других вспомогательных микросхем. Поскольку все они были рассчитаны на совместное использование, разработка аппаратных средств системы заметно упрощалась, и это стало не последней причиной популярности i4004.

Успех МП 4004 послужил для фирмы Intel стимулом для разработки в 1972 г. первого 8-разрядного МП 8008. В отличие от своих предшественников новый МП имеет архитектуру ЭВМ принстонского типа. Он включает 3500 транзисторов, выполняет свыше 60000 операций в секунду, адресует 16 Кбайт памяти, а также имеет внутренний стек емкостью 8 байт и одноуровневую систему прерываний. МП работает на частоте 800 кГц при длительности командного цикла 12,5 мкс. Количество команд увеличено до 65. Однако системы команд МП 8008 и 4004 несовместимы. МП 8008 уже имел достаточную мощность, чтобы служить ЦП микроЭВМ. На протяжении первой половины 70-х годов фирма Intel продолжала убеждать инженеров-разработчиков, что «ЭВМ на кристалле» является реальной альтернативой традиционным устройствам с жесткими связями.

В 1974 г. фирма Intel выпустила более совершенный 8-разрядный МП 8080, который был совместим снизу-вверх с МП 8008. Он содержал более 4500 транзисторов и выпускался по n-МОП технологии с проектными нормами 6 мкм. МП адресует память емкостью 64 Кбайт и имеет средства работы со внешним стеком. МП 8080 в течение более 10 лет являлся мировым стандартом среди 8-разрядных микропроцессоров. К 1976 г., когда эта фирма выпустила МП 8085 (8-разрядный МП, для которого требовался один источник питания +5 В и меньшая номенклатура вспомогательных ИС) на рынке уже имелся большой выбор 8-разрядных МП. В число

конкурирующих изделий входили МП Z80 фирмы Zilog, MSC6502 фирмы MOS Technology, MC6800, а в последствии 6809 фирмы Motorola. Некоторые из этих МП были изготовлены в огромных количествах, что привело к снижению цен и способствовало применению в потребительских товарах, в том числе в домашних компьютерах.

Для построения управляющих устройств и других прикладных применений разрабатываются однокристальные микроЭВМ, содержащие в своем составе небольшие ОЗУ и ПЗУ: 8048 фирмы Intel, 6801 фирмы Motorola, Z8 фирмы Zilog и TMS9940 фирмы Texas Instruments.

Однокристальные МП выполняются с использованием МОП- (MOS-) технологий, позволяющих размещать на одном кристалле большое число элементов – МОП-транзисторов. Однако МОП-структуры, которые использовались в первых поколениях МП, существенно уступали в быстродействии биполярным структурам. Биполярные БИС (например, маломощные ТТЛ-схемы с диодами Шотки) обладали по сравнению с МОП-кристаллами намного большим быстродействием, но значительно меньшей плотностью упаковки элементов на кристалле. Появился второй тип МП – многокристальный биполярный разрядно-модульный микропрограммируемый МП, основанный на конструктивном принципе функционально-разрядного слоя, предполагающем реализацию на кристалле малоразрядной МП секции. В этом случае для обеспечения заданной разрядности обрабатываемых слов МП составляется из соответствующего количества одинаковых кристаллов МП секций, объединяемых микропрограммным управляющим блоком, реализованным на отдельных кристаллах. Наиболее известное семейство разрядно-модульных МП – Am2900 фирмы Advanced Micro Devices (AMD). МП этого типа обеспечивают большую гибкость в достижении нужных пользователю характеристик проектируемой микросистемы, предоставляя пользователю возможность задавать специализированную систему команд, ориентированную на определенное применение. Однако при этом пользователь должен разработать микропрограммы, реализующие эти команды, и поместить их в управляющую память МП. Использование микропрограммируемых МП связано с определенными трудностями, требует от разработчика довольно высокой квалификации в вопросах ВТ. Поэтому дальнейшее развитие микропроцессоров идет по пути разработки различных вариантов однокристальных МП с фиксированной системой команд.

В 1977 г. появились 16-разрядные МП. Первые из них – PAGE фирмы National Semiconductor, CP1600 фирмы GIM и TMS9900 фирмы Texas Instruments – имели лишь несколько большее быстродействие по сравнению с 8-разрядными МП, но более поздние 16-разрядные МП, появляв-

шиеся начиная с 1978 г., имели гораздо более высокую производительность. Быстродействие МП 8086 фирмы Intel в десять раз больше, чем у i8080, а последующие 16-разрядные МП, такие как MC68000 фирмы Motorola, Z8000 фирмы Zilog, TMS99000 фирмы Texas Instruments, NS16032 фирмы National Semiconductor, MC68010 фирмы Motorola и 80286 фирмы Intel, имели все более высокое быстродействие по мере совершенствования полупроводниковой технологии и внутренней архитектуры.

Первые 32-разрядные МП появились в 1981 г. Это были МП Focus фирмы Hewlett Packard и WE32000 фирмы AT&T. В 1983 г. появились МП NS32032 фирмы National Semiconductor и NCR/32 фирмы National Cash Register. В 1985 г. рынок был заполнен 32-разрядными МП. В начале этого года появился MC68020 фирмы Motorola, а к его окончанию МП T414 фирмы Inmos и 80386 фирмы Intel. Впоследствии в 1986 г. был выпущен МП Z80000 фирмы Zilog.

В дальнейшем развитие микропроцессорной техники неразрывно связано с компанией Intel, которая сохраняет до сегодняшнего дня лидерство в этой динамичной и высокотехнологичной области науки и техники.

15 ноября 2000 г. состоялась презентация Pentium 4 – новой модели 32-разрядных микропроцессоров компании Intel. Тем самым был положен конец сомнениям по поводу дальнейшего развития этого семейства. В связи с появлением в 2000 г. нового семейства высокопроизводительных 64-разрядных процессоров Itanium, многие специалисты считали, что долгий век семейства процессоров 80x86 – Pentium, начатого в 1978 г. моделью 8086 и представляемого в настоящее время последними моделями Pentium III, Pentium III Xeon, Celeron, близится к окончанию. Однако выпуск Pentium 4 показал, что это не так. В процессоре Pentium 4 сохраняется архитектура IA-32 (Intel Architecture – 32), характерная для всех 32 - разрядных микропроцессоров Intel, начиная с i386. Поэтому пользователь имеет дело с хорошо знакомым набором регистров и способов адресации, работает с базовой системой команд и известными вариантами реализации прерываний и исключений. Однако внутренняя структура (микроархитектура) процессора Pentuim 4 значительно отличается от предшествующей микроархитектуры семейства P6, к которому относятся Pentium II, Pentium III и Celeron.

Перечисленными процессорами не исчерпывается весь мировой ассортимент микропроцессоров. Это только представители семейства процессоров, имеющих обобщенное название x86. Ряд фирм (например, AMD, Cyrix, IBM) выпускает процессоры, совместимые с перечисленными процессорами Intel и имеющими свои характерные особенности. Обычно они слегка отставали от изделий Intel, выпускаемых в то же время. Однако семейство процессоров K7 фирмы AMD (Duron, Athlon) изменило ситуацию.

МОДУЛЬ I. ОРГАНИЗАЦИЯ МИКРОПРОЦЕССОРНОЙ СИСТЕМЫ

1. Основные типы архитектур микропроцессорных систем. Фон-неймановская (принстонская) и гарвардская архитектуры. Организация пространств памяти и ввода/вывода

Под **организацией** понимают состав компонентов (аппаратных или программных средств), связи между ними и их функциональные характеристики.

ЭВМ имеет многоуровневую иерархическую организацию со своими составными компонентами на каждом уровне:

1) нижний уровень − уровень физических компонентов − физическая организация (представляется в виде принципиальной схемы);

2) уровень реализуемых в ЭВМ функций − логическая (функциональная) организация (представляется в виде функциональной схемы);

3) верхний уровень − уровень аппаратуры (состав, функциональные связи и характеристики аппаратных модулей) − структурная организация (представляется в виде структурной схемы).

Все компоненты микропроцессорной системы представляются для процессора в виде набора ячеек памяти или портов ввода/вывода, которые образуют два основных пространства: соответственно пространство памяти и пространство ввода/вывода.

Фон-неймановская (принстонская) и гарвардская архитектуры. В большинстве современных микропроцессорных систем для хранения программ и данных используется общая шина памяти. Такая организация получила название архитектуры Дж. фон Неймана, предложившего кодирование программ в формате, соответствующем формату данных. ЭВМ с такой архитектурой называют машинами фон-неймановского или принстонского типа. В них области для хранения программ (Program Space – PS) и данных (Data Space – DS) образуют единое пространство и могут размещаться в любом месте общей памяти. При этом нет никаких признаков, указывающих на тип информации в ячейке памяти. Содержимое ячейки интерпретируется ЦП, и задача программиста − следить за тем, чтобы данные и программа обрабатывались по-разному. Фон-неймановская архитектура характерна для универсальных МП.

В специализированных МП и микроконтроллерах используется другая схема, известная как архитектура Гарвардской лаборатории или гарвардская архитектура. В ее классическом варианте программы и данные хранятся в двух раздельных памятях, что позволяет полностью совмещать во времени выборку и исполнение команд. ЭВМ, спроектированные в соответствии с концепцией разделения памяти на два вида, называют маши-

нами гарвардского типа. В таких системах память программ и память данных разделены и имеют свои собственные адресные пространства и способы доступа к ним. Программа находится всегда в одной памяти, а данные – в другой. Такое разделение позволяет повысить быстродействие и упростить схемотехническую реализацию микропроцессорной системы.

Дальнейшее совершенствование архитектур обоих типов состояло в выделении специального пространства данных небольшого объема, которое представляет собой набор программно-доступных регистров (Register Space). В отличие от памяти и портов ввода/вывода регистры располагаются всегда внутри МП вместе с АЛУ, что обеспечивает быстрый физический доступ к информации, хранящейся в них. В некоторые интервалы времени программа наиболее интенсивно работает лишь с небольшим объемом данных. Для временного хранения этих данных и предназначена регистровая область – набор программно доступных регистров.

Регистровая область может быть как полностью изолирована от пространства данных DS, так и частично пересекаться с ним, что дает возможность рассматривать отдельные регистры МП как обычные ячейки памяти данных. Такая организация является целесообразной, если в МП поддерживается быстрый доступ ко всей или хотя бы к некоторой части памяти данных.

Все современные МП имеют регистровые области независимо от того, к какому типу они принадлежат: принстонскому или гарвардскому. Внутренняя логическая организация регистровой области очень разнообразна и зависит от типа МП. Функциональная структура регистровой области будет рассмотрена позже. Пока отметим в ее составе лишь один регистр, который называется программным счетчиком PC (Program Counter). Данный регистр является обязательным для всех МП и связан с адресацией памяти программ. Он служит указателем следующего элемента программной последовательности, подлежащего выборке и исполнению.

Пространство ввода/вывода представляет набор адресуемых буферных схем и регистров, которые называются портами и через которые осуществляется связь с внешними и внутренними аппаратными средствами микропроцессорной системы.

В микропроцессорной системе может использоваться два варианта организации пространства ввода/вывода:

– изолированный ввод/вывод. Порты ввода/вывода размещены в специальном пространстве ввода/вывода (Input/Output Space – IOS), изолированном от других пространств данных. В этом случае МП имеет специальный набор команд ввода/вывода.

– совмещенный ввод/вывод или ввод/вывод с отображением на память. В этом случае изолированное пространство ввода/вывода отсутству-

ет, а в пространстве памяти данных DS выделяются области, в которых размещаются порты. Организация доступа к портам в такой микропроцессорной системе ничем не отличается от процесса обращения к данным в памяти.

На рисунке 4 представлены четыре типовых набора областей для хранения программ и данных. Стрелками показан процесс изоляции отдельных областей, приводящий к появлению нового типового набора. Все наборы существуют реально, каждый имеет свои преимущества и недостатки, учет которых позволяет создавать высокоэффективные системы различного назначения.

В отличие от регистровой области пространства памяти программ PS и данных DS, а также область ввода/вывода IOS организованы проще. Память представляет собой линейно упорядоченный набор n-разрядных ячеек с произвольным доступом (одномерный массив) − линейная память. Все ячейки пронумерованы, таким образом каждой ячейке набора соответствует число, называемое ее адресом. Все адреса занимают целочисленный диапазон от 0 до 2^m-1 (m − разрядность адреса), который образует адресное пространство памяти. В большинстве случаев процессор может адресоваться к памяти с точностью до одного байта, т. е. наименьшей адресуемой единицей является байт и память имеет байтовую организацию.

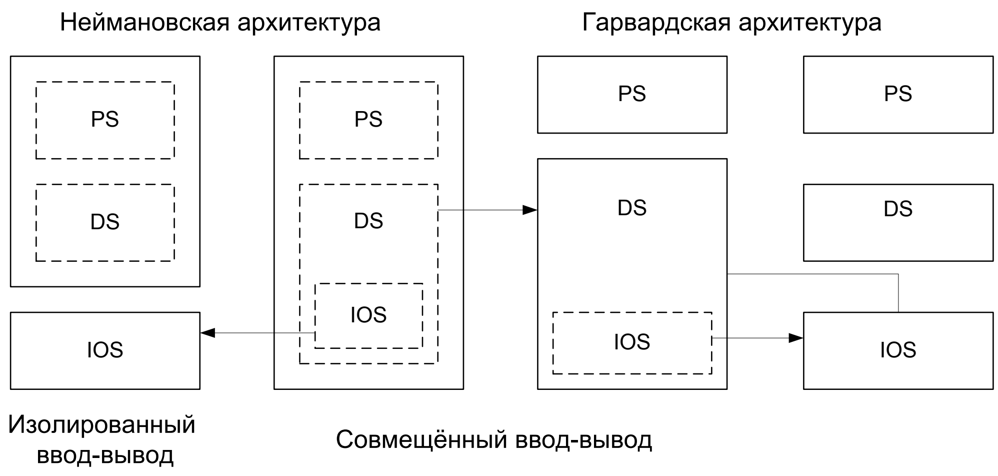


Рис. 4. Области микропроцессора для хранения программ и данных

Организация пространства памяти показана на рис. 5. При этом память изображается таким образом, чтобы ячейки со старшими адресами располагались ниже, чем с младшими. Нумерация отдельных разрядов в ячейке производится справа налево начиная с нуля, при этом разряд с нулевым номером является младшим.

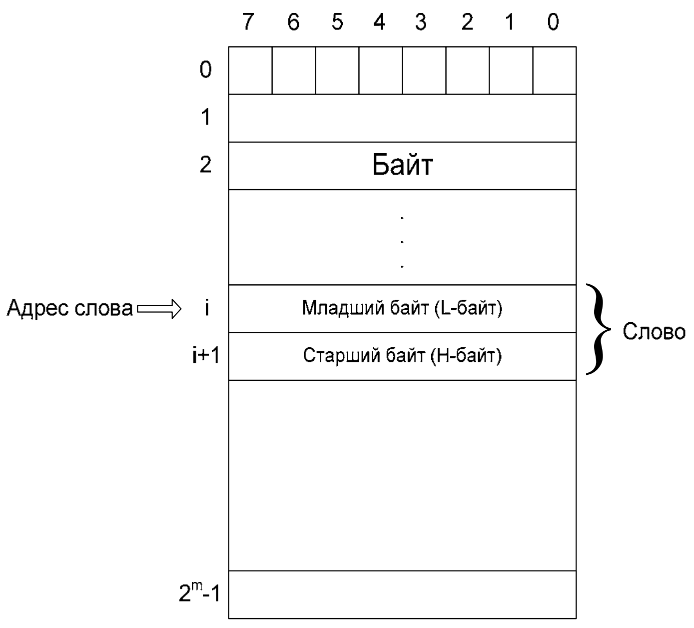


Рис. 5. Организация пространства памяти

Программные объекты (команды и операнды) могут иметь длину, превышающую один байт, например, два байта – 16-разрядное слово или просто слово, четыре байта – 32-разрядное слово или двойное слово, восемь байтов – 64-разрядное слово или учетверенное слово. Такие объекты располагаются в смежных ячейках пространства памяти, причем обычно младший байт размещается в ячейке с меньшим адресом. Адресом объекта служит наименьший из адресов ячеек, занимаемых им, т. е. в данном случае адрес его младшего байта. Такой порядок называется Little-Endian Memory Format. Он используется в микропроцессорах с архитектурой x86. В других семействах процессоров применяют и обратный порядок – Big-Endian Memory Format, в котором объекты располагаются в смежных ячейках памяти, начиная со старшего байта, а младшие байты размещаются в последующих ячейках (например, в микроконтроллерах семейства 68HC11 фирмы Motorola). В этом случае адресом объекта является адрес его старшего байта. Для взаимного преобразования форматов объектов в процессорах имеются специальные команды. Операция обращения к памяти предполагает считывание или запись всего объекта как единого целого. Например, 16-разрядные слова в памяти хранятся в двух соседних ячейках. Старший байт слова занимает ячейку с большим адресом, а младший − ячейку с меньшим адресом. При этом адрес младшего байта служит адресом слова (см. рис. 5).

Часто организация памяти предусматривает определенные ограничения на возможное расположение многобайтовых объектов. Например, слова в памяти могут находиться только по четным адресам. Тогда при досту-

пе к слову значение младшего разряда его адреса, указывающего на байт в слове, во внимание не принимается, т. е. такая память имеет границу слов.

Например, в МП 8086 любые два смежных байта в памяти могут рассматриваться как 16-разрядное слово. Таким образом, слова данных можно свободно размещать по любому адресу, что позволяет экономить память благодаря ее плотной упаковки. Однако для экономии времени выполнения программы целесообразно размещать слова данных в памяти по четным адресам, так как МП передает такие слова за один цикл шины. Слова с нечетными адресами (не выровненные) также допустимы, но для их передачи требуется уже два цикла шины, что снижает производительность МП. Особенно важно иметь выровненные слова для операций со стеком, так как в них участвуют только слова. Поэтому указатель стека SP необходимо всегда инициализировать на четный адрес. Команды в МП 8086 всегда выбираются словами по четным адресам, за исключением первой выборки после передачи управления по нечетному адресу, когда выбирается один байт. Поток команд разделяется на байты внутри МП, так что выравнивание команд не влияет на производительность и поэтому не используется.

В процессорах фирмы Intel, начиная с 486, на уровне привилегий 3 может быть включен контроль выравнивания операндов по соответствующей границе: слова по четному адресу, двойного слова по адресу, кратному четырем, учетверенного слова по адресу, кратному восьми. На уровнях привилегий 0, 1, 2 контроль выравнивания не производится.

Рассмотренная организация памяти соответствует нижнему (физическому) уровню представления памяти. Пространство ввода/вывода имеет такую же организацию. Существует более высокий (логический) уровень организации памяти, на котором работает программист и который связан с архитектурой процессора.

Магистрально-модульный принцип организациимикропроцессорной системы. Большинство современных микропроцессорных систем построено по магистрально-модулъному принципу. В соответствии с этим принципом память и подсистема ввода/вывода выполняются в виде отдельных функционально законченных модулей, которые подключаются к единой внутрисистемной магистрали.

В подсистеме памяти выделяют модули постоянных запоминающих устройств (ПЗУ), которые используются для хранения программ и констант, модули оперативных запоминающих устройств (ОЗУ), предназначенных для хранения переменных и загружаемого извне программ.

В составе подсистемы ВВ в простейшем случае выделяются адресуемые МП буферные схемы и регистры – порты ввода/вывода. Они предназначены для связи с простыми внешними устройствами, такими как светодиодные индикаторы, переключатели и т. п. Более сложные модули подсистемы ввода/вывода, предназначенные для управления внешним интер-

фейсным оборудованием и реализации специальных функций ввода/вывода, строятся на основе портов ввод/вывод и называются адаптерами или контроллерами периферийных устройств.

Наиболее сложными из модулей подсистемы ввода/вывода являются процессоры (сопроцессоры) ввода/вывода, которые работают по собственным программам, хранящимся в памяти, и по сути дела представляют собой отдельные микропроцессорные системы.

В зависимости от способа подключения отдельных модулей микропроцессорной системы к системной магистрали различают три типовые структуры микропроцессорных систем:

– магистральная;
– магистрально-каскадная;
– магистрально-радиальная.

В магистральной структуре все модули подсистем памяти и ввода/вывода подключаются непосредственно к системной магистрали. Это наиболее простая структура. Недостатками магистральной структуры являются:

– все модули должны поддерживать протокол обмена по системной магистрали и содержать средства сопряжения с ней, которые в зависимости от микропроцессора могут быть достаточно сложными;
– небольшое быстродействие, т. к. медленные периферийные устройства могут надолго занимать системную магистраль.

В магистрально-каскадной и магистрально-радиальной структурах отдельные модули подключаются с помощью специальных контроллеров (адаптеров) шин, основное назначение которых – реализовать приоритетные соотношения при использовании магистрали.

В магистрально-каскадной структуре отдельные модули подключаются к контроллеру шины с помощью дополнительного общего канала, например, магистрали или шины ввода/вывода, т. е. по магистральной схеме. В магистрально-радиальной структуре каждый модуль подключается к контроллеру шины с помощью индивидуального канала, т. е. по радиальной схеме.

Архитектура с иерархией шин. В настоящее время примерно одинаковое распространение получили два способа построения вычислительных машин: с непосредственными связями и на основе шины.

В системах, построенных по первому способу, между взаимодействующими устройствами (процессор, память, устройство ввода/вывода) имеются непосредственные связи. Особенности связей (число линий в шинах, пропускная способность и т. п.) определяются видом информации, характером и интенсивностью обмена. Достоинством архитектуры с непосредственными связями можно считать возможность развязки узких мест путем улучшения структуры и характеристик только определенных связей, что экономически может быть наиболее выгодным решением. У фон-неймановских ЭВМ таким узким местом является канал пе-

ресылки данных между процессором и памятью, и развязать его достаточно непросто. Кроме того, системы с непосредственными связями плохо поддаются реконфигурации.

В варианте с общей шиной все устройства вычислительной машины подключаются к системной магистрали, служащей единственным трактом для потоков команд, данных и управления. Наличие общей шины существенно упрощает реализацию ЭВМ, позволяет легко менять состав и конфигурацию машины. Благодаря этим свойствам шинная архитектура получила широкое распространение в микроЭВМ. Вместе с тем, именно с шиной связан и основной недостаток архитектуры: в каждый момент передавать информацию по шине может только одно устройство. Основную нагрузку на шину создают обмены между процессором и памятью, связанные с извлечением из памяти команд и данных и записью в память результатов вычислений. На операции ввода/вывода остается лишь часть пропускной способности шины. Практика показывает, что даже при достаточно быстрой шине для 90% приложений этих остаточных ресурсов обычно не хватает, особенно в случае ввода или вывода больших массивов данных.

Поэтому при сохранении фон-неймановской концепции последовательного выполнения команд программы шинная архитектура в чистом ее виде оказывается недостаточно эффективной. Более распространена архитектура с иерархией шин, где помимо системной шины имеется еще несколько дополнительных шин. Они могут обеспечивать непосредственную связь между устройствами с наиболее интенсивным обменом, например процессором и кэш-памятью. Другой вариант использования дополнительных шин – объединение однотипных устройств ввода/вывода с последующим выходом с дополнительной шины на системную. Это позволяет снизить нагрузку на общую шину и более эффективно расходовать ее пропускную способность. Наибольшее распространение получили микропроцессорные системы с одной шиной, с двумя или тремя видами шин.

В структурах с одной шиной имеется одна системная шина, обеспечивающая обмен информацией между процессором и памятью, а также между устройствами ввода/вывода, с одной стороны, и процессором либо памятью – с другой. Для такого подхода характерны простота и низкая стоимость. Однако одношинная организация не в состоянии обеспечить высокую скорость обмена, причем узким местом становится именно шина.

Хотя контроллеры устройств ввода/вывода могут быть подсоединены непосредственно к системной шине, больший эффект достигается применением одной или нескольких шин ввода/вывода. Устройства ввода/вывода подключаются к шинам ввода/вывода, которые берут на себя основной обмен, не связанный с выходом на процессор или память. Подключение осуществляется с помощью адаптеров шин, которые обеспечивают буферизацию данных при их пересылке между системной шиной

и контроллерами устройств ввода/вывода. Это позволяет микропроцессорной системе поддерживать работу множества устройств ввода/вывода и одновременно развязать обмен информацией по тракту процессор-память и обмен информацией с устройствами ввода/вывода. Подобная схема существенно снижает нагрузку на скоростную шину процессор-память и способствует повышению общей производительности микропроцессорной системы.

Для подключения быстродействующих периферийных устройств в систему шин может быть добавлена высокоскоростная шина расширения. Шины ввода/вывода подключаются к шине расширения, а уже с нее через адаптер к шине процессор-память. Схема еще более снижает нагрузку на шину процессор-память. Такую организацию шин называют архитектурой с «пристройкой» (mezzanine architecture).

2. Магистраль микропроцессорной системы. Стандартная структура шины. Трехшинная магистраль с раздельными шинами передачи адреса и данных. Совмещение шины адреса и шины данных. Циклы обращения к магистрали. Временные диаграммы работы шины. Организация обращения к магистрали с синхронными асинхронным доступом

На физическом уровне МП взаимодействует с памятью и периферийными устройствами через единый набор системных шин − внутрисистемную магистраль. Физическое совмещение линий связи МП с памятью и периферийными устройствами было необходимым в связи с технологической особенностью производства БИС – ограниченным числом физических выводов, допустимых на кристалле. До середины 80-х годов стандартная микропроцессорная БИС имела 40 выводов, через которые необходимо было связываться как с памятью, так и с периферийными устройствами. В общем случае магистраль обеспечивает три вида передачи данных:

- процессор ↔ память;
- процессор ↔ интерфейс периферийного устройства;
- память ↔ интерфейс периферийного устройства (канал прямого доступа к памяти).

Рассмотрим первые два вида. В обоих случаях передачей данных управляет МП. Память и интерфейс по управляющим сигналам от процессора осуществляют передачу данных. Направление передачи данных определяется МП. Пересылка данных внутрь процессора называется считыванием (вводом), обратный процесс − записью (выводом).

В общем случае магистраль микропроцессорной системы состоит из набора шин. Шиной системы называют физическую группу линий передачи сигналов, имеющих схожие функции в рамках системы. Стандартная структура магистрали микропроцессорной системы включает три шины (рис. 6):

– шина данных DB (Data Bus);
– шина адреса (адресная шина) AB (Address Bus);
– шина управления CB (Control Bus).

Магистраль такого типа называется трехшинной с раздельными шинами передачи адреса и данных.

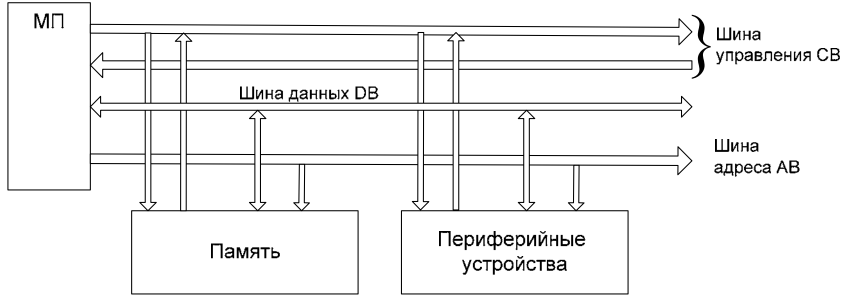


Рис. 6. Стандартная структура магистрали микропроцессорной системы

Шина данных. По этой шине передаются данные, т. е. производится обмен данными между МП и другими устройствами системы – памятью и периферийными устройствами. Шина данных используется как для передачи данных в направлении МП → память и МП → периферийное устройство, так и для их передачи в обратном направлении, т. е. шина данных является двунаправленной. Имеется возможность установки выходов в третье состояние. Хотя передача данных по шине данных может производиться в обоих направлениях, однако в каждый заданный момент времени она осуществляется лишь в одном направлении. Это означает, что для передачи данных в систему и их приема из системы микропроцессор переводится в соответствующий режим. Более того, по всем разрядам шины в каждый момент времени данные передаются лишь в одном направлении, т. е. в любой момент по всем линиям шины они могут либо только вводиться, либо только выводиться.

Обычно разрядность шины данных и длину слов, обрабатываемых в МП (разрядность машинного слова МП), выбирают одинаковыми. Однако в ряде случаев разрядность шины данных может быть меньше или больше разрядности машинного слова МП. Например, в микропроцессоре 8088 длина машинного слова составляет 16 разрядов, а шина данных 8-разрядная, и наоборот, МП Pentium является 32-разрядным процессором, его шина данных 64-разрядная.

Шина адреса. Используется для передачи физического адреса ячейки памяти или порта ввода/вывода, к которым осуществляется обращение. Эта шина предназначена для того, чтобы выбирать правильный тракт для электрического соединения в пределах микропроцессорной системы. Шина адреса является выходной по отношению к МП. Разрядность адресной шины определяет наибольшее число адресов, к которым может обращаться МП. Если

разрядность адресной шины МП равна m, то он способен адресовать пространство физической памяти и пространство ввода/вывода объемом 2^m.

Шина управления. Служит для передачи сигналов управления обменом данными через магистраль и работой микропроцессорной системы. Как правило, часть этих сигналов является выходными, а другая часть – входными сигналами. Однако некоторые линии шины управления могут быть двунаправленными. Линии шины управления объединяются в группы по функциональному назначению. Конкретный состав сигналов шины управления зависит от типа МП. Отметим наиболее типичные из них:

1) синхросигнал обеспечивает тактирование работы микропроцессора. Все события в системе привязываются к какому-либо фронту этого сигнала;

2) четность данных. Сигналы на этих двунаправленных линиях определяют четность данных, передаваемых по шине данных. Как правило, на каждый байт шины данных отводится отдельный сигнал. Внутренний модуль генерации/контроля четности данных микропроцессора формирует или проверяет сигналы четности. Четность понимается в том смысле, что соответствующий байт содержит четное количество единичных разрядов. Значения сигналов на этих линиях не влияет на ход выполнения программы. При обнаружении ошибки четности при вводе данных МП либо формирует специальный управляющий сигнал либо генерирует внутреннее прерывание;

3) сигналы определения цикла магистрали указывают тип выполняемого цикла магистрали. Они разделяют циклы записи и циклы чтения, циклы данных и циклы управления, циклы обращения к памяти и циклы ввода/вывода, а также некоторые другие;

4) сигналы управления магистралью определяют начало цикла магистрали, позволяют другим устройствам системы управлять передачей и разрядностью данных, завершением цикла магистрали;

5) сигналы управления состоянием процессора изменяют состояние МП в ходе выполнения или перед выполнением программы. Это необходимо для распределения функций управления магистралью между несколькими активными устройствами, при обработке прерываний, сбросе и инициализации.

Организация обмена по магистрали. Команды и данные передаются между МП и другими устройствами системы в ходе операции обмена, которая может включать один или несколько магистральных циклов, т. е. физический обмен через магистраль выполняется словами определенной разрядности в виде следующих друг за другом обращений к магистрали. Время осуществления одного считывания, записи, ввода или вывода называется циклом обращения к магистрали или просто циклом магистрали (циклом шины). За один цикл обращения к магистрали между МП, памятью или периферийным устройством передается одно слово. Таким обра-

зом, циклы магистрали обеспечивают доступ к пространству физической памяти и пространству ввода/вывода. Существует несколько типовых циклов магистрали. Основные циклы магистрали связаны с возможными операциями, выполняемыми в микропроцессорной системе. К ним относятся циклы чтения и записи. При совмещенном вводе/выводе по этим циклам осуществляется как обращение к памяти, так и к портам ввода/вывода. При изолированном вводе/выводе эти циклы разделяются на циклы обращения к памяти и цикла обращения к портам ввода/вывода. Поэтому в микропроцессорной системе с изолированным вводом/выводом выделяется четыре основных цикла:

– цикл чтения из памяти;
– цикл записи в память;
– цикл ввод из порта ввода;
– цикл вывода в порт вывода.

В случае гарвардской архитектуры вводится также цикл чтения памяти программ. Рассмотрим основные сигналы, связанные с выполнением приведенных выше циклов магистрали.

Основными сигналами являются сигналы двух типов:

– сигналы управления записью/чтением, связанные с обращением к памяти;

– сигналы управления записью/чтением (вводом/выводом), связанные с обращением к портам ввода/вывода.

Когда применяется изолированный ввод/вывод, передаются четыре управляющих сигнала (рис. 7):

– чтение данных из памяти MEMRD;
– запись данных в память MEMWR;
– ввод данных из порта ввода IORD;
– вывод данных в порт вывода IOWR.

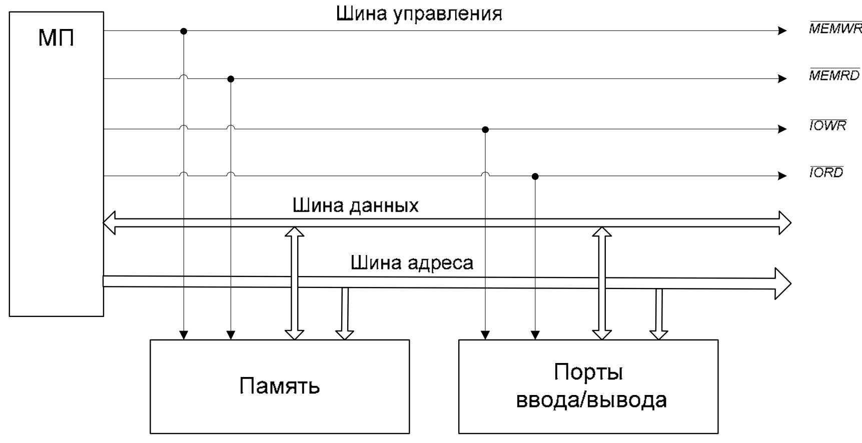


Рис. 7. Управляющие сигналы при изолированном вводе/выводе

Для гарвардской архитектуры добавляется сигнал чтение памяти программ PMEM.

Сигналы MEMRD, MEMWR, IORD, IOWR, PMEM являются как сигналами, определяющими цикл магистрали, так и управляющими синхронизирующими сигналами, показывающими, в какой интервал времени в цикле шины должна осуществляться соответствующая операция. Управляющие синхронизирующие сигналы также называются стробирующими сигналами. Эти сигналы делятся на два типа:

– сигналы, имеющие в обычном состоянии низкий уровень и в установленное время переходящие на высокий уровень;

– сигналы, имеющие в обычном состоянии высокий уровень и в установленное время переходящие на низкий уровень.

Первые называются сигналами с высоким активным уровнем, а вторые − сигналами с низким активным уровнем. Сигналы MEMRD, MEMWR, IORD, IOWR, PMEN относятся ко второму типу.

В случае ввода/вывода с отображением на память порты ввода/вывода и память не различаются по способу доступа, поэтому можно использовать два стробирующих сигнала RD и WR.

Кроме рассмотренной существуют системы с другим составом управляющих сигналов, например с тремя управляющими сигналами:

– MEM/IO, RD, WR. Сигнал MEM/IO указывает, к какому из пространств (памяти или порту ввода/вывода) осуществляется обращение в данном цикле (выбор пространства), т. е. разделяет циклы обращения к памяти и циклы ввода/вывода. RD − строб чтения, WR − строб записи. Оба эти сигнала являются общими как для памяти, так и для портов ввода/вывода;

– MEM/IO, RD/WR, STRB. Сигнал RD/WR указывает, является ли данный цикл циклом записи или циклом чтения (выбор операции чтения или записи), т. е. разделяет циклы чтения и циклы записи. STRB − строб, используемый как для чтения, так и для записи.

Как отмечалось выше, память в микропроцессорной системе адресуется с точностью до байта. Если шина данных МП является многобайтной (например, в 16-разрядном МП шина данных состоит из двух байт, в 32-разрядном МП – из четырех и т. д.), то удобно обращение к определенному байту в слове осуществлять с помощью специальных управляющих сигналов. Каждый из этих сигналов выбирает определенный байт шины данных. В этом случае младшие разряды шины адреса становятся не нужными, и они не используются. Оставшиеся старшие разряды адресной шины адресуют многобайтное слово. Так в 16-разрядной системе отсутствует самый младший разряд шины адреса A_0 и оставшиеся разряды адреса A_{m-1} – A_1 адресуют 16-разрядное слово, а два управляющих сигнала обеспечивают

обращение к старшему BHE и младшему BLE байтам шины данных (рис. 8). При этом память состоит из двух параллельно работающих блоков, один из которых хранит старшие байты, а второй – младшие байты всех слов. Аналогично в 32-разрядной системе адресная шина не имеет двух младших разрядов A_1,A_0 и оставшиеся разряды адреса A_{m-1} – A_2 – адресуют 32-разрядное слово, а четыре управляющих сигнала BE3-BE0 прямо определяют выбираемые байты внутри этого слова. При этом память состоит из четырех параллельно работающих блоков, каждый из которых хранит соответствующие байты всех слов.

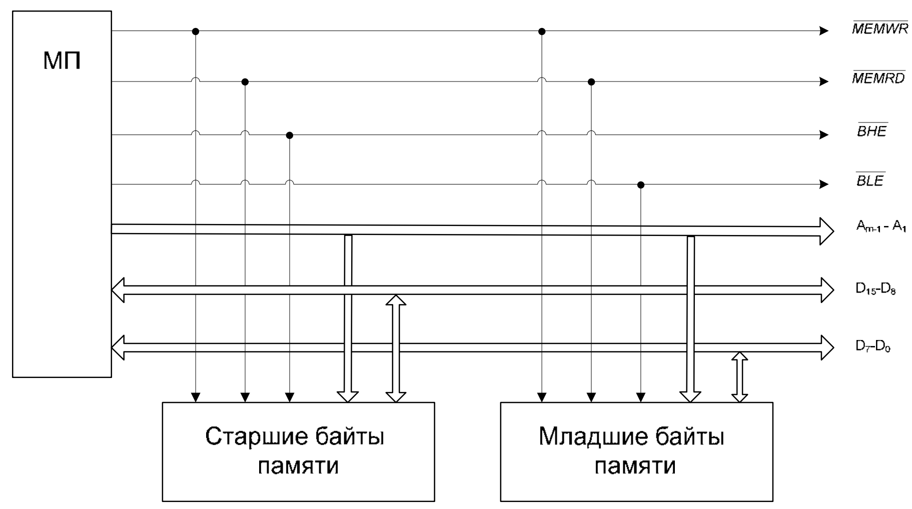


Рис. 8. Специальные управляющие сигналы при работе с памятью

Простые циклы обмена по магистрали. Простые циклы не используют механизм пакетной передачи данных, т. е. за один такой цикл обращения к магистрали между МП и памятью или портом ввода/вывода передается одно слово. МП управляет работой шины синхронно с входной тактовой частотой. Элементарным интервалом времени при реализации протоколов обмена является такт магистрали, равный одному периоду синхросигнала. Каждый цикл шины содержит несколько тактов.

Протокол обмена по магистрали предполагает выполнение определенной последовательности действий:

1) адресация памяти или порта ввода/вывода;

2) коммутация направления передачи (задание операции обмена – чтение или запись);

3) передача данных (выполнение операции обмена);

4) фиксация данных.

В стандартном цикле магистрали для реализации каждого из приведенных выше действий отводится по одному такту, т. е. стандартный цикл магистрали содержит четыре обязательных такта Т1 – Т4.

Рассмотрим процесс считывания/записи в память. Схема соединения памяти с магистралью приведена на рис. 9.

В начале цикла чтения памяти (рис. 10) МП по адресной шине передает адрес, по которому происходит выборка ячейки в памяти, и указанная ячейка памяти подключается к линиям X_{n-1}, … X_0. В первой половине такта Т2 сигнал MEMRD переходит на низкий уровень, а во второй половине такта Т4 он возвращается на высокий уровень. Низким уровнем этого сигнала открывается тристабильный вентиль B_R и линии X_{n-1}, … X_0 соединяются с линиями D_{n-1}, … D_0 шины данных. При этом вентиль B_w закрыт. С момента передачи адреса по адресной шине в память до выдачи содержимого указанной ячейки памяти требуется определенное время, которое называют временем обращения к памяти. Во время считывания микропроцессором данные на шине данных (выходе памяти) должны поддерживаться в неизменном состоянии. Выполнение этого требования обеспечивается за счет наличия в цикле магистрали такта Т3: к концу этого такта содержимое указанной ячейки памяти должно находиться на шине данных. По заднему фронту положительного импульса такта Т4 содержимое шины данных заносится в МП (данные считываются МП и фиксируются во внутреннем регистре).

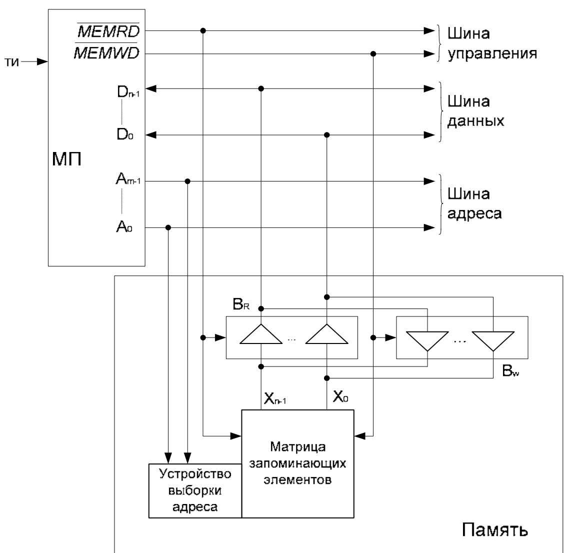


Рис. 9. Схема соединения памяти с магистралью

В начале цикла записи в память (рис. 10) МП передает адрес. С первой половины такта Т2 до окончания такта Т4 МП осуществляет вывод записываемых данных на линии $D_{n-1},\dots,D_0$ шины данных. Низким уровнем сигнала MEMRW открывается тристабильный вентиль B_w, линии $D_{n-1},\dots,D_0$ соединяются с линиями $X_{n-1},\dots,X_0$ и начинается процесс записи в выбранную ячейку памяти. Для записи информации в память также требуется не-

которое время, в течение которого происходит изменение состояния запоминающих элементов, ячейки памяти. Поэтому в течение времени, пока сигнал MEMRW имеет низкий уровень, данные на входе памяти должны поддерживаться в неизменном состоянии. Для этого служит такт T3. Когда уровень сигнала MEMRW на такте T4 становится высоким, содержимое линий $X_{n-1},\ldots,X_0$ фиксируется в ячейке памяти, указанной адресом.

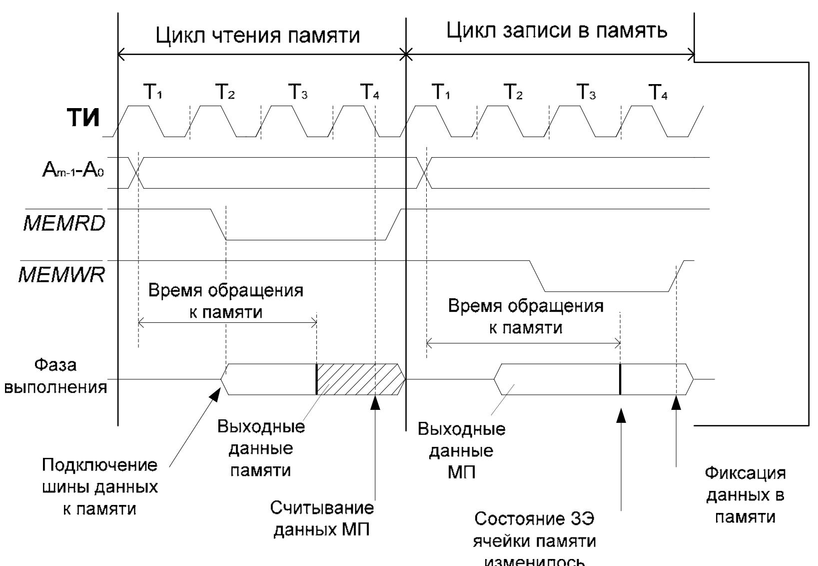


Рис. 10. Диаграмма циклов чтения/записи

Рассмотрим процесс ввода/вывода данных. Схема соединения портов ввода/вывода с магистралью приведена на рис. 11.

Каждый порт ввода представляет собой ряд тристабильных вентилей, при открывании которых данные, поступающие в этот порт передаются на линии $D_{n-1},\ldots,D_0$ шины данных, откуда уже считываются МП.

Порт вывода представляет собой регистр, работа которого заключается в следующем. МП выводит данные на линии $D_{n-1},\ldots,D_0$ шины данных. Эти данные стробирующим сигналом C заносятся в регистр, который обеспечивает их сохранность до записи новых данных.

Для выбора портов ввода/вывода используется дешифратор. Младшими k битами адресной шины можно осуществлять выбор 2^k портов ввода или вывода. Временная диаграмма работы в циклах ввода/вывода данных аналогична временной диаграмме в циклах чтения/записи в память. Различие состоит в том, что вместо сигналов MEMRD и MEMWR МП выдает сигналы IORD и IOWR.

Временные диаграммы работы шины в системе с тремя управляющими сигналами приведены на рис. 12 (на рисунке не показана шина данных). Обратите внимание, что сигнал MEM/IO формируется в начале цикла одновременно с адресом и поддерживается неизменным в течение всего

цикла магистрали. Аналогично формируется сигнал RD/WR, так как направление передачи остается неизменным в течение всего цикла шины. Управляющие сигналы RD и WR стробируют выполнение операций чтение и запись и вырабатываются аналогично сигналам MEMRD (IORD) и MEMWR (IOWR) соответственно. Стробирующий сигнал STRB определяет время выполнения операции чтение или запись и формируется так же, как и сигналы MEMRD, MEMWR, IORD или IOWR.

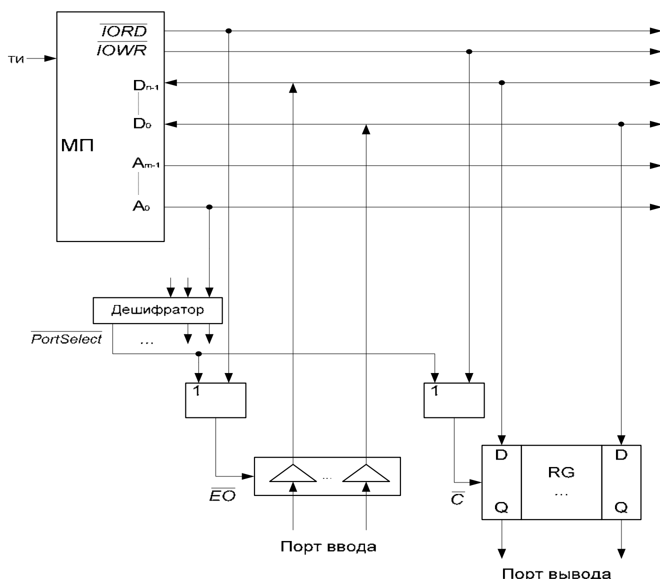


Рис. 11. Схема соединения портов ввода/вывода с магистралью

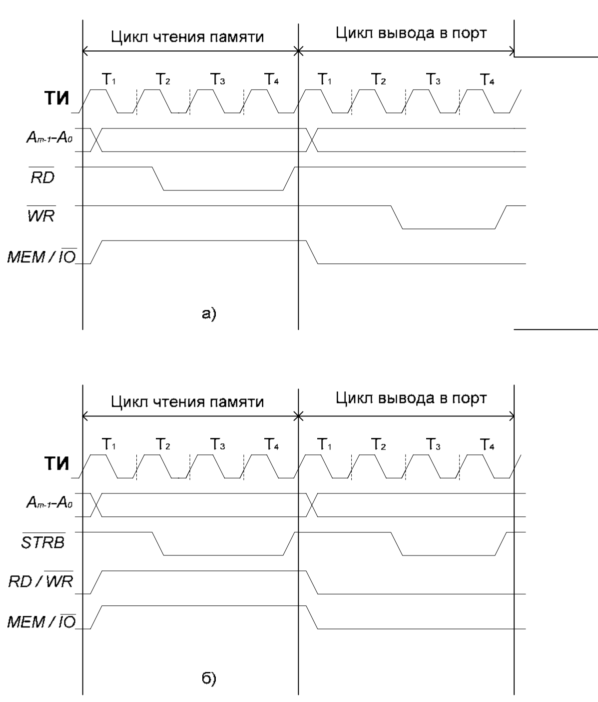


Рис. 12. Временные диаграммы работы шины в системе с тремя управляющими сигналами

В описанной выше стандартной временной диаграмме работы магистрали временные соотношения в циклах чтения/записи полностью задаются МП. В этом случае память и порты ввода/вывода должны постоянно находиться в рабочем (готовом) состоянии, что не всегда возможно. Для медленных устройств система должна позволять произвольно увеличивать длительность циклов шины. Для этого из памяти или из портов ввода/вывода передаются управляющие сигналы, задающие время окончания цикла (подтверждающие окончание цикла) (рис. 13). Как правило, для этой цели используется сигнал READY (ГОТОВНОСТЬ), но могут также использоваться сигналы WAIT (ОЖИДАНИЕ) и TRANSFERACKNOWLEDGE (ПОДТВЕРЖДЕНИЕ ПЕРЕДАЧИ).

Временная диаграмма работы шины с применением сигнала готовности READY приведена на рис. 14. МП по заднему фронту положительного импульса такта T3 (момент времени 1) анализирует состояние сигнала READY. Если данный сигнал имеет высокий уровень, цикл дополняется еще одним тактом (ожидания) TW. По заднему фронту положительного импульса такта TW (момент времени 2) опять анализируется состояние сигнала READY. Если уровень этого сигнала низкий, новые дополнительные такты не вводятся, а следующий такт T4 является последним тактом цикла. Если сигнал READY, анализируемый в такте TW, имеет высокий уровень, цикл дополняется новыми тактами. Таким образом, длительность цикла можно изменять в зависимости от готовности памяти или порта ввода/вывода. Разумеется, в памяти или интерфейсе периферийного устройства должна быть схема, формирующая сигнал READY.

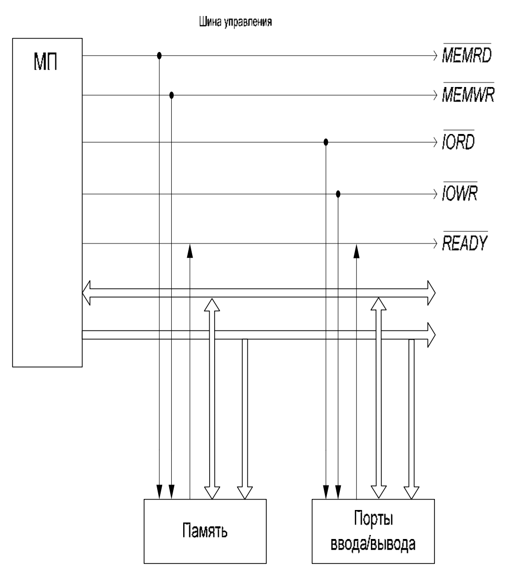


Рис. 13. МП с применением сигнала готовности READY

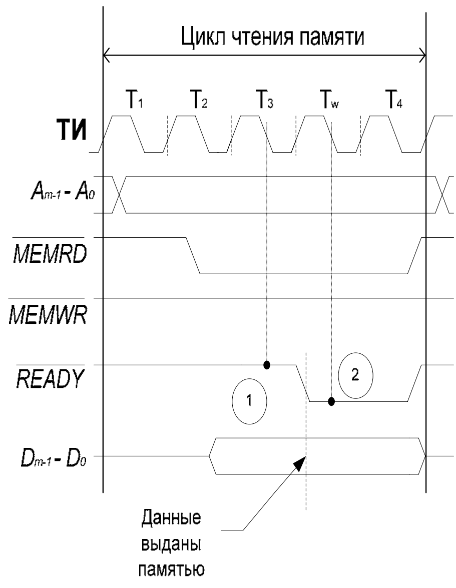


Рис. 14. Временная диаграмма работы шины с применением сигнала готовности READY

Механизм конвейерного формирования адреса при передаче данных. При работе с относительно медленной памятью и портами ввода/вывода длительность циклов магистрали может быть достаточно большой из-за необходимости введения дополнительных тактов ожидания. Сократить число тактов ожидания, необходимых для согласования с относительно медленной памятью и портами, можно за счет применения в процессоре механизма конвейерного формирования адреса. Механизм конвейерного формирования адреса предусматривает возможность начала нового цикла, не дожидаясь завершения физического обмена данными предыдущего цикла. Опережающее начало нового цикла осуществляется за счет более раннего начала формирования адреса микропроцессором и выполнения дешифрации этого адреса устройством памяти или ввода/вывода.

Подключение памяти к шине адреса при обычном обращении приведено на рис. 15. Старшие разряды адреса используются дешифратором адреса для выбора конкретного устройства памяти, а младшие разряды поступают непосредственно в устройство памяти и выбирают в требуемую ячейку. В такой схеме микропроцессор не может выдавать адрес до окончания текущего цикла магистрали. На рисунке 16 показана структура, в которой реализован механизм конвейерного формирования адреса за счет введения дополнительных схем на выходе дешифратора и шины адреса. После дешифрации выданного микропроцессором адреса формируется специальный управляющий сигнал, по которому выходные сигналы дешифратора и младшие разряды адреса запоминаются в соответствующих регистрах-фиксаторах. Далее устройства памяти работают в соответствии с выходными сигналами этих регистров. Следовательно, микропроцессор может выдавать адрес следующего цикла магистрали еще до окончания передачи данных текущего цикла. Временная диаграмма работы такой схемы показана на рис. 17.

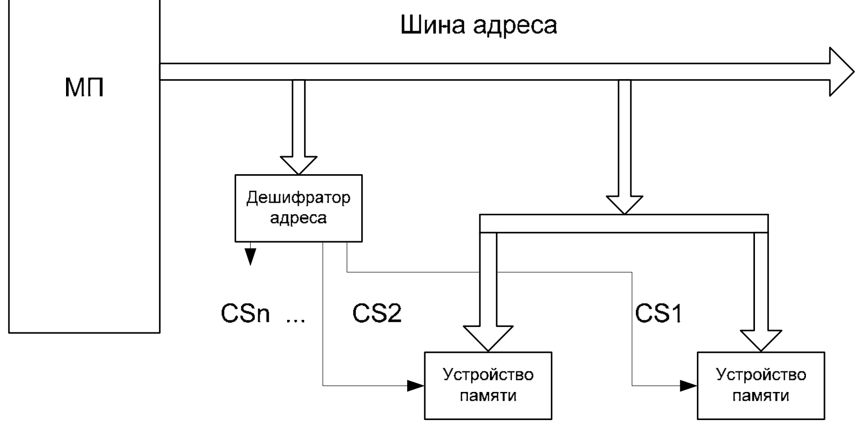


Рис. 15. Подключение памяти к шине адреса при обычном обращении

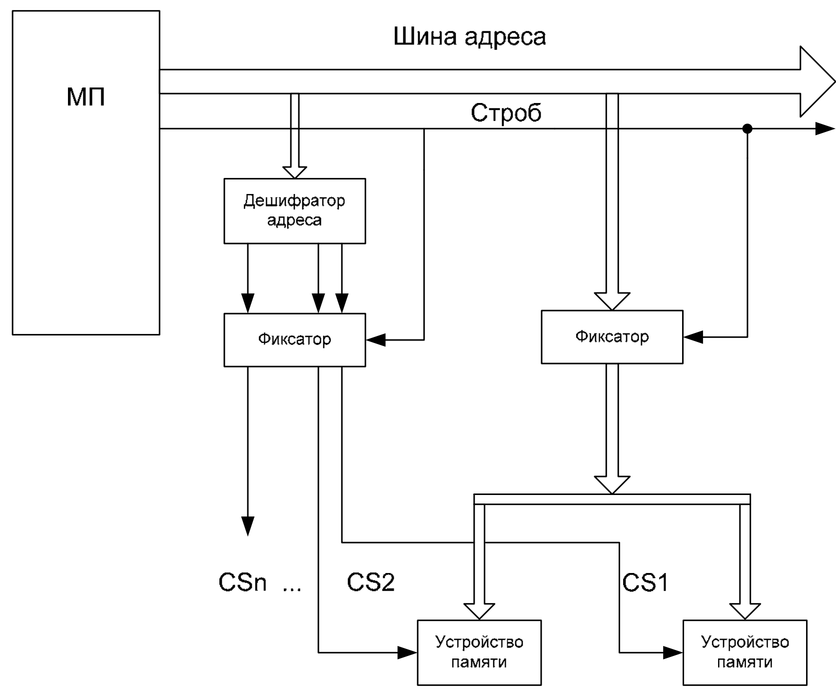


Рис. 16. Конвейерное формирование адреса

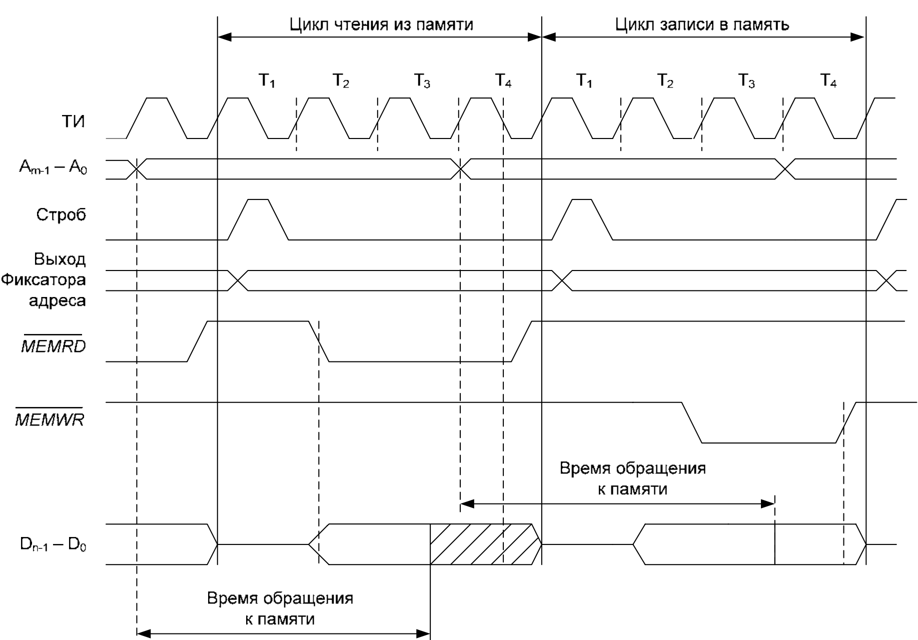


Рис. 17. Временная диаграмма работы схемы конвейерного формирования адреса

Последовательности циклов и пакетная передача. Для обслуживания некоторых внутренних запросов при работе с памятью микропроцессорной системы процессору может понадобиться последовательность циклов обмена, во время которых передаются данные, расположенные в смежной области адресного пространства. Такая ситуация может иметь место при выборке операндов, имеющих разрядность большую, чем раз-

рядность шины данных (например, 32-разрядный процессор может обращаться к 64- или 128-разрядным операндам), или при заполнении строки кэш-памяти (например, строка КЭШа процессора Pentium III имеет длину 32 байта, следовательно, для ее пересылки требуется четыре 64-разрядных цикла магистрали). Во всех таких случаях, когда требуется больше одного цикла для передачи данных, микропроцессор может выполнять пакетные циклы. Во время пакетного цикла между МП и памятью передается более одного слова, причем эти слова занимают смежные адреса и направление передачи для всех слов одинаково (т. е. все слова читаются из памяти или записываются в память). Такой протокол обмена по магистрали называется режимом пакетной передачи (Burst Mode).

Выполнение стандартного цикла магистрали можно разбить на две фазы:

– фаза адресации-идентификации, которая включает адресацию памяти и коммутацию направления передачи;

– фаза выполнения операции, которая включает саму передачу данных и их фиксацию.

В стандартном цикле фаза 1 занимает такты T1 и T2, а фаза 2 – такты T3 и T4. В ходе пакетных циклов очередное слово передается в каждой фазе, а не через фазу, как в обычных циклах обмена. При этом на передачу первого слова затрачивается две фазы, а далее данные передаются в каждой фазе. Такой протокол обмена возможен, если адрес и сигналы идентификации типа цикла выдавать только в первой фазе пакета, а в каждой из последующих фаз передавать данные, адрес для которых уже не передается по шине, а вычисляется из первого по правилам, известным и микропроцессору и памяти.

Пакетный цикл (рис. 18) начинается МП так же, как и обычный, в первой фазе на шине адреса устанавливается адрес первого слова пакета, а на шине управления – сигналы идентификации типа цикла (например, MEM / IO и RD / WR). В следующей фазе передается первое слово данных, и, если оно не единственное, специальный управляющий сигнал BM , который указывает, что данный цикл пакетный. Далее МП продолжает цикл как пакетный, не вводя фазы адресации-идентификации, а сразу перейдет к передаче следующего слова данных. О завершении пакетного цикла микропроцессор сообщает памяти снятием сигнала BM.

На рисунке 18 приведена временная диаграмма пакетного цикла чтения из памяти. Пакетный цикл включает передачу четырех слов. Из рис. 18 видно, что для передачи четырех слов с помощью пакетного цикла требуется 10 тактов, в то время как передача четырех слов с помощью обычных циклов занимает 4 цикла х 4 такта = 16 тактов.

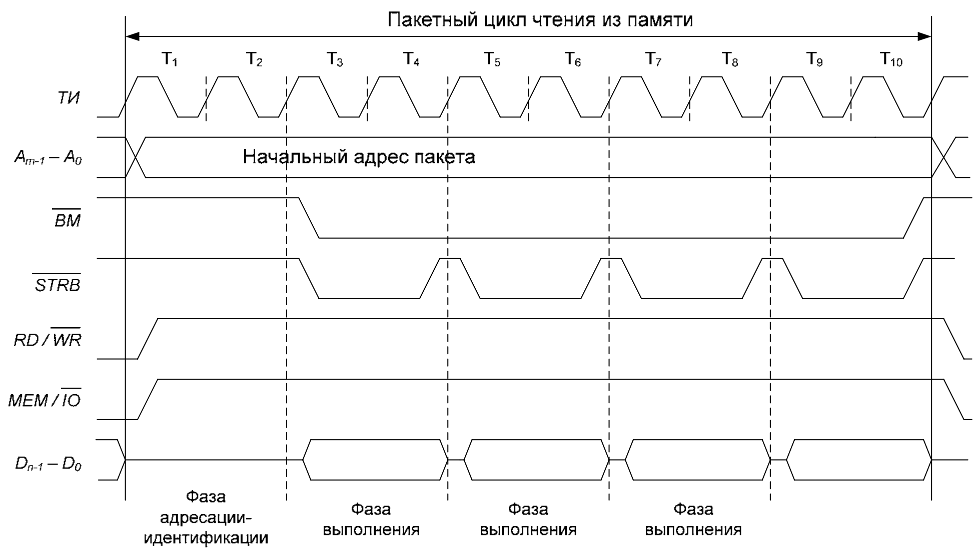


Рис. 18. Временная диаграмма пакетного цикла

Совмещение адресной шины и шины данных. В некоторых МП с целью сокращения ширины физической магистрали используют совмещение адресной шины с шиной данных. В течение первого такта цикла магистрали шина данных не используется, поэтому этот интервал можно использовать для передачи по шине данных адресных сигналов (адреса). Этап передачи адресной информации по совмещенной шине адреса/данных AD (Address/Data Bus) отделяется по времени от этапа передачи данных и стробируется специальным сигналом ALE (Address Latch Enable), который включается в состав шины управления. Данную магистраль называют двухшинной с совмещенными шинами передачи адреса и данных. Если разрядность данных меньше разрядности адреса, то по совмещенной шине передаются только младшие разряды адреса, а старшие разряды при этом передаются по адресной шине.

Входящий в состав шины управления сигнал ALE используется для разделения функций, выполняемых совмещенной шиной AD. По этому сигналу присутствующая на шине AD адресная информация должна быть принята (зафиксирована) во внешний (по отношению к МП) адресный регистр-фиксатор. Для этой цели обычно служит срез сигнала ALE (переход из высокого уровня в низкий). Обычно каждый модуль микропроцессорной системы с двухшинной магистралью (модуль памяти или интерфейс периферийного устройства) содержит локальный адресный регистр для запоминания адресной информации. Для фиксации адресной информации может быть использован и один общий регистр, в результате МП с двухшинной магистралью преобразуется в МП с тремя раздельными шинами (рис. 19). Когда уровень управляющего сигнала, приходящего на вход C регистра-фиксатора, становится высоким, входная информация без изме-

нения передается на выход. При переходе управляющего сигнала на входе С в низкий уровень информация фиксируется в регистре.

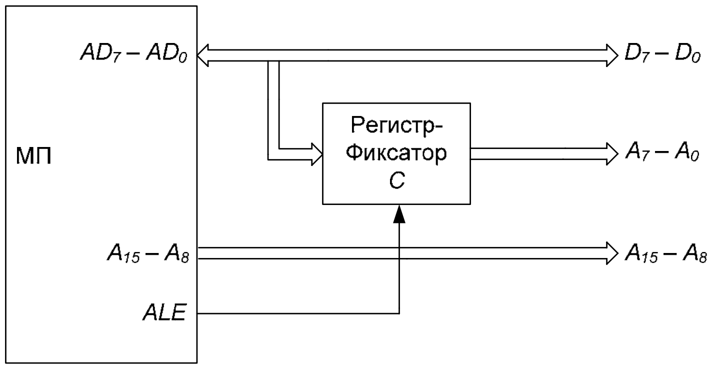


Рис. 19. МП с совмещением адресной шины с шиной данных

Временная диаграмма работы микросистемы с совмещением адресной шины с шиной данных приведена на рис. 20. В течение первого такта T_1 по общей шине $AD_7 - AD_0$ передаются адресные разряды $A_7 - A_0$. Эти разряды по сигналу ALE фиксируются в регистре-фиксаторе, который находится вне МП.

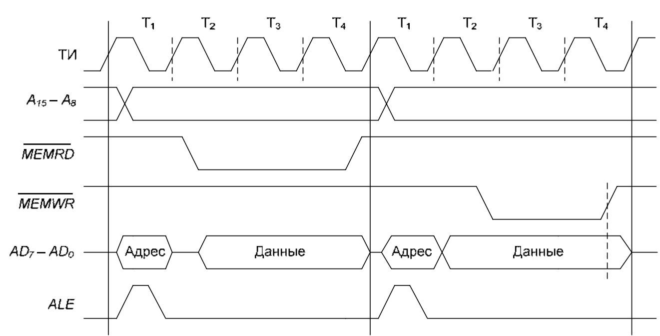


Рис. 20. Временная диаграмма работы микросистемы
с совмещением адресной шины с шиной данных

3. Архитектура системы ввода/вывода. Способы организации передачи данных. Система непосредственного ввода/вывода. Система канального ввода/вывода. Программно-управляемый ввод/вывод. Прямой ввод/вывод. Условный ввод/вывод. Ввод/вывод с программным квитированием. Ввод/вывод по прерываниям

Микропроцессорная система состоит из трех подсистем: микропроцессора, подсистемы памяти и подсистемы ввода/вывода. Подсистема ввода/вывода отвечает за связь с устройствами ввода/вывода.

Обычно употребляется термин периферийные устройства, которые подразделяются на устройства ввода/вывода (клавиатура, мышь, принтер, монитор и др.) и внешние запоминающие устройства. Внешние запоминающие устройства занимают в микропроцессорной системе особое поло-

жение. С точки зрения выполняемой функции они относятся к подсистеме памяти (внешняя память, предназначенная для хранения информации, совместно с основной памятью может использоваться для организации виртуальной памяти), а связь с ними осуществляется так же, как и с устройствами ввода/вывода, т. е. с помощью подсистемы ввода/вывода. Поэтому при рассмотрении подсистемы ввода/вывода все периферийные устройства будем называть устройствами ввода/вывода.

Связь устройств микропроцессорной системы друг с другом осуществляется с помощью специальных совокупностей средств и правил, которые называются интерфейсами. *Интерфейс* – совокупность линий и шин, сигналов, электронных схем и алгоритмов, предназначенная для осуществления обмена информацией между устройствами компьютерной системы.

При разработке систем ввода/вывода должны быть решены следующие проблемы:

– возможность реализации системы с переменным составом оборудования, в первую очередь с различным набором устройств ввода/вывода, с тем, чтобы пользователь мог выбирать состав оборудования (конфигурацию) системы в соответствии с ее назначением, легко дополнять систему новыми устройствами;

– для эффективного и высокопроизводительного использования оборудования системы возможность параллельной во времени работы процессора над программой и выполнения периферийными устройствами процедур ввода/вывода;

– упрощение для пользователя и стандартизация программирования операций ввода/вывода, обеспечение независимости программирования ввода/вывода от особенностей того или иного устройства ввода/вывода;

– автоматическое распознавание и реакция системы на многообразие ситуаций, возникающих в устройствах ввода/вывода (например, готовность устройства, отсутствие носителя, различные нарушения нормальной работы устройства и т. п.).

Основными путями решения указанных проблем являются:

– модульность;

– унифицированные (не зависящие от типа устройства ввода/вывода) форматы данных, которыми устройства ввода/вывода обмениваются с системой;

– унифицированные интерфейсы;

– унифицированные (не зависящие от типа устройства ввода/вывода) формат и набор команд процессора для операций ввода/вывода.

Для обеспечения параллельной во времени работы устройств ввода/вывода с выполнением программы обработки данных процессором схемы управления вводом/выводом отделяются от процессора и им придается достаточная степень автономности.

Различают два основных вида архитектуры систем ввода/вывода:

– система непосредственного ввода/вывода;
– система канального ввода/вывода.

В системе непосредственного ввода/вывода (рис. 21) обмен данными с устройствами ввода/вывода выполняется процессором.

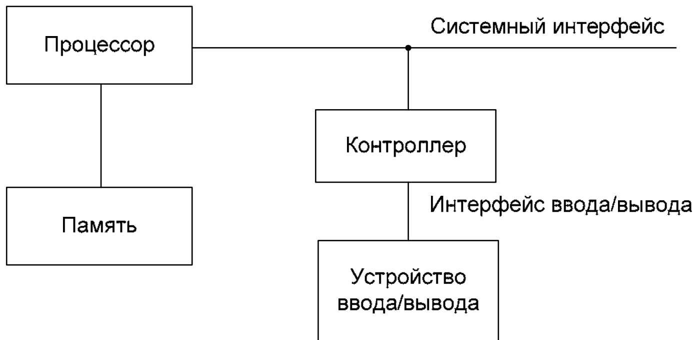


Рис. 21. Система непосредственного ввода/вывода

В системе канального ввода/вывода (рис. 22) обмен данными, подлежащими вводу в процессор или выводу из него, происходит между процессором и памятью, а память связана с устройствами ввода/вывода через канал или процессор ввода/вывода.

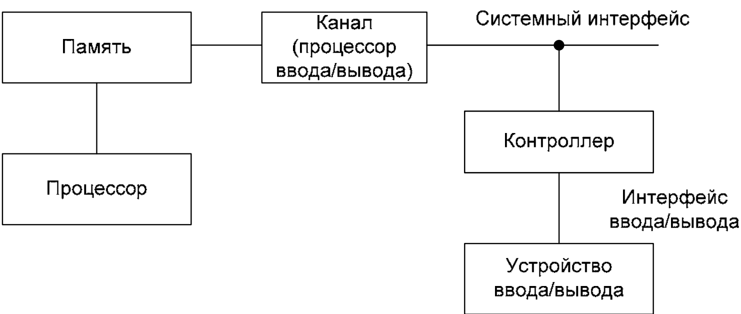


Рис. 22. Система канального ввода/вывода

При любой форме взаимодействия с микропроцессорной системой аппаратура ввода/вывода обычно состоит из собственно устройства ввода/вывода и устройства управления этим устройством ввода/вывода – контроллера устройства ввода/вывода.

Интерфейс между процессором или каналом ввода/вывода и контроллером устройства ввода/вывода называется *системным интерфейсом*, а интерфейс между контроллером и устройством ввода/вывода – *интерфейсом ввод/вывода.* Системный интерфейс, как правило, является общим для всех видов устройств ввода/вывода, а интерфейсы ввода/вывода специализированы для конкретных видов устройств ввода/вывода.

В общем случае ввод и вывод данных осуществляются путем обмена даными между основной памятью и устройствами ввода/вывода. Данные, находящиеся в основной памяти, подаются на устройства вы-

вода, а от устройства ввода данные поступают для занесения в основную память. Для управления этим обменом необходимо обеспечить выполнение следующих функций:

1) управление адресацией всех данных, подлежащих вводу и выводу, обновление адресов при передаче каждого слова;

2) синхронизация обмена данными между устройствами ввода/вывода и основной памятью или процессором;

3) управление работой устройств ввода/вывода. Необходимо осуществлять не только передачу данных, но и реализацию управляющих действий, например, перемещение магнитной головки на нужную дорожку;

4) преобразование информации. Обычно основная память связана с внутренней системной шиной, а связь с устройствами ввода/вывода может осуществляться по-разному в зависимости от специфики этих устройств. Поэтому при передаче информации по шинам между основной памятью или процессором и устройствами ввода/вывода должно осуществляться ее преобразование.

Таким образом, при вводе/выводе принципиально необходимы действия, связанные с синхронизацией обмена данными ввода/вывода и их адресацией. В системах непосредственного ввода/вывода такие действия (перечисленные выше функции) выполняет процессор, а в системах канального ввода/вывода их выполнение возложено на различные канальные устройства (контроллеры ПДП, процессоры ввода/вывода).

Функции управления вводом/выводом могут быть:

- общими, не зависящими от типа устройств ввода/вывода;
- специфичными для каждого типа устройств ввода/вывода.

Выполнение общих функций возлагают на общие для групп устройств ввода/вывода унифицированные устройства – контроллеры ПДП, процессоры (каналы) ввода/вывода, а специфических – на специализированные для каждого типа устройств ввода/вывода устройства управления – контроллеры устройств ввода/вывода.

С точки зрения программиста, работающего на уровне машинных команд, подсистему ввода/вывода можно представить в виде пространства ввода/вывода IOS и набора команд ввода/вывода, обеспечивающих к нему доступ. Организация пространства ввода/вывода подобна организации пространства памяти: IOS организовано в виде набора n-разрядных ячеек – портов, каждый из которых может быть адресован независимо от других.

Между микропроцессором и периферийными устройствами происходит обмен информацией двух типов:

- служебной;
- собственно данными.

Служебная информация от МП инициирует действия, связанные с обменом данными, и передается с помощью управляющих слов CW (Control Word). Служебные сообщения от периферийных устройств информируют МП об их текущем состоянии и называются словами состояния SW (Status Word). В отличие от них данные передаются с помощью слов данных DW (Data Word).

Объем служебной информации, которой обмениваются периферийные устройства и микропроцессор, а также ее интерпретация зависят от типа периферийного устройства. Для наиболее простых устройств, таких как прямо управляемые клавишные матрицы или светодиодные индикаторы, служебная информация не нужна. В других случаях, например при взаимодействии с НГМД, управляющая информация и данные о состоянии устройства могут иметь большой объем. При этом каждое периферийное устройство (ПУ) воспринимает определенный, присущий только ему набор команд управления. Организовать в этом случае передачу каждой команды ПУ по отдельной линии магистрали (шины управления) не представляется возможным по двум причинам. Во-первых, при разработке микропроцессора достаточно трудно предусмотреть все возможные применения микропроцессорной системы на его основе, а, следовательно, и используемые в ней ПУ. Во-вторых, для каждого дополнительного управляющего сигнала потребуется отдельный вывод в БИС микропроцессора, т. е. возникают чисто конструктивные ограничения на количество используемых в шине управления магистрали управляющих сигналов, связанные с числом выводов в БИС микропроцессора. Решение указанной проблемы осуществляется путем мультиплексирования шины данных: в одни моменты времени она используется для передачи данных, в другие моменты – для передачи служебной информации. При этом для связи с периферийным устройством отводится ряд портов ввода/вывода, через которые и проходит вся информация: управляющая, слова состояния и непосредственно данные. С точки зрения программиста множество портов ввода/вывода, связанных с данным периферийным устройством, образует пространство доступа к этому периферийному устройству.

В общем случае размер пространства доступа не зависит от объема информации, которой обмениваются периферийное устройство и микропроцессор, так как используется последовательная передача массива информации через один и тот же порт. При этом существуют правила обмена информацией между конкретным периферийным устройством и микропроцессором, которые называют протоколом обмена. Взаимодействие с периферийным устройством организуется с помощью драйвера – специальной программы, составленной на основе этих правил.

Подключение любого периферийного устройства к магистрали микропроцессорной системы осуществляется через контроллер ПУ.

Рассмотрим организацию типичного контроллера ПУ (рис. 23). Основу контроллера составляют регистры, которые служат для хранения передаваемой информации. Взаимодействие микропроцессора с этими регистрами осуществляется через порты ввода/вывода из пространства доступа к ПУ. Регистры и порты тесно связаны, иногда их трудно отделить друг от друга. Зачастую их отождествляют друг с другом. В этом смысле каждый регистр имеет свой адрес. Под адресом регистра понимается адрес порта, через который осуществляется доступ к этому регистру.

В контроллере ПУ используются регистры четырех типов в зависимости от типа информации, для хранения которой они предназначены:

– регистр входных данных или входной регистр (доступен микропроцессору только по чтению);

– регистр выходных данных или выходной регистр (доступен микропроцессору только по записи);

– регистр состояния (доступен микропроцессору только по чтению). Содержит информацию о текущем состоянии ПУ (включено/выключено, готово/не готово к обмену данными, ошибка и т. п.);

– регистр управления (доступен микропроцессору только по записи). Служит для приема из МП команд для ПУ.

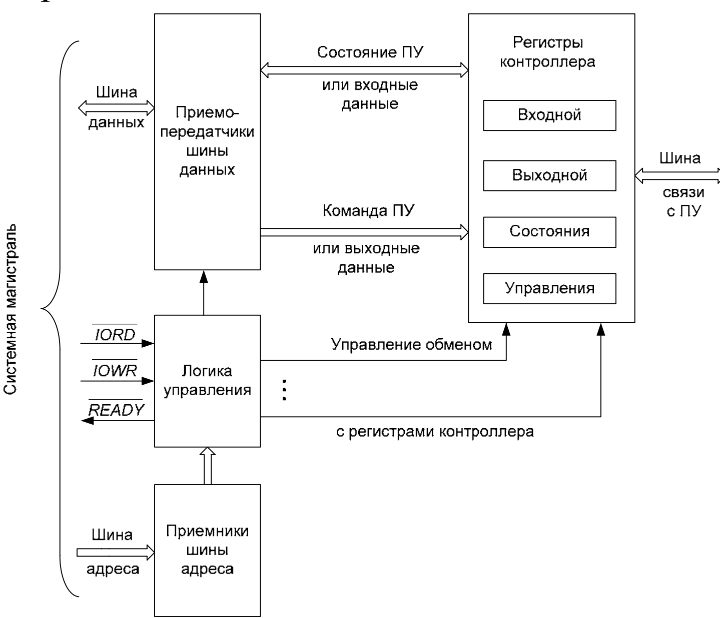


Рис. 23. Организация контроллера ПУ

В контроллерах сложных ПУ регистров каждого типа может быть несколько. В контроллерах, предназначенных для подключения простых ПУ, удается совместить регистры состояния и управления, что позволяет сократить количество используемых в контроллере портов ввода/вывода, а следовательно, и адресов, выделенных для данного ПУ.

Логика управления контроллера выполняет селекцию адресов регистров (портов ввода/вывода) контроллера, прием, обработку и формирова-

ние управляющих сигналов системной магистрали, а также выработку внутренних управляющих сигналов, обеспечивая тем самым обмен информацией между регистрами контроллера и шиной данных магистрали микропроцессорной системы.

Приемо-передатчики шины данных и шины адреса служат для физического подключения схем контроллера к соответствующим шинам системной магистрали.

На практике часто используют программируемые контроллеры, режимы работы которых устанавливаются специальными командами МП. Программируемый контроллер необходимо настраивать на конкретный режим обмена данными, присущий ПУ: синхронный или асинхронный, с использованием сигналов прерывания или без их использования, на заданную скорость обмена и т. д. Настройка таких контроллеров на требуемый режим обмена производится программным путем с помощью специальных команд (управляющих слов), передаваемых из МП в контроллер ПУ перед началом обмена. Управляющее слово записывается в специальный регистр и инициирует заданный режим обмена с ПУ. В качестве примера можно привести микросхему фирмы Intel 8255 – программируемый контроллер параллельного интерфейса.

В микропроцессорных системах используются два основных способа организации передачи данных между системой и устройствами ввода/вывода (рис. 24):

– программно-управляемый обмен;
– прямой доступ к памяти (ПДП).

Программно-управляемый обмен – обмен, управляемый программой, т. е. когда процедуры обмена информацией с периферийным устройством инициируются и выполняются непосредственно программой, реализуемой процессором через его регистры. Программно-управляемый обмен осуществляется при непосредственном участии и под управлением процессора.

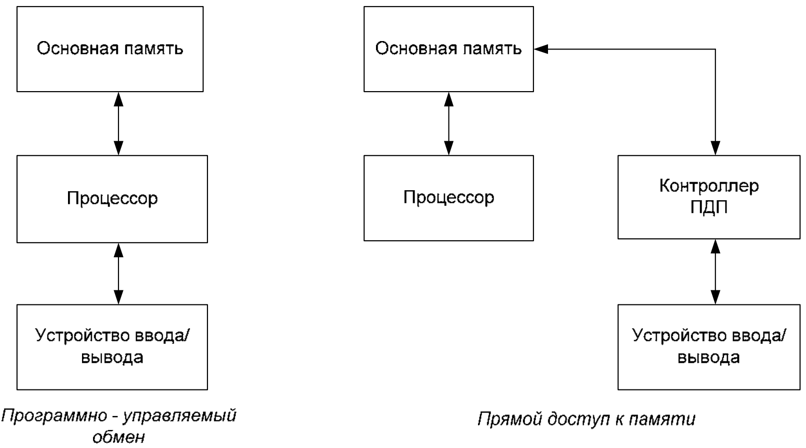


Рис. 24. Два основных способа организации передачи данных между системой и устройствами ввода/вывода

С точки зрения аппаратных затрат программно-управляемый обмен является наиболее эффективным типом обмена, поэтому он находит самое широкое применение в различных микропроцессорных системах.

Программно-управляемый обмен может осуществляться одним из трех способов, называемых:

1) прямой (безусловный, синхронный) обмен;
2) условный (асинхронный) обмен (обмен по условию);
3) обмен с прерыванием программы.

В зависимости от используемого способа обмена программно-управляемый ввод/вывод называется соответственно

1) прямым вводом/выводом;
2) условным вводом/выводом;
3) вводом/выводом по прерыванию.

Прямой ввод/вывод

Используется только для процессов, строго фиксированных во времени и полностью определенных. Процедуры ввода или вывода выполняются независимо от состояния периферийного устройства. Процедуры прямого ввода/вывода в чистом виде возможны только при работе с всегда готовыми к обмену простейшими устройствами, для которых точно известно время выполнения операции (например, индикатор). Они являются составными элементами более сложных процедур программно-управляемого обмена – условного ввода/вывода и ввода/вывода по прерыванию.

Прямой ввод/вывод реализуется с помощью портов пространства IOS. Порты ввода/вывода являются простейшими схемными элементами, на основе которых реализуется связь с периферийными устройствами. Они образуют первый, наиболее близкий к МП уровень аппаратных средств подсистемы ввода/вывода. В простейшем случае этот уровень является единственным.

Быстродействия шинных формирователей и регистровых схем достаточно, чтобы поддерживать обмен с максимальной для МП скоростью, а периферийные устройства считаются всегда готовыми к обмену (поэтому такой обмен называется синхронным). Если быстродействия этих элементов недостаточно для поддержания обмена с максимальной для МП скоростью, необходимо использовать расширение длительности цикла магистрали с помощью сигнала готовности READY.

Условный ввод/вывод

Прямой ввод/вывод является наиболее простым видом обмена, требующим минимальных затрат аппаратных и программных средств. Однако, как правило, скорость работы периферийных устройств во много раз ниже скорости работы МП, что приводит к проблеме синхронизации обмена. Поэтому прежде чем приступить к чтению новых данных из порта ввода, необходимо удостовериться, что ПУ готово предоставить или уже пре-

доставило эти данные. В противном случае операция сведется к вводу недействительных или старых данных. Аналогичная ситуация складывается и при выводе данных, когда требуется проверка готовности ПУ к приему новых данных. В противном случае неразрешенный со стороны ПУ вывод может привести к потере данных.

Типичное решение проблемы синхронизации обмена состоит в использовании условного ввода/вывода. При условном вводе/выводе операции обмена сопровождаются специальным сигналом готовности RDY, вырабатываемом периферийным устройством и входящим в состав его слова состояния SW. Этот сигнал служит для информирования МП о готовности периферийного устройства принять или передать новые данные. После завершения операции ввода/вывода сигнал готовности (соответствующий разряд слова состояния SW) должен быть снят и выставлен заново только при новой готовности к обмену. С этой целью периферийное устройство следует проинформировать об окончании операции, для чего используется включенный в управляющее слово CW сигнал подтверждения ACK. Протокол обмена служебной информацией такого типа называется квитированием. Он обеспечивает надежную асинхронную передачу данных со скоростями, определяемыми периферийным устройством.

Состояние сигнала готовности RDY может быть определено микропроцессором путем чтения регистра состояния, в котором находится слово состояния ПУ, через соответствующий порт ввода. Сигнал подтверждения ACK может формироваться микропроцессором путем записи управляющего слова в регистр управления через соответствующий ему порт вывода. Такой вариант формирования сигналов квитирования называется *программным квитированием.*

В сравнении с прямым условный ввод/вывод с программным квитированием связан с увеличением аппаратных затрат, а также с потерями времени МП на ожидание готовности периферийного устройства. Однако это наиболее распространенный вид обмена с периферийными устройствами. Он используется в системах, где эффективность не связана с ожиданиями.

Признаком окончания операции может служить само обращение к порту данных. Это упрощает как схему порта, так и процедуру обмена, освобождая пользователя от работы с управляющим словом. При этом на входе порта ввода предусматривается регистр-защелка, фиксирующий входные данные по стробу STB, генерируемому периферийным устройством. Запись во входной буфер устанавливает флажок готовности IBF (Input Buffer Full), инициируя операцию ввода. Флажок сбрасывается автоматически при чтении МП содержимого входного буфера. При выводе роль флажка готовности выполняет флажок OBF (Output Buffer Full). Сигналом

окончания операции вывода и установки флажка OBF служит сигнал подтверждения ACK, генерируемый периферийным устройством.

Ввод/вывод по прерыванию

Эффективность работы микропроцессорной системы может быть существенно повышена, если отказаться от малопроизводительного ожидания готовности ПУ на программном уровне и передать эту функцию специальным аппаратным средствам. В это время процессор может выполнять некоторую полезную работу, связанную с обработкой данных или обслуживанием других ПУ.

При готовности приступить к очередной операции ввода/вывода ПУ посылает в микропроцессор запрос на прерывание, по получении которого МП временно приостанавливает (прерывает) выполнение текущей программы и передает управление специальной подпрограмме, организующей нужный вид обмена данными. После обслуживания ПУ микропроцессор возвращается к прерванной программе, продолжая ее с момента приостановки. Обслуживание прерываний осуществляется в незаметном для основной программы режиме, поэтому их наличие прямо не влияет на работу последней, за исключением времени ее исполнения.

Обслуживание ввода/вывода по прерыванию является альтернативой двум другим видам программно-управляемого обмена. Если при чисто программном управлении, как начало процедуры, так и непосредственное ее исполнение находятся под управлением программы, то обслуживание по прерыванию инициируется аппаратными средствами.

Таким образом, обмен с прерываниями отличается от условного (асинхронного) программно-управляемого обмена тем, что переход к выполнению команд, физически реализующих обмен данными (командам ввода/вывода), осуществляется с помощью специальных аппаратных средств. Команды обмена данными в этом случае выделяют в отдельный программный модуль – подпрограмму обработки прерывания. Задачей аппаратных средств обработки прерываний в МП является приостановка выполнения одной программы (основной) и передача управления подпрограмме обработки прерывания. Действия, выполняемые при этом МП, как правило, те же, что и при обращении к подпрограмме. Только при обращении к подпрограмме они инициируются специальной командой, а при обработке прерывания – сигналом от ПУ, который называют запросом прерывания.

Эта важная особенность обмена с прерыванием программы позволяет организовать обмен данными с ПУ в произвольные моменты времени, не зависящие от программы, выполняемой в МП. Таким образом, появляется возможность обмена данными с ПУ в реальном масштабе времени, определяемом внешней по отношению к микропроцессорной системе сре-

дой. Кроме того, обмен с прерыванием программы существенным образом экономит время процессора, затрачиваемое на обмен, за счет того, что исчезает необходимость в организации программных циклов ожидания готовности ПУ (особенно при обмене с медленнодействующими ПУ).

Прямой доступ к памяти – обмен, управляемый внешним устройством. ПДП осуществляется непосредственно с памятью без вмешательства программы, минуя МП, в общем случае по специальному информационному каналу под управлением специального устройства – контроллера прямого доступа к памяти.

4. Понятие прерывания процессора. Организация подсистемы прерываний в МПС. Контекстное переключение. Организация радиальной системы прерываний. Метод поллинга. Организация векторной системы прерываний. Вектор прерывания

Совокупность специальных аппаратных средств, осуществляющих прием запроса на прерывание и переход к подпрограмме обработки этого запроса, команд и программ обслуживания запросов прерывания называется системой прерываний.

Кроме обслуживания ПУ (выполнения обмена) в микропроцессорной системе существуют и другие события, которые могут вызвать прерывание процессора. Типы прерываний зависят от конкретной микропроцессорной системы. Прерывания распадаются на два основных класса:

1) внешние прерывания. Вызываются асинхронными событиями, которые происходят вне прерываемой программы. Например, прерывания от таймера или подсистемы ввода/вывода;

2) внутренние прерывания. Вызываются событиями, которые связаны с работой процессора и являются синхронными с его операциями. Например:

– арифметическое переполнение, попытка деления на 0, переполнение или исчезновение порядка при выполнении операций с плавающей запятой;

– внутренние прерывания происходят при обращении к защищенным или несуществующим ячейкам памяти, а также к отсутствующему сегменту или странице;

– прерывания вызывает попытка использовать незадействованный код операции, а также попытка использовать привилегированные команды в пользовательском режиме;

– внутренние прерывания могут вызываться также сбоями системы, например, ошибкой четности;

– при выполнении специальных команд.

Прерывание программы по требованию ПУ не должно оказывать на прерванную программу никакого влияния кроме увеличения времени ее выполнения за счет приостановки на время выполнения подпрограммы об-

работки прерывания. Для этого после приема запроса на прерывание состояние МП необходимо сохранить. Для того чтобы прерванная программа могла быть продолжена после обслуживания очередного запроса на прерывание с того места, на котором она была приостановлена, состояние процессора должно быть восстановлено. Сохранность состояний сигналов управления в аппаратуре обеспечивается тем, что фиксация прерывания, а следовательно, переход к его обработке и возврат к прерванной программе осуществляются в строго определенные моменты времени, когда состояния этих сигналов однозначны и известны. Программа состоит из ряда команд, выполнение которых приводит к изменению состояния процессора. Это изменение дискретно: состояние наблюдается лишь в отдельные моменты времени – точки наблюдения, которые соответствуют начальным и конечным моментам выполнения команд. Таким образом, фиксация прерывания осуществляется в конце очередного командного цикла перед фазой выборки следующей команды. В этом случае состояние процессора определяется содержимым его программно-доступных регистров и вектора состояния и называется контекстом процессора. Под вектором состояния понимается содержимое внутренних регистров процессора, в которых может храниться информация трех типов:

1) информация о состоянии процессора, которая включает

– состояние выполнения. МП может находиться либо в состоянии выполнения команд (в активном состоянии), либо в состоянии ожидания, когда выполнение прекращается;

– режим – пользовательский или системный;

– маски прерывания;

2) информация о доступном контексте в памяти и соответствующих правах доступа, например, таблицы сегментов, указатели защиты памяти и т. п.;

3) информация о ходе выполнения текущей программы, которая включает признаки выполнения операции (код условия) и адрес следующей выполняемой команды.

Контекст процессора может быть сохранен в памяти микропроцессорной системы, а затем восстановлен непосредственно перед возвратом в прерванную программу. Этот процесс называется *контекстным переключением* и выполняется он как аппаратными, так и программными средствами. Всякий раз, когда процессор воспринимает запрос на прерывание, он активизирует процедуру обслуживания, передавая ее стартовый адрес в программный счетчик. Чтобы не потерялось старое содержимое программного счетчика, которое является адресом возврата в прерванную программу, оно должно быть автоматически сохранено. Лучше всего для этой цели использовать системный стек, тогда возврат к прерванной программе будет заключаться в передаче управления по адресу, находящемуся

на вершине стека. Обычно аппаратными средствами обработки прерывания автоматически сохраняется не только содержимое программного счетчика, но и весь вектор состояния процессора (как правило, содержимое программного счетчика и регистра состояния процессора). В этом же стеке можно сохранять и остальную часть текущего контекста процессора – программно-доступные регистры. Как правило, это выполняется программно с помощью обычных команд типа PUSH (запись в стек) непосредственно в подпрограмме обработки прерывания. При этом восстановление контекста осуществляется с помощью команд типа POP (чтение из стека). Контекстное переключение, выполняемое командами PUSH и POP, может занимать значительное время. Для его сокращения вводятся специальные команды PUSHA и POPA, которые сохраняют в стеке и восстанавливают сразу весь набор регистров МП. Еще один механизм контекстного переключения связан с переключением регистровых наборов в МП. Кроме того, контекстное переключение может выполняться и полностью аппаратным путем.

Таким образом, хотя существует несколько различных способов обработки прерывания, следующая последовательность действий присуща большинству микропроцессорных систем:

1) фиксируются характеристики произошедшего прерывания (тип прерывания);

2) сохраняется состояние прерванной программы – контекст процессора;

3) анализируется тип прерывания и передается управление соответствующей подпрограмме обработки этого прерывания;

4) обрабатывается прерывание – выполняется соответствующая подпрограмма;

5) восстанавливается контекст процессора, что приводит к возобновлению выполнения прерванной программы.

Как уже отмечалось ранее, процесс обработки запросов на прерывание во многом подобен процессам вызова и возврата из подпрограммы. Однако в первом случае вызов осуществляется специальной командой, которая сформирована и подставлена в общую командную последовательность с помощью аппаратуры системы прерываний, а во втором случае команда вызова подпрограммы действительно присутствует в последовательности команд. По этой причине запросы на прерывание можно рассматривать как аппаратные вызовы подпрограмм.

Во всех системах прерываний предусмотрен механизм программно-управляемой блокировки запросов, который реализуется с помощью набора флажков, разрешающих или запрещающих восприятие запросов на прерывание процессором. Эти флажки образуют маску прерываний и упаковываются в отдельный регистр маски прерываний, либо входят в состав регистра флагов (признаков) процессора.

Существуют две системы прерываний:

– радиальная система прерываний;

– векторная система прерываний.

Организация радиальной системы прерываний. Физический интерфейс простейшей системы прерываний может быть представлен единственной линией запроса на прерывание IRQ (Interrupt ReQuest). Для программиста такая система прерываний представляется в виде отдельной точки входа в процедуру обслуживания.

Формирование запросов на прерывание – запросов ПУ на обслуживание – происходит в контроллерах соответствующих ПУ (рис. 25).

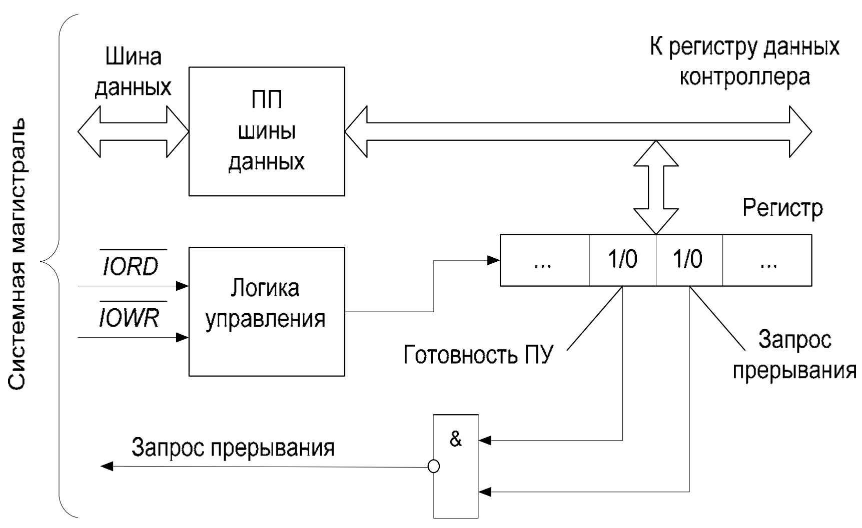


Рис. 25. Простейшая система прерываний

В простейших случаях в качестве сигнала запроса прерывания может использоваться сигнал готовности ПУ, поступающий из контроллера ПУ с выхода триггера готовности (соответствующий разряд регистра состояния). Однако такое простое решение обладает существенным недостатком – процессор не имеет возможности управлять прерываниями, т. е. разрешать или запрещать их для отдельных ПУ. В результате организация обмена данными в режиме прерывания с несколькими ПУ существенно усложняется. Поэтому регистр состояния контроллера ПУ дополняют еще одним разрядом – разрешение прерывания. Запись 1 или 0 в этот разряд регистра состояния производится программным путем (прямой вывод) по одной из линий шины данных системной магистрали. Управляющий сигнал ЗАПРОС ПРЕРЫВАНИЯ от ПУ формируется с помощью схемы совпадения только при наличии единиц в разрядах «Готовность ПУ» и «Разрешение прерывания» регистра состояния контроллера ПУ.

При необходимости обслуживания нескольких ПУ в такой системе сигналы запросов на прерывание от всех ПУ поступают на один вход процессора (запросы объединяются по схеме «монтажное ИЛИ»). В этом слу-

чае возникает проблема идентификации ПУ, запросившего обслуживания. Данная проблема решается с помощью специальной процедуры, называемой поллингом. Функция поллинга состоит в последовательном опросе состояния всех ПУ и выявлении готовых к обслуживанию. В данном случае поллинг реализуется программным способом путем анализа разряда готовности регистров состояния контроллеров ПУ.

Организация прерываний с программным опросом готовности предполагает наличие в памяти микропроцессорной системы единой подпрограммы обслуживания прерываний от всех ПУ. Структура такой подпрограммы приведена на рис. 26. Обслуживание ПУ с помощью единой подпрограммы обработки прерываний производится следующим образом.

После выполнения очередной команды основной программы процессор проверяет наличие запроса на прерывание от ПУ. Если запрос прерывания есть и в процессоре прерывание разрешено, то МП переключается на выполнение подпрограммы обработки прерываний.

После сохранения содержимого регистров процессора, используемых в подпрограмме, начинается последовательный опрос регистров состояния контроллеров всех ПУ, работающих в режиме прерывания. Как только подпрограмма обнаружит готовое к обмену ПУ, сразу выполняются действия по его обслуживанию. Завершается подпрограмма обработки прерывания после опроса готовности всех ПУ восстановлением содержимого регистров процессора.

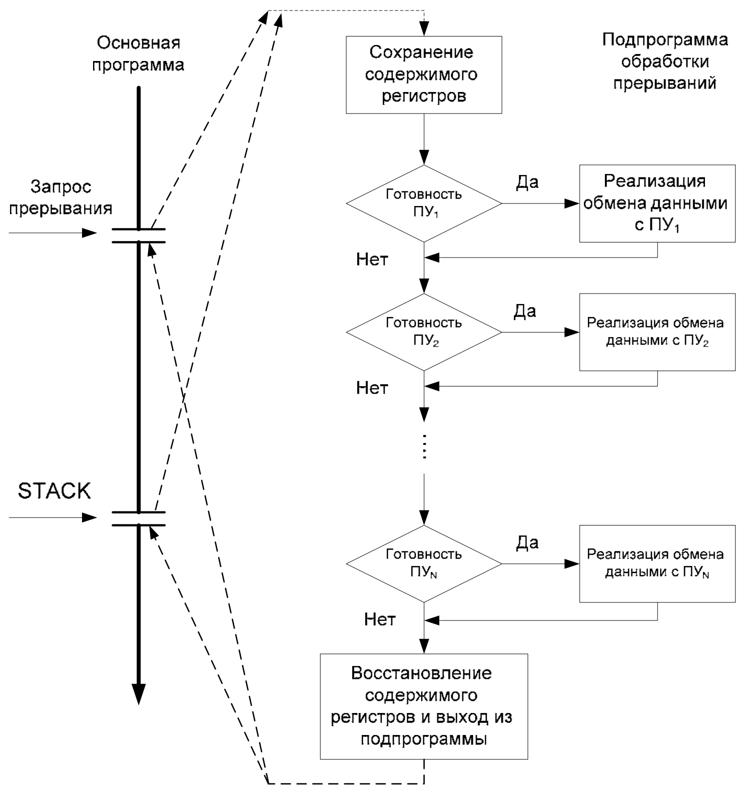


Рис. 26. Подпрограмма обслуживания прерываний

Приоритет ПУ в системе с программным опросом готовности ПУ однозначно определяется порядком их опроса в подпрограмме обработки прерывания. Чем раньше в подпрограмме опрашивается готовность ПУ, тем меньше время реакции на его запрос и выше приоритет. Приоритет запроса отражает важность и срочность его обслуживания. При назначении приоритетов учитываются частота появления запроса, длительность процесса обслуживания, последствия задержки обслуживания и др. Необходимость проверки готовности всех ПУ существенно увеличивает время обслуживания тех ПУ, которые опрашиваются последними. Это является основным недостатком рассматриваемого способа организации прерываний. Поэтому обслуживание прерываний с опросом готовности ПУ используется только в тех случаях, когда отсутствуют жесткие требования на время обработки запросов прерываний ПУ.

Для увеличения числа одновременно обслуживаемых источников прерываний в процессоре вводится несколько линий с фиксированными стартовыми адресами подпрограмм обслуживания. Такие линии называются *радиальными*, а система прерываний – *радиальной системой прерываний.* В радиальной системе прерываний проблема идентификации ПУ решается путем выделения каждому ПУ своей линии для передачи запроса на прерывание. В зависимости от того, по какой линии приходит запрос на прерывание, управление передается подпрограмме обслуживания соответствующего ПУ.

Обычно часть радиальных линий отводится для приема внутренних прерываний процессора, отражающих его критические состояния и требующих немедленного обслуживания. Остальные отводятся для приема внешних (по отношению к процессору) запросов.

В зависимости от числа запросов, одновременно находящихся на обслуживании, различают одно- и многоуровневые системы прерываний. В одноуровневой системе прерываний в каждый момент времени допускается только один подтвержденный запрос. При этом обработка всех других запросов откладывается до окончания текущего обслуживания. Если несколько устройств одновременно запросили обслуживание, процессор выбирает только одно из них на основании приоритета каждого из запросов.

Наиболее практичной и естественной является система прерываний с фиксированными линейно упорядоченными приоритетами. В такой системе все приоритеты строго упорядочены, что обеспечивает однозначный выбор одного из них. Приоритеты запросов могут динамически изменяться по заданному алгоритму (аппаратно или программно), однако в каждый момент времени все приоритеты остаются строго упорядоченными. Широко применяемым алгоритмом изменения приоритетов является циклический. В соответствии с ним после каждого очередного обслуживания за-

проса происходит циклический сдвиг приоритетов с присвоением минимального приоритета только что обработанному запросу. Такая схема приводит к равномерному распределению внимания процессора между всеми запросами и может быть использована при обслуживании группы одинаковых устройств, когда выделение какого-либо из них нежелательно или их нельзя однозначно упорядочить по приоритетам.

Многоуровневая система разрешает многократные (по числу уровней) прерывания одних процедур обслуживания другими. Каждый уровень связывается с определенной радиальной линией и строго упорядоченным приоритетом. Процедура обслуживания некоторого уровня может быть прервана только запросами более высокого уровня. Фоновую работу процессора связывают с минимальным приоритетом, ее может прервать любой запрос.

Для работы многоуровневой системы прерываний необходимо знать приоритет текущей процедуры. Для этого вводится понятие приоритет процессора. Приоритет процессора всегда равен приоритету текущей выполняемой процедуры: при выполнении фоновой задачи процессор имеет минимальный приоритет, при выполнении процедуры обслуживания запроса некоторого уровня приоритет процесса принимает значение, равное приоритету этого уровня. При поступлении запроса на прерывание его приоритет сравнивается с приоритетом процессора и, если приоритет запроса выше, то такой запрос прерывает работу процессора. При этом приоритет процессора повышается, принимая значение, равное приоритету запроса.

Каждая внешняя радиальная линия с фиксированным адресом подпрограммы обработки прерывания может использоваться для приема запросов от нескольких источников прерываний. В этом случае линии запросов на прерывание от ПУ объединяются по схеме «монтажное ИЛИ» и подключаются к некоторой радиальной линии процессора. В этом случае после принятия общего запроса к обслуживанию возникает задача идентификации источника, выставившего запрос, и передачи управления соответствующей процедуре обслуживания. Эта задача решается программным способом с помощью процедуры поллинга, как это было рассмотрено ранее.

Организация векторной системы прерываний. Повышение эффективности системы прерываний связано с передачей функции идентификации ПУ, запросившего обслуживания, внешним по отношению к процессору средствам. В векторной системе прерываний ПУ, запросившее обслуживание, само идентифицирует себя с помощью вектора прерывания, который принимается МП. Для передачи вектора прерывания необходима специальная шина. Однако она, как правило, физически совмещается с шиной данных системной магистрали, при этом ввод вектора прерывания осуществляется в специальном цикле магистрали, который называется

циклом подтверждения прерывания. Такое совмещение требует включения в шину управления линии подтверждения прерывания INTA, по которой передается сигнал от процессора, разрешающий выдачу вектора прерывания в ответ на запрос прерывания от ПУ. Микропроцессор, получив вектор прерывания, сразу переключается на выполнение требуемой подпрограммы обработки прерывания. Так же как и радиальная система, векторная система прерываний предполагает наличие для каждого ПУ собственной подпрограммы обработки прерывания. При этом вектор прерывания определяет, какой подпрограмме обработки прерывания процессор должен передать управление. Вектор прерывания может представлять собой:

– полную команду вызова подпрограммы вместе с адресом подпрограммы обработки прерывания;

– адрес подпрограммы обработки прерывания;

– указатель на адрес подпрограммы обработки прерывания, в качестве которого может использоваться либо адрес, по которому в памяти хранится адрес подпрограммы обработки прерывания (иногда такой указатель называют адресом вектора прерывания), либо тип прерывания.

В современных микропроцессорных системах в качестве вектора прерывания, как правило, используется тип прерывания, по которому процессор определяет адрес подпрограммы обработки прерывания. Преобразование типа (вектора) прерывания в стартовый адрес подпрограммы обработки прерывания выполняется с помощью специальной таблицы прерываний IDT (Interrupt Descriptor Table). При этом вектор рассматривается как индекс таблицы, которая размещается в системной памяти, начиная с некоторого (обычно нулевого) адреса. В процессоре может быть специальный регистр IDTR, позволяющий размещать таблицу прерываний в любой части адресного пространства памяти. Таблица прерываний кроме стартового адреса может содержать дополнительную информацию, например вектор начального состояния подпрограммы обработки прерывания данного типа.

Аналогичным образом могут обрабатываться и запросы радиальных прерываний, если каждому радиальному запросу поставить в соответствие указатель на адрес подпрограммы обработки прерывания. Такие запросы называются *запросами с фиксированными векторами*. В этом случае возможно построение комбинированной системы прерываний, в которой кроме линии для приема векторного запроса имеются линии для приема радиальных запросов с фиксированными векторами. Радиальные запросы с фиксированными векторами соотносятся с некоторыми векторными прерываниями, однако при их обработке передачи (соответственно и приема МП) вектора прерывания не происходит. Такой подход позволяет совместить достоинства обоих систем прерывания.

Существует два подхода к построению векторной системы прерываний, которые различаются используемым *методом формирования вектора прерывания*. Первый подход использует *метод децентрализованного управления* – определение запроса с наибольшим приоритетом и формирование вектора прерывания осуществляется непосредственно ПУ. Второй подход использует метод централизованного управления и состоит в передаче функции формирования вектора прерывания специальному устройству – контроллеру прерывания.

При формировании вектора прерывания средствами ПУ логика работы программного поллинга переносится на аппаратные средства – определение наиболее приоритетного запроса осуществляется с помощью аппаратного опроса готовности ПУ. Такой подход называется аппаратным поллингом. Линии запросов от всех ПУ объединяются по схеме «монтажное ИЛИ» и подключаются к общей линии запроса прерывания IRQ процессора (рис 27).

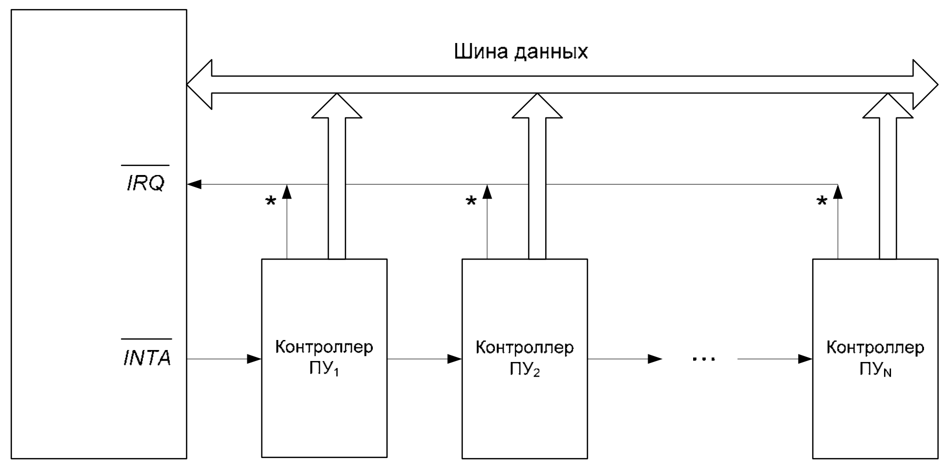


Рис. 27. Формирование вектора прерывания средствами ПУ

Процессор при поступлении в него по линии IRQ запроса прерывания формирует управляющий сигнал подтверждения прерывания INTA, который поступает сначала в контроллер ПУ с наивысшим приоритетом. Если это ПУ не требовало обслуживания, то его контроллер пропускает сигнал подтверждения прерывания на следующий контроллер, иначе дальнейшее распространение сигнала прекращается и контроллер выдает в шину данных вектор прерывания. Такая схема носит ярко выраженный шлейфовый характер. Одна линия подтверждения прерывания проходит последовательно через контроллеры ПУ и образует последовательную приоритетную структуру, называемую дейзи-цепочкой. Приоритет определяется физическим положением каждого ПУ. Ближайшее к процессору ПУ имеет наибольший приоритет. В этой структуре в каждом контроллере ПУ есть схема, которая включается в дейзи-цепочку (рис. 28).

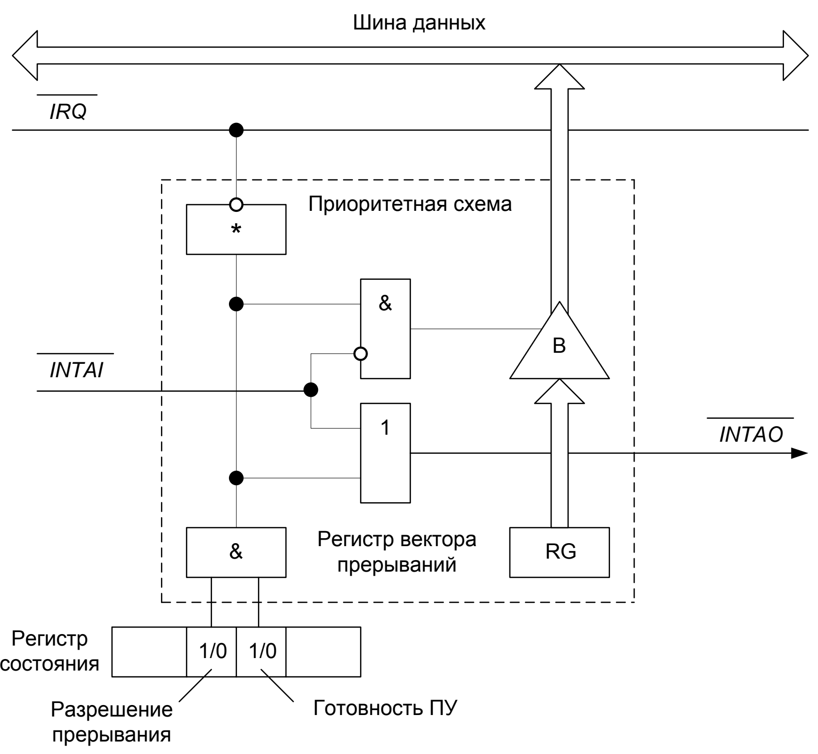


Рис. 28. Включение ПУ в дейзи-цепочку

Эта схема имеет вход INTAI , которая соединяется с выходом INTAO такой же схемы предыдущего контроллера ПУ в дейзи-цепочке с большим приоритетом. На вход INTAI первого контроллера в дейзи-цепочке поступает сигнал подтверждения прерывания INTA от процессора. При наличии хотя бы одного ПУ, запрашивающего обслуживания, в МП поступает запрос прерывания. В ответ МП формирует сигнал подтверждения прерывания. Этот сигнал поступает на вход приоритетной схемы контроллера ПУ, где он объединяется по схеме ИЛИ с не инвертированным сигналом запроса на прерывание данного ПУ. Если данное ПУ запрашивает обслуживание (не инвертированный сигнал запроса прерывания находится в высоком уровне), то входной сигнал подтверждения прерывания INTAI дальше не распространяется – на выходе распространяется INTO высокий уровень. Одновременно не инвертированный сигнал запроса прерывания объединяется по И с инвертированным входным сигналом подтверждения прерывания INTAI, в результате формируется сигнал, который выдает на шину данных вектор прерывания из соответствующего регистра. При этом ПУ с более низкими приоритетами, т. е. более удаленные в дейзи-цепочке, не получают подтверждения прерывания, даже если они тоже запрашивают обслуживание. Если данное ПУ не запрашивает обслуживание (не инвертированный сигнал запроса прерывания находится в низком уровне), то входной сигнал подтверждения прерывания INTAI распространяется дальше на выход INTO и поступает на контроллер следующего в дейзи-це-

почке ПУ. При этом не формируется сигнал, выдающий на шину данных вектор прерывания данного ПУ.

Дейзи-цепочка имеет два преимущества. Во-первых, в системной магистрали нужна только одна линия запроса прерывания, если схемы запросов от ПУ имеют выход с открытым коллектором (одна линия запроса используется и в системе с программным опросом готовности ПУ, однако аппаратный опрос готовности ПУ производится гораздо быстрее). Во-вторых, в систему можно ввести новое ПУ с любым требуемым приоритетом, просто подключая его в нужную физическую позицию. Количество ПУ в системе ограничивается только числом векторов прерываний. Однако дейзи-цепочка медленнее параллельного способа, реализуемого в контроллере прерываний, так как сигнал подтверждения прерывания распространяется через каждое ПУ. Еще один недостаток шлейфовой структуры – трудность управления приоритетами. Стоящие в дейзи-цепочке ближе к процессору ПУ обладают более высоким приоритетом, поэтому изменение приоритетов требует изменения последовательности включения ПУ, что во многих случаях затруднительно.

Наиболее эффективно векторная система прерываний реализуется с помощью контроллера прерываний КПР (рис. 29).

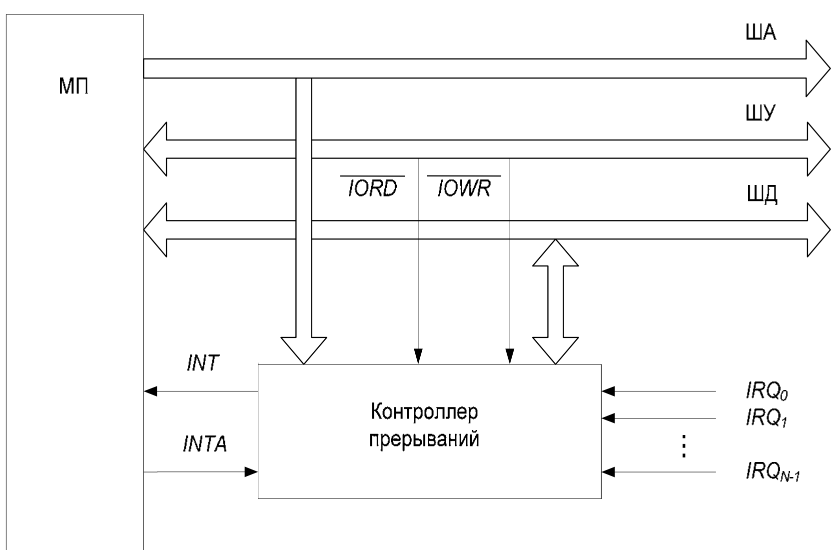


Рис. 29. Векторная система прерываний с использованием контроллера прерываний

Контроллер прерываний может рассматриваться как расширение процессора, по этой причине его иногда называют сопроцессором обработки прерываний. На основе КПР формируется многоуровневая приоритетная система векторных прерываний. КПР обеспечивает прием и обработку N запросов на прерывание. Приоритетная логика КПР выбирает запрос на прерывание с наивысшим приоритетом из числа поступивших и сравнивает его с текущим приоритетом запроса, находящегося на обслу-

живании. При превышении текущего приоритета КПР генерирует сигнал запроса прерывания INT, который поступает в процессор. МП подтверждает прием запроса INT генерацией сигнала подтверждение прерывания INTA, в ответ на который КПР выдает на шину данных системной магистрали соответствующий вектор прерывания. До тех пор, пока некоторый запрос находится в обслуживании, все запросы с равным или меньшим приоритетом игнорируются. В то же время запросы с более высоким приоритетом приводят к генерации сигнала INT, инициируя вложенные прерывания МП. Для оперативного управления работой контроллера предусматривается возможность его программирования, что позволяет динамически изменять приоритеты запросов, формируемые вектора прерываний и т. п.

Рассмотрим организацию контроллера прерываний. Логика управления векторными приоритетными прерываниями в одноуровневой системе показана на рис. 30.

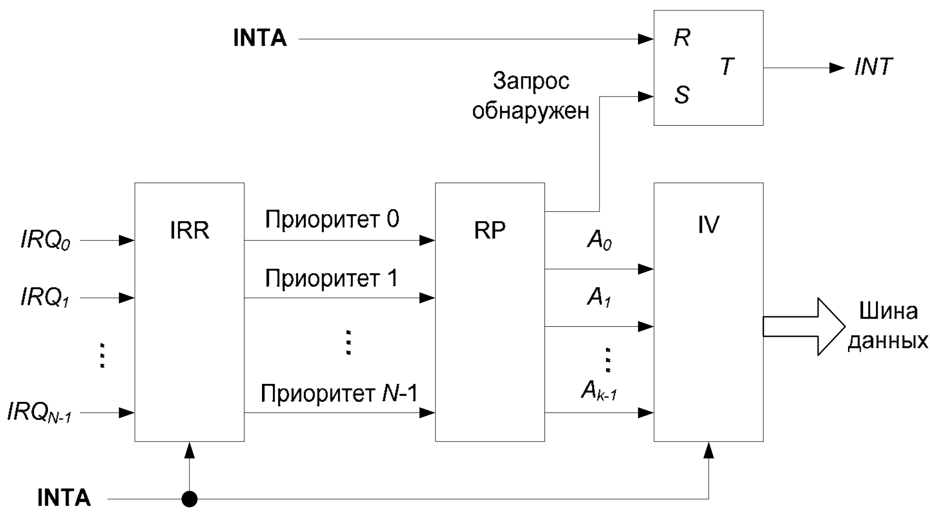


Рис. 30. Организация контроллера прерываний в одноуровневой системе

Запросить прерывание ПУ может в любой момент, формируя сигнал на своей линии запроса прерывания. Запросы от ПУ запоминаются в регистре запросов прерываний IRR (Interrupt Request Register). Шифратор приоритетов PR выбирает запрос с наибольшим приоритетом и преобразует его в k-разрядный код $A_{k-1}…A_1A_0$, представляющий собой приоритетный уровень выбранного запроса ($k=\log_2 N$, где N – число приоритетных уровней). Одновременно шифратор приоритетов формирует сигнал «Запрос обнаружен», который устанавливает в единицу триггер запроса прерывания. Сигнал с выхода этого триггера поступает на соответствующий вход МП INT. Если запросов от ПУ нет, сигнал «Запрос обнаружен» на выходе шифратора приоритетов неактивен, и запрос на прерывание в МП не генерируется. Сформированный шифратором приоритетов код $A_{k-1}…A_1A_0$ поступает на схему формирования вектора прерывания IV, выход которой

подключается к шине данных системной магистрали. МП, получив запрос от КПР, формирует в ответ сигнал подтверждения прерывания INTA, указывающий, что процессор готов принимать вектор прерывания. По этому сигналу вектор прерывания с выхода схемы формирования вектора прерывания поступает на шину данных, и МП принимает его в цикле подтверждения прерывания. Одновременно сигнал INTA сбрасывает триггер запроса прерывания и разряд регистра IRR, соответствующий принятому к обслуживанию запросу.

Рассмотренный контроллер прерываний может использоваться только в одноуровневой системе прерываний, т. к. предполагается, что МП не реагирует на новые запросы до окончания обслуживания текущего запроса. В многоуровневой системе допускаются вложенные прерывания. Для этого логика приоритетов КПР должна выбирать запрос на прерывание с наивысшим приоритетом из числа поступивших и сравнивать его приоритет с текущим приоритетом обслуживаемого запроса. Только при превышении приоритета поступившего запроса КПР генерирует сигнал INT в МП. Это реализуется введением в логику управления приоритетными векторными прерываниями регистра обслуживания ISR (In Service Register), еще одного шифратора приоритетов и компаратора (рис. 31).

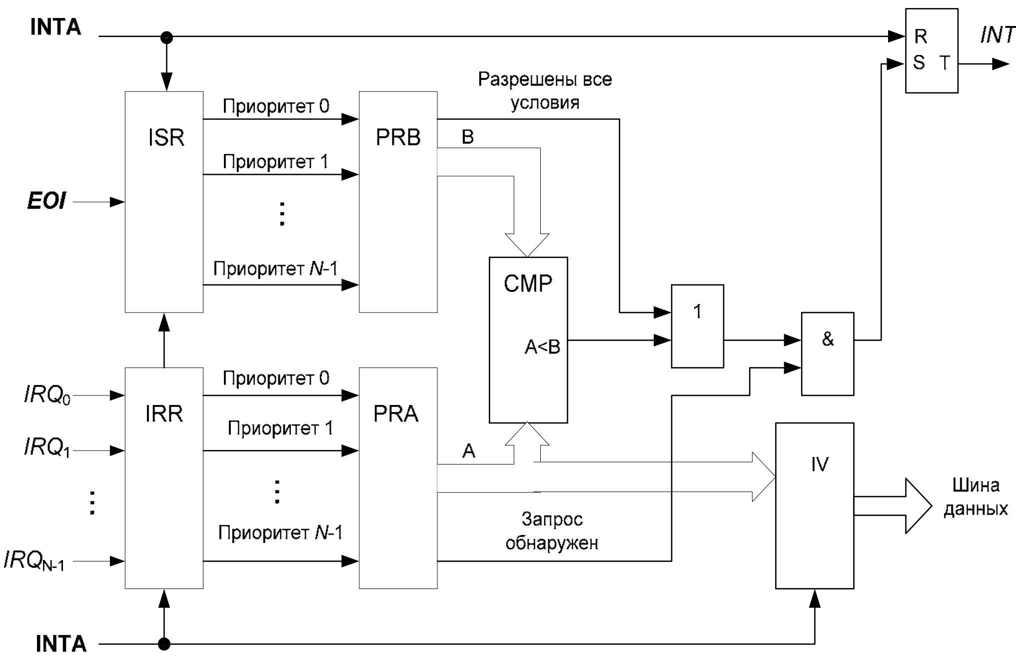


Рис. 31. Организация контроллера прерываний в многоуровневой системе

Регистр обслуживания ISR содержит все запросы, которые в данное время находятся на обслуживании. Шифратор приоритетов PRB выбирает среди запросов, находящихся на обслуживании, запрос с наибольшим приоритетом и преобразует его в k-разрядный код $B_{k-1}...B_1B_0$, представляющий собой приоритетный уровень выбранного запроса. Компаратор срав-

нивает его с кодом $A_{k-1}\ldots A_1A_0$. Если $A_{k-1}\ldots A_1A_0 < B_{k-1}\ldots B_1B_0$, КПР генерирует сигнал INT в МП. Процессор подтверждает прием запроса INT формированием сигнала подтверждения прерывания INTA, под воздействием которого запрос с высшим приоритетом (поступивший запрос принимаемый к обслуживанию) из IRR фиксируется в соответствующем разряде ISR. Разряд в регистре IRR, соответствующий принятому к обслуживанию запросу, сбрасывается и разрешается прием нового запроса. Одновременно с этим КПР выдает вектор прерывания, который принимается МП. Установленный в ISR бит остается в состоянии 1 до окончания процедуры обслуживания. В конце процедуры обслуживания в КПР должна быть передана специальная команда окончания прерывания EOI, которая сбрасывает соответствующий бит в регистре ISR. До тех пор, пока некоторый бит в регистре ISR установлен в 1, все запросы с равным или меньшим приоритетом игнорируются. В то же время запросы с более высоким приоритетом приводят к формированию сигнала INT, инициируя вложенные прерывания МП.

5. Прямой доступ к памяти. Организация прямого доступа к памяти. Контроллер ПДП

Прямой доступ к памяти (ПДП) является одним из способов обмена данными с ПУ. В этом режиме обмен данными между ПУ и памятью микропроцессорной системы происходит без участия процессора. Обменом в режиме ПДП управляет не программа, выполняемая процессором, а внешнее по отношению к процессору специальное устройство, называемое контроллером ПДП (КПДП). Прямой доступ к памяти используется для быстрого ввода/вывода блоков данных и разгрузки процессора от управления операциями ввода/вывода. Обмен блоками данных с помощью программно-управляемого обмена осуществляется относительно медленно, так как на обмен каждым байтом затрачивается несколько команд процессора. Прямой доступ к памяти освобождает процессор от управления операциями ввода/вывода, позволяя тем самым осуществлять параллельно во времени выполнение процессором программы с обменом данными между ПУ и памятью, производить этот обмен со скоростью, ограниченной только пропускной способностью памяти или ПУ. Таким образом, ПДП, разгружая процессор от обслуживания операций ввода/вывода, способствует возрастанию общей производительности микропроцессорной системы.

Для реализации режима ПДП необходимо обеспечить непосредственную связь контроллера ПДП и памяти микропроцессорной системы, т. е. специальный информационный канал, по которому осуществляется обмен в режиме ПДП, – канал ПДП. Для этой цели можно использовать специально выделенную магистраль, связывающую контроллер ПДП с па-

мятью. Однако такое решение нельзя признать оптимальным, так как это приведет к значительному усложнению микропроцессорной системы в целом, особенно при подключении нескольких ПУ. С целью сокращения количества линий в шинах микропроцессорной системы контроллер ПДП подключается к памяти посредством шин системной магистрали. При этом возникает проблема совместного использования шин системной магистрали процессором и контроллером ПДП. Можно выделить два основных способа ее решения:

– реализация обмена в режиме ПДП с захватом цикла;
– реализация обмена в режиме ПДП с блокировкой процессора.

Существует две разновидности прямого доступа к памяти с захватом цикла. Наиболее простой способ организации ПДП состоит в том, что для обмена используются те циклы процессора, в которых он не обменивается данными с памятью. В такие циклы контроллер ПДП может обмениваться данными с памятью, не мешая работе процессора. Однако возникает необходимость выделения таких циклов, чтобы не произошло временного перекрытия обмена ПДП с операциями обмена, инициируемыми процессором. В некоторых процессорах формируется специальный управляющий сигнал, указывающий циклы, в которых процессор не использует память. Если процессор не формирует такого сигнала, то для выделения свободных циклов необходимо применение в контроллере ПДП специальной схемы, что приводит к усложнению последнего. Применение этого способа организации ПДП не снижает производительности системы, но при этом обмен в режиме ПДП возможен только в случайные моменты времени одиночными словами.

Наиболее распространенным является ПДП с захватом цикла и принудительным отключением процессора от шин системной магистрали. Для реализации такого режима ПДП системная магистраль дополняется двумя управляющими сигналами – «Требование прямого доступа к памяти» HOLD и «Предоставление прямого доступа к памяти» HLDA.

Управляющий сигнал HOLD формируется контроллером ПДП. Процессор, получив этот сигнал, приостанавливает выполнение текущей команды, не дожидаясь ее завершения, отключается от шин системной магистрали и выдает контроллеру ПДП управляющий сигнал HLDA. С этого момента все шины системной магистрали управляются контроллером ПДП. Контроллер ПДП, используя шины системной магистрали, осуществляет обмен одним словом данных с памятью и затем, сняв сигнал HOLD, возвращает управление системной магистралью процессору. Как только контроллер ПДП будет готов к обмену следующим словом данных, он вновь захватывает цикл процессора и т. д. В промежутках между захватами цик-

лов контроллером ПДП процессор продолжает выполнять команды программы. Тем самым выполнение программы замедляется, но значительно в меньшей степени, чем при обмене в режиме прерывания.

Передача блока данных с использованием ПДП предполагает выполнение определенной последовательности действий (рис. 32):

1) начальная установка (предварительная подготовка) контроллера ПДП;
2) запуск контроллера ПДП;
3) многократное занятие цикла процессора;
4) завершение обмена.

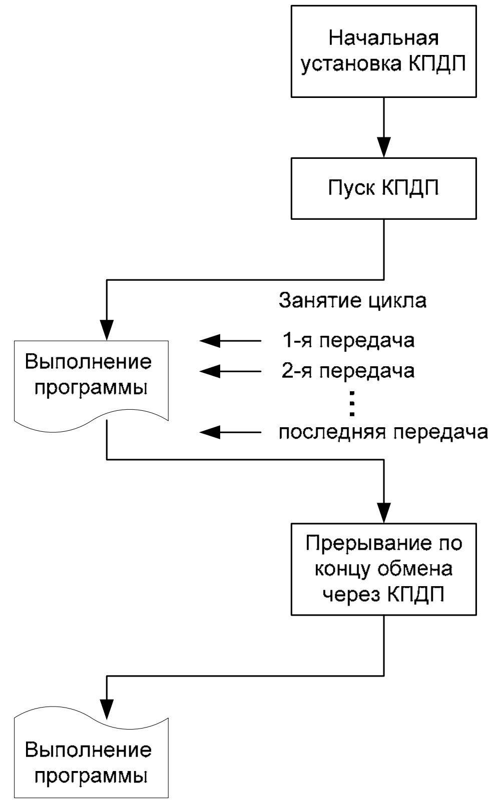


Рис. 32. Алгоритм передачи блока данных с использованием контроллера ПДП

Программа используется только для начальной установки и пуска обмена через канал ПДП. После этого процессор может выполнять основную программу, которая не связана с обменом. Во время выполнения этой программы каждый раз при поступлении запроса на ПДП контроллер ПДП будет занимать цикл процессора, и осуществлять передачу. После окончания обмена для передачи управления программе завершения обмена в режиме ПДП используется прерывание. Затем основная программа может быть продолжена.

Рассмотрим организацию контроллера ПДП, обеспечивающего ввод данных в память микропроцессорной системы в режиме ПДП (рис. 33).

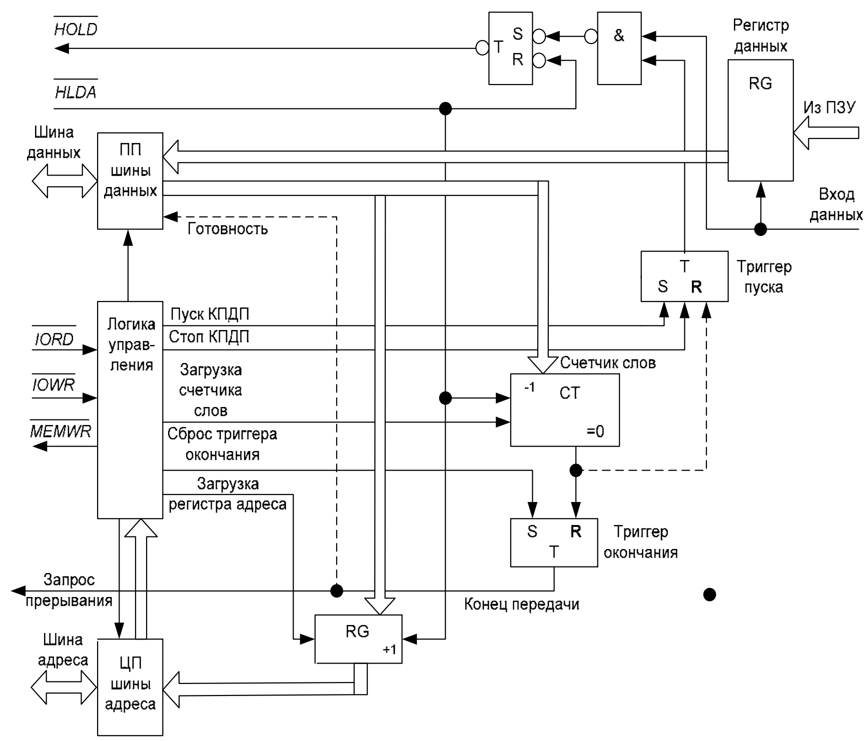


Рис. 33 Организация контроллера ПДП

Начальная подготовка к обмену в режиме ПДП состоит в выделении ПУ области памяти, используемой при обмене, и указании ее размера, т. е. количества записываемых в память или читаемых из памяти слов информации. Следовательно, контроллер ПДП должен иметь в своем составе регистр адреса и счетчик слов. Перед началом обмена с ПУ в режиме ПДП процессор должен выполнить программу загрузки, которая обеспечивает запись в указанные регистры контроллера ПДП начального адреса выделенной ПУ памяти и ее размера в словах заданной разрядности.

Таким образом, перед началом ввода из ПУ очередного блока данных процессор загружает в регистры контроллера ПДП следующую информацию: в счетчик слов – количество принимаемых слов, а в регистр адреса – начальный адрес области памяти для вводимых данных. Тем самым контроллер подготавливается к выполнению операции ввода данных из ПУ в память в режиме ПДП.

Запуск контроллера ПДП осуществляется командой вывода, по которой устанавливается в 1 триггер пуска. Триггер пуска подключает ПУ к контроллеру ПДП. После команды пуска контроллера ПДП должна быть команда разрешения прерывания. В дальнейшем ввод блока данных через канал ПДП осуществляется без участия команд программы.

Когда ПУ подготовит слово данных, оно посылается в регистр данных контроллера. При этом каждое слово сопровождается управляющим сигналом из ПУ «Ввод данных», который обеспечивает запись слова данных в регистр данных контроллера. По этому же сигналу (при установлен-

ном в 1 триггере пуска) устанавливается в 1 триггер запроса, сигнал с выхода которого поступает на вход процессора «Требование прямого доступа к памяти» HOLD. После формирования процессором ответного сигнала «Предоставление прямого доступа к памяти» HLDA следующий машинный цикл занимается под обмен. При этом осуществляется одна пересылка слова данных в ячейку памяти по адресу, находящемуся в регистре адреса контроллера. По сигналу HLDA контроллер выставляет на шины адреса и данных системной магистрали содержимое своих регистров адреса и данных соответственно. Формируя управляющий сигнал MEMWR, контроллер ПДП обеспечивает запись слова данных из своего регистра данных в память. Сигнал «Предоставление прямого доступа к памяти» HLDA используется в контроллере также для выполнения трех операций:

- сброса триггера запроса;
- увеличения содержимого регистра адреса на 1;
- уменьшения содержимого счетчика слов на 1.

По каждому сигналу HLDA из содержимого счетчика слов вычитается 1, и когда оно становится равным 0, устанавливается в 1 триггер окончания передачи блока данных, сигнал с выхода которого используется в качестве запроса на прерывание и поступает на соответствующий вход процессора. Процессор прерывает выполнение программы и передает управление подпрограмме обработки прерывания для завершения обмена.

Завершение обмена осуществляется путем отключения ПУ от контроллера ПДП командой вывода, по которой сбрасывается в 0 триггер пуска. Кроме того, аналогичным образом сбрасывается в 0 триггер окончания передачи блока данных. По окончании обработки прерывания управление возвращается основной программе.

Если нет необходимости продолжать выполнение некоторой программы параллельно с передачей в режиме ПДП, используется в качестве сигнала готовности, который доступен процессору через регистр состояния контроллера ПДП. В этом случае прерывание не используется (выход триггера окончания передачи не подключается к входу запроса на прерывание процессора или прерывание в процессоре запрещается). В течение обмена через канал ПДП процессор находится в цикле ожидания окончания передачи, опрашивая соответствующий разряд готовности регистра состояния контроллера ПДП по команде ввода. Как только процессор обнаружит готовность, он переходит к процедуре завершения обмена (шаг 4 рассмотренной выше последовательности), после чего выполнение программы продолжается.

Завершение обмена путем отключения ПУ от контроллера ПДП – сброс в 0 триггера пуска – может осуществляться не по команде вывода в подпрограмме обработки прерывания, а автоматически, когда содержимое

счетчика слов становится равным 0 (также как и установка в 1 триггера окончания передачи). В этом случае в контроллере ПДП отпадает необходимость в триггере пуска, а подключение/отключение ПУ к контроллеру осуществляется в зависимости от состояния счетчика слов. При загрузке в счетчик количества принимаемых слов сигнал «0=» устанавливается в 1 и подключает ко входу S триггера запроса управляющий сигнал из ПУ «Ввод данных». После передачи всех слов содержимое счетчика становится равным 0, сигнал «0=» сбрасывается в 0 и запрещает поступление управляющего сигнала из ПУ «Ввод данных» на вход S триггера запроса, отключая тем самым ПУ от контроллера ПДП.

Выше были рассмотрены только процесс подготовки контроллера ПДП и непосредственно передача данных в режиме ПДП. На практике любой сеанс обмена данными с ПУ в режиме ПДП всегда включает также и этап подготовки ПУ к обмену. На этом этапе процессор в режиме программно-управляемого обмена опрашивает состояние ПУ, проверяя его готовность к обмену, и посылает в ПУ команды, обеспечивающие его подготовку к обмену данными по каналу ПДП. Такая подготовка может сводиться, например, к перемещению головок на требуемую дорожку в НМД. Затем выполняется загрузка регистров контроллера ПДП, после чего обмен данными в режиме ПДП начинается либо по инициативе контроллера ПДП, как это было рассмотрено выше, либо по инициативе ПУ.

Следует отметить, что использование в микропроцессорной системе обмена в режиме ПДП с захватом цикла требует от программиста очень ясного понимания процессов, происходящих в системе при выполнении программы, и четкой синхронизации процесса выполнения программы и ввода /вывода по каналу ПДП.

Прямой доступ к памяти с блокировкой процессора отличается от режима ПДП с захватом цикла тем, что управление системной магистралью передается контроллеру ПДП не на время передачи одного слова, а на время обмена блоком данных. Такой режим ПДП необходим в тех случаях, когда время между двумя сигналами «Требование прямого доступа к памяти» HOLD сопоставимо с циклом процессора. В этом случае процессор не успевает выполнить хотя бы одну команду между очередными операциями обмена в режиме ПДП.

В микропроцессорной системе можно использовать несколько ПУ, работающих в режиме ПДП. Предоставление таким ПУ шин системной магистрали для обмена данными производится на приоритетной основе. В этом случае приоритеты ПУ реализуются так же, как и при обмене данными в режиме прерывания. Как правило, для каждого ПУ используется своя пара управляющих сигналов «Требование прямого доступа к памяти» HOLD и «Предоставление прямого доступа к памяти» HLDA и отдельный канал в контроллере ПДП.

6. Память микропроцессорной системы. Функции памяти. Архитектура и иерархия памяти. Организация кэш-памяти. Виртуальная память

Память микропроцессорной системы представляет собой иерархическую структуру. В основе реализации иерархии памяти современных компьютеров лежат два принципа: принцип локальности обращений и соотношение стоимость/производительность. Принцип локальности обращений говорит о том, что большинство программ к счастью не выполняют обращений ко всем своим командам и данным равновероятно, а оказывают предпочтение некоторой части своего адресного пространства.

Иерархия памяти современных компьютеров строится на нескольких уровнях, причем более высокий уровень меньше по объему, быстрее и имеет большую стоимость в пересчете на байт, чем более низкий уровень. Уровни иерархии взаимосвязаны: все данные на одном уровне могут быть также найдены на более низком уровне, и все данные на этом более низком уровне могут быть найдены на следующем нижележащем уровне и так далее, пока мы не достигнем основания иерархии.

Иерархия памяти обычно состоит из многих уровней, но в каждый момент времени мы имеем дело только с двумя близлежащими уровнями. Минимальная единица информации, которая может либо присутствовать, либо отсутствовать в двухуровневой иерархии, называется блоком. Размер блока может быть либо фиксированным, либо переменным. Если этот размер зафиксирован, то объем памяти является кратным размеру блока.

Успешное или неуспешное обращение к более высокому уровню называются соответственно попаданием (hit) или промахом (miss). *Попадание* – есть обращение к объекту в памяти, который найден на более высоком уровне, в то время как промах означает, что он не найден на этом уровне. Доля попаданий (hit rate) или коэффициент попаданий (hit ratio) есть доля обращений, найденных на более высоком уровне. Иногда она представляется процентами. Доля промахов (miss rate) есть доля обращений, которые не найдены на более высоком уровне.

Поскольку повышение производительности является главной причиной появления иерархии памяти, частота попаданий и промахов является важной характеристикой. Время обращения при попадании (hit time) есть время обращения к более высокому уровню иерархии, которое включает в себя, в частности, и время, необходимое для определения того, является ли обращение попаданием или промахом. Потери на промах (miss penalty) есть время для замещения блока в более высоком уровне на блок из более низкого уровня плюс время для пересылки этого блока в требуемое устройство (обычно в процессор). Потери на промах далее включают в себя две компоненты: время доступа (access time) – время обращения к первому

слову блока при промахе, и время пересылки (transfer time) – дополнительное время для пересылки оставшихся слов блока. Время доступа связано с задержкой памяти более низкого уровня, в то время как время пересылки связано с полосой пропускания канала между устройствами памяти двух смежных уровней.

Чтобы описать некоторый уровень иерархии памяти надо ответить на следующие четыре вопроса:

- Где может размещаться блок на верхнем уровне иерархии? (размещение блока).
- Как найти блок, когда он находится на верхнем уровне? (идентификация блока).
- Какой блок должен быть замещен в случае промаха? (замещение блоков).
- Что происходит во время записи? (стратегия записи).

Организация кэш-памяти. Концепция кэш-памяти возникла довольно давно и сегодня кэш-память имеется практически в любом классе компьютеров, и чаще всего многоуровневая. Все термины, которые были определены раньше могут быть использованы и для кэш-памяти, хотя слово «строка» (line) часто употребляется вместо слова «блок» (block).

В таблице 1 представлен типичный набор параметров, который используется для описания кэш-памяти.

Таблица 1

Типовые значения ключевых параметров для кэш-памяти персональных компьютеров

Размер блока (строки)	4 – 128 байт
Время попадания (hit time)	1 – 4 такта синхронизации (обычно 1 такт)
Потери при промахе (miss penalty) (Время доступа – access time) (Время пересылки – transfer time)	8 – 32 такта синхронизации (6 – 10 тактов синхронизации) (2 – 22 такта синхронизации)
Доля промахов (miss rate)	1% – 20%
Размер кэш-памяти	4 Кбайт – 16 Мбайт

Рассмотрим организацию кэш-памяти более детально.

Принципы размещения блоков в кэш-памяти определяют три основных типа их организации:

1. Если каждый блок основной памяти имеет только одно фиксированное место, на котором он может появиться в кэш-памяти, то такая кэш-память называется кэшем с прямым отображением (direct mapped). Это наиболее простая организация кэш-памяти, при которой для отображения адресов блоков основной памяти на адреса кэш-памяти просто используются младшие разряды адреса блока. Таким образом, все блоки основной памяти,

имеющие одинаковые младшие разряды в своем адресе, попадают в один блок кэш-памяти.

2. Если некоторый блок основной памяти может располагаться на любом месте кэш-памяти, то кэш называется полностью ассоциативным (fully associative).

3. Если некоторый блок основной памяти может располагаться на ограниченном множестве мест в кэш-памяти, то кэш называется множественно-ассоциативным (set associative). Обычно множество представляет собой группу из двух или большего числа блоков в кэше. Если множество состоит из n блоков, то такое размещение называется множественно-ассоциативным с n каналами (n-way set associative). Для размещения блока прежде всего необходимо определить множество. Множество определяется младшими разрядами адреса блока памяти (индексом). Далее, блок может размещаться на любом месте данного множества.

Диапазон возможных организаций кэш-памяти очень широк: кэш-память с прямым отображением есть просто одноканальная множественно-ассоциативная кэш-память, а полностью ассоциативная кэш-память с m блоками может быть названа m-канальной множественно-ассоциативной.

У каждого блока в кэш-памяти имеется адресный тег, указывающий, какой блок в основной памяти данный блок кэш-памяти представляет. Эти теги обычно одновременно сравниваются с выработанным процессором адресом блока памяти.

Кроме того, необходим способ определения того, что блок кэш-памяти содержит достоверную или пригодную для использования информацию. Наиболее общим способом решения этой проблемы является добавление к тегу так называемого бита достоверности (valid бит).

Адресация множественно-ассоциативной кэш-памяти осуществляется путем деления адреса, поступающего из процессора, на три части: поле смещения используется для выбора байта внутри блока кэш-памяти, поле индекса определяет номер множества, а поле тега используется для сравнения. Если общий размер кэш-памяти зафиксировать, то увеличение степени ассоциативности приводит к увеличению количества блоков в множестве, при этом уменьшается размер индекса и увеличивается размер тега.

При возникновении промаха, контроллер кэш-памяти должен выбрать подлежащий замещению блок. Польза от использования организации с прямым отображением заключается в том, что аппаратные решения здесь наиболее простые. Выбирать просто нечего: на попадание проверяется только один блок и только этот блок может быть замещен. При полностью ассоциативной или множественно-ассоциативной организации кэш-памяти имеются несколько блоков, из которых надо выбрать кандидата в случае промаха. Как правило, для замещения блоков применяются две основных стратегии: случайная и LRU.

В первом случае, чтобы иметь равномерное распределение, блоки-кандидаты выбираются случайно. В некоторых системах, чтобы получить воспроизводимое поведение, которое особенно полезно во время отладки аппаратуры, используют псевдослучайный алгоритм замещения.

Во втором случае, чтобы уменьшить вероятность выбрасывания информации, которая скоро может потребоваться, все обращения к блокам фиксируются. Заменяется тот блок, который не использовался дольше всех (LRU – Least-Recently Used).

Достоинство случайного способа заключается в том, что его проще реализовать в аппаратуре. Когда количество блоков для поддержания трассы увеличивается, алгоритм LRU становится все более дорогим и часто только приближенным. В таблице 2 показаны различия в долях промахов при использовании алгоритма замещения LRU и случайного алгоритма.

Таблица 2

Сравнение долей промахов для алгоритма LRU и случайного алгоритма замещения при нескольких размерах кэша и разных ассоциативностях при размере блока 16 байт

Ассоциативность:	2-канальная		4-канальная		8-канальная	
Размер кэш-памяти	LRU	Random	LRU	Random	LRU	Random
16 KB	5.18%	5.69%	4.67%	5.29%	4.39%	4.96%
64 KB	1.88%	2.01%	1.54%	1.66%	1.39%	1.53%
256 KB	1.15%	1.17%	1.13%	1.13%	1.12%	1.12%

При обращениях к кэш-памяти на реальных программах преобладают обращения по чтению. Все обращения за командами являются обращениями по чтению и большинство команд не пишут в память. Обычно операции записи составляют менее 10% общего трафика памяти. Желание сделать общий случай более быстрым означает оптимизацию кэш-памяти для выполнения операций чтения, однако при реализации высокопроизводительной обработки данных нельзя пренебрегать и скоростью операций записи. К счастью, общий случай является и более простым. Блок из кэш-памяти может быть прочитан в то же самое время, когда читается и сравнивается его тег. Таким образом, чтение блока начинается сразу, как только становится доступным адрес блока. Если чтение происходит с попаданием, то блок немедленно направляется в процессор. Если же происходит промах, то от заранее считанного блока нет никакой пользы, правда нет и никакого вреда. Однако при выполнении операции записи ситуация коренным образом меняется. Именно процессор определяет размер записи (обычно от 1 до 8 байтов) и только эта часть блока может быть изменена. В общем случае это подразумевает выполнение над блоком последовательности операций чтение-модификация-запись: чтение оригинала блока, модификацию его части и запись нового значения блока. Более того, моди-

фикация блока не может начинаться до тех пор, пока проверяется тег, чтобы убедиться в том, что обращение является попаданием. Поскольку проверка тегов не может выполняться параллельно с другой работой, то операции записи отнимают больше времени, чем операции чтения.

Очень часто организация кэш-памяти в разных машинах отличается именно стратегией выполнения записи. Когда выполняется запись в кэш-память имеются две базовые возможности:

1) сквозная запись (write through, store through) – информация записывается в два места: в блок кэш-памяти и в блок более низкого уровня памяти;

2) запись с обратным копированием (write back, copy back, store in) – информация записывается только в блок кэш-памяти. Модифицированный блок кэш-памяти записывается в основную память только когда он замещается. Для сокращения частоты копирования блоков при замещении обычно с каждым блоком кэш-памяти связывается так называемый бит модификации (dirty бит). Этот бит состояния показывает был ли модифицирован блок, находящийся в кэш-памяти. Если он не модифицировался, то обратное копирование отменяется, поскольку более низкий уровень содержит ту же самую информацию, что и кэш-память.

Оба подхода к организации записи имеют свои преимущества и недостатки. При записи с обратным копированием операции записи выполняются со скоростью кэш-памяти, и несколько записей в один и тот же блок требуют только одной записи в память более низкого уровня. Поскольку в этом случае обращения к основной памяти происходят реже, вообще говоря требуется меньшая полоса пропускания памяти, что очень привлекательно для мультипроцессорных систем. При сквозной записи промахи по чтению не влияют на записи в более высокий уровень, и, кроме того, сквозная запись проще для реализации, чем запись с обратным копированием. Сквозная запись имеет также преимущество в том, что основная память имеет наиболее свежую копию данных. Это важно в мультипроцессорных системах, а также для организации ввода/вывода.

Когда процессор ожидает завершения записи при выполнении сквозной записи, то говорят, что он приостанавливается для записи (write stall). Общий прием минимизации приостановок по записи связан с использованием буфера записи (write buffer), который позволяет процессору продолжить выполнение команд во время обновления содержимого памяти. Следует отметить, что приостановки по записи могут возникать и при наличии буфера записи.

При промахе во время записи имеются две дополнительные возможности:

1) разместить запись в кэш-памяти (write allocate) (называется также выборкой при записи (fetch on write)). Блок загружается в кэш-память, вслед за чем выполняются действия аналогичные выполняющимся при выполнении записи с попаданием. Это похоже на промах при чтении;

2) не размещать запись в кэш-памяти (называется также записью в окружение (write around)). Блок модифицируется на более низком уровне и не загружается в кэш-память.

Обычно в кэш-памяти, реализующей запись с обратным копированием, используется размещение записи в кэш-памяти (в надежде, что последующая запись в этот блок будет перехвачена), а в кэш-памяти со сквозной записью размещение записи в кэш-памяти часто не используется (поскольку последующая запись в этот блок все равно пойдет в память).

Принципы организации основной памяти в современных компьютерах. Основная память представляет собой следующий уровень иерархии памяти. Основная память удовлетворяет запросы кэш-памяти и служит в качестве интерфейса ввода/вывода, поскольку является местом назначения для ввода и источником для вывода. Для оценки производительности основной памяти используются два основных параметра: задержка и полоса пропускания. Традиционно задержка основной памяти имеет отношение к кэш-памяти, а полоса пропускания или пропускная способность относится к вводу/выводу. В связи с ростом популярности кэш-памяти второго уровня и увеличением размеров блоков у такой кэш-памяти, полоса пропускания основной памяти становится важной также и для кэш-памяти.

Задержка памяти традиционно оценивается двумя параметрами: временем доступа (access time) и длительностью цикла памяти (cycle time). Время доступа представляет собой промежуток времени между выдачей запроса на чтение и моментом поступления запрошенного слова из памяти. Длительность цикла памяти определяется минимальным временем между двумя последовательными обращениями к памяти.

Основная память современных компьютеров реализуется на микросхемах статических и динамических ЗУПВ (Запоминающее Устройство с Произвольной Выборкой). Микросхемы статических ЗУВП (СЗУПВ) имеют меньшее время доступа и не требуют циклов регенерации. Микросхемы динамических ЗУПВ (ДЗУПВ) характеризуются большей емкостью и меньшей стоимостью, но требуют схем регенерации и имеют значительно большее время доступа.

В процессе развития ДЗУВП с ростом их емкости основным вопросом стоимости таких микросхем был вопрос о количестве адресных линий и стоимости соответствующего корпуса. В те годы было принято решение о необходимости мультиплексирования адресных линий, позволившее сократить наполовину количество контактов корпуса, необходимых для передачи адреса. Поэтому обращение к ДЗУВП обычно происходит в два этапа: первый этап начинается с выдачи сигнала RAS – row-access strobe (строб адреса строки), который фиксирует в микросхеме поступивший адрес строки, второй этап включает переключение адреса для указания адреса столбца и подачу сигнала CAS – column-access strobe (строб адреса

столбца), который фиксирует этот адрес и разрешает работу выходных буферов микросхемы. Названия этих сигналов связаны с внутренней организацией микросхемы, которая как правило представляет собой прямоугольную матрицу, к элементам которой можно адресоваться с помощью указания адреса строки и адреса столбца.

Дополнительным требованием организации ДЗУВП является необходимость периодической регенерации ее состояния. При этом все биты в строке могут регенерироваться одновременно, например, путем чтения этой строки. Поэтому ко всем строкам всех микросхем ДЗУПВ основной памяти компьютера должны производиться периодические обращения в пределах определенного временного интервала порядка 8 миллисекунд.

Это требование кроме всего прочего означает, что система основной памяти компьютера оказывается иногда недоступной процессору, так как она вынуждена рассылать сигналы регенерации каждой микросхеме. Разработчики ДЗУПВ стараются поддерживать время, затрачиваемое на регенерацию, на уровне менее 5% общего времени. Обычно контроллеры памяти включают в свой состав аппаратуру для периодической регенерации ДЗУПВ.

В отличие от динамических, статические ЗУПВ не требуют регенерации и время доступа к ним совпадает с длительностью цикла. Для микросхем, использующих примерно одну и ту же технологию, емкость ДЗУВП по грубым оценкам в 4 – 8 раз превышает емкость СЗУПВ, но последние имеют в 8 – 16 раз меньшую длительность цикла и большую стоимость. По этим причинам в основной памяти практически любого компьютера, проданного после 1975 г., использовались полупроводниковые микросхемы ДЗУПВ (для построения кэш-памяти при этом применялись СЗУПВ). Естественно были и исключения, например, в оперативной памяти суперкомпьютеров компании Cray Research использовались микросхемы СЗУПВ.

Для обеспечения сбалансированности системы с ростом скорости процессоров должна линейно расти и емкость основной памяти. В последние годы емкость микросхем динамической памяти учетверялась каждые три года, увеличиваясь примерно на 60% в год. К сожалению скорость этих схем за этот же период росла гораздо меньшими темпами (примерно на 7% в год). В то же время производительность процессоров начиная с 1987 г. практически увеличивалась на 50% в год.

Очевидно, согласование производительности современных процессоров со скоростью основной памяти вычислительных систем остается на сегодняшний день одной из важнейших проблем. Приведенные в предыдущем разделе методы повышения производительности за счет увеличения размеров кэш-памяти и введения многоуровневой организации кэш-памяти могут оказаться не достаточно эффективными с точки зрения стоимости систем. Поэтому важным направлением современных разработок являются методы повышения полосы пропускания или пропускной способности па-

мяти за счет ее организации, включая специальные методы организации ДЗУПВ. Хотя для организации кэш-памяти в большей степени важно уменьшение задержки памяти, чем увеличение полосы пропускания. Однако при увеличении полосы пропускания памяти возможно увеличение размера блоков кэш-памяти без заметного увеличения потерь при промахах. Основными методами увеличения полосы пропускания памяти являются: увеличение разрядности или «ширины» памяти, использование расслоения памяти, использование независимых банков памяти, обеспечение режима бесконфликтного обращения к банкам памяти, использование специальных режимов работы динамических микросхем памяти.

Увеличение разрядности основной памяти. Кэш-память первого уровня во многих случаях имеет физическую ширину шин данных соответствующую количеству разрядов в слове, поскольку большинство компьютеров выполняют обращения именно к этой единице информации. В системах без кэш-памяти второго уровня ширина шин данных основной памяти часто соответствует ширине шин данных кэш-памяти. Удвоение или учетверение ширины шин кэш-памяти и основной памяти удваивает или учетверяет соответственно полосу пропускания системы памяти. Реализация более широких шин вызывает необходимость мультиплексирования данных между кэш-памятью и процессором, поскольку основной единицей обработки данных в процессоре все еще остается слово. Эти мультиплексоры оказываются на критическом пути поступления информации в процессор. Кэш-память второго уровня несколько смягчает эту проблему, т. к. в этом случае мультиплексоры могут располагаться между двумя уровнями кэш-памяти, т. е. вносимая ими задержка не столь критична. Другая проблема, связанная с увеличением разрядности памяти, определяется необходимостью определения минимального объема (инкремента) для поэтапного расширения памяти, которое часто выполняется самими пользователями на месте эксплуатации системы. Удвоение или учетверение ширины памяти приводит к удвоению или учетверению этого минимального инкремента. Наконец, имеются проблемы и с организацией коррекции ошибок в системах с широкой памятью.

Память с расслоением. Наличие в системе множества микросхем памяти позволяет использовать потенциальный параллелизм, заложенный в такой организации. Для этого микросхемы памяти часто объединяются в банки или модули, содержащие фиксированное число слов, причем только к одному из этих слов банка возможно обращение в каждый момент времени. Как уже отмечалось, в реальных системах имеющаяся скорость доступа к таким банкам памяти редко оказывается достаточной. Следовательно, чтобы получить большую скорость доступа, нужно осуществлять одновременный доступ ко многим банкам памяти. Одна из общих методик, используемых для этого, называется расслоением памяти. При расслоении

банки памяти обычно упорядочиваются так, чтобы N последовательных адресов памяти i, i + 1, i + 2, ..., i + N-1 приходились на N различных банков. В i-том банке памяти находятся только слова, адреса которых имеют вид kN + i (где 0 (k (M-1, а M число слов в одном банке). Можно достичь в N раз большей скорости доступа к памяти в целом, чем у отдельного ее банка, если обеспечить при каждом доступе обращение к данным в каждом из банков. Имеются разные способы реализации таких расслоенных структур. Большинство из них напоминают конвейеры, обеспечивающие рассылку адресов в различные банки и мультиплексирующие поступающие из банков данные. Таким образом, степень или коэффициент расслоения определяют распределение адресов по банкам памяти. Такие системы оптимизируют обращения по последовательным адресам памяти, что является характерным при подкачке информации в кэш-память при чтении, а также при записи, в случае использования кэш-памятью механизмов обратного копирования. Однако, если требуется доступ к непоследовательно расположенным словам памяти, призводительность расслоенной памяти может значительно снижаться. Обобщением идеи расслоения памяти является возможность реализации нескольких независимых обращений, когда несколько контроллеров памяти позволяют банкам памяти (или группам расслоенных банков памяти) работать независимо.

Если система памяти разработана для поддержки множества независимых запросов (как это имеет место при работе с кэш-памятью, при реализации многопроцессорной и векторной обработки), эффективность системы будет в значительной степени зависеть от частоты поступления независимых запросов к разным банкам. Обращения по последовательным адресам, или в более общем случае обращения по адресам, отличающимся на нечетное число, хорошо обрабатываются традиционными схемами расслоенной памяти. Проблемы возникают, если разница в адресах последовательных обращений четная. Одно из решений, используемое в больших компьютерах, заключается в том, чтобы статистически уменьшить вероятность подобных обращений путем значительного увеличения количества банков памяти.

Использование специфических свойств динамических ЗУПВ. Как упоминалось раньше, обращение к ДЗУПВ состоит из двух этапов: обращения к строке и обращения к столбцу. При этом внутри микросхемы осуществляется буферизация битов строки, прежде чем происходит обращение к столбцу. Размер строки обычно является корнем квадратным от емкости кристалла памяти: 1024 бита для 1Мбит, 2048 бит для 4 Мбит и т. д. С целью увеличения производительности все современные микросхемы памяти обеспечивают возможность подачи сигналов синхронизации, которые позволяют выполнять последовательные обращения к

буферу без дополнительного времени обращения к строке. Имеются три способа подобной оптимизации:

1) блочный режим (nibble mode) – ДЗУВП может обеспечить выдачу четырех последовательных ячеек для каждого сигнала RAS;

2) страничный режим (page mode) – Буфер работает как статическое ЗУПВ; при изменении адреса столбца возможен доступ к произвольным битам в буфере до тех пор, пока не поступит новое обращение к строке или не наступит время регенерации;

3) режим статического столбца (static column) – Очень похож на страничный режим за исключением того, что не обязательно переключать строб адреса столбца каждый раз для изменения адреса столбца.

Начиная с микросхем ДЗУПВ емкостью 1 Мбит, большинство ДЗУПВ допускают любой из этих режимов, причем выбор режима осуществляется на стадии установки кристалла в корпус путем выбора соответствующих соединений. Преимуществом такой оптимизации является то, что она основана на внутренних схемах ДЗУПВ и незначительно увеличивает стоимость системы, позволяя практически учетверить пропускную способность памяти. Например, блочный режим был разработан для поддержки режимов, аналогичных расслоению памяти. Кристалл за один раз читает значения четырех бит и подает их наружу в течение четырех оптимизированных циклов. Если время пересылки по шине не превосходит время оптимизированного цикла, единственное усложнение для организации памяти с четырехкратным расслоением заключается в несколько усложненной схеме управления синхросигналами. Страничный режим и режим статического столбца также могут использоваться, обеспечивая даже большую степень расслоения при несколько более сложном управлении. Одной из тенденций в разработке ДЗУПВ является наличие в них буферов с тремя состояниями. Это предполагает, что для реализации традиционного расслоения с большим числом кристаллов памяти в системе должны быть предусмотрены буферные микросхемы для каждого банка памяти.

Новые поколения ДЗУВП разработаны с учетом возможности дальнейшей оптимизации интерфейса между ДЗУПВ и процессором. В качестве примера можно привести изделия компании RAMBUS. Эта компания берет стандартную начинку ДЗУПВ и обеспечивает новый интерфейс, делающий работу отдельной микросхемы более похожей на работу системы памяти, а не на работу отдельного ее компонента. RAMBUS отбросила сигналы RAS/CAS, заменив их шиной, которая допускает выполнение других обращений в интервале между посылкой адреса и приходом данных. Такого рода шины называются шинами с пакетным переключением (packet-switched bus) или шинами с расщепленными транзакциями (split-transaction bus). Такая шина позволяет работать кристаллу как отдельному банку па-

мяти. Кристалл может вернуть переменное количество данных на один запрос и даже самостоятельно выполняет регенерацию. RAMBUS предлагает байтовый интерфейс и сигнал синхронизации, так что микросхема может тесно синхронизироваться с тактовой частотой процессора. Большинство систем основной памяти используют методы, подобные страничному режиму ДЗУПВ, для уменьшения различий в производительности процессоров и микросхем памяти.

Виртуальная память и организация защиты памяти. Общепринятая в настоящее время концепция виртуальной памяти появилась достаточно давно. Она позволила решить целый ряд актуальных вопросов организации вычислений. Прежде всего, к числу таких вопросов относится обеспечение надежного функционирования мультипрограммных систем. В любой момент времени компьютер выполняет множество процессов или задач, каждая из которых располагает своим адресным пространством. Было бы слишком накладно отдавать всю физическую память какой-то одной задаче, тем более что многие задачи реально используют только небольшую часть своего адресного пространства. Поэтому необходим механизм разделения небольшой физической памяти между различными задачами. Виртуальная память является одним из способов реализации такой возможности. Она делит физическую память на блоки и распределяет их между различными задачами. При этом она предусматривает также некоторую схему защиты, которая ограничивает задачу теми блоками, которые ей принадлежат. Большинство типов виртуальной памяти сокращают также время начального запуска программы на процессоре, поскольку не весь программный код и данные требуются ей в физической памяти, чтобы начать выполнение.

Другой вопрос, тесно связанный с реализацией концепции виртуальной памяти, касается организации вычислений на компьютере задач очень большого объема. Если программа становилась слишком большой для физической памяти, часть ее необходимо было хранить во внешней памяти (на диске) и задача приспособить ее для решения на компьютере ложилась на программиста. Программисты делили программы на части и затем определяли те из них, которые можно было бы выполнять независимо, организуя оверлейные структуры, которые загружались в основную память и выгружались из нее под управлением программы пользователя. Программист должен был следить за тем, чтобы программа не обращалась вне отведенного ей пространства физической памяти. Виртуальная память освободила программистов от этого бремени. Она автоматически управляет двумя уровнями иерархии памяти: основной памятью и внешней (дисковой) памятью. Кроме того, виртуальная память упрощает также загрузку

программ, обеспечивая механизм автоматического перемещения программ, позволяющий выполнять одну и ту же программу в произвольном месте физической памяти. Системы виртуальной памяти можно разделить на два класса: системы с фиксированным размером блоков, называемых страницами, и системы с переменным размером блоков, называемых сегментами. Ниже рассмотрены оба типа организации виртуальной памяти.

Страничная организация памяти. В системах со страничной организацией основная и внешняя память (главным образом дисковое пространство) делятся на блоки или страницы фиксированной длины. Каждому пользователю предоставляется некоторая часть адресного пространства, которая может превышать основную память компьютера и которая ограничена только возможностями адресации, заложенными в системе команд. Эта часть адресного пространства называется виртуальной памятью пользователя. Каждое слово в виртуальной памяти пользователя определяется виртуальным адресом, состоящим из двух частей: старшие разряды адреса рассматриваются как номер страницы, а младшие – как номер слова (или байта) внутри страницы.

Управление различными уровнями памяти осуществляется программами ядра операционной системы, которые следят за распределением страниц и оптимизируют обмены между этими уровнями. При страничной организации памяти смежные виртуальные страницы не обязательно должны размещаться на смежных страницах основной физической памяти. Для указания соответствия между виртуальными страницами и страницами основной памяти операционная система должна сформировать таблицу страниц для каждой программы и разместить ее в основной памяти машины. При этом каждой странице программы, независимо от того находится ли она в основной памяти или нет, ставится в соответствие некоторый элемент таблицы страниц. Каждый элемент таблицы страниц содержит номер физической страницы основной памяти и специальный индикатор. Единичное состояние этого индикатора свидетельствует о наличии этой страницы в основной памяти. Нулевое состояние индикатора означает отсутствие страницы в оперативной памяти.

Для увеличения эффективности такого типа схем в процессорах используется специальная полностью ассоциативная кэш-память, которая также называется буфером преобразования адресов (TLB tranclation-lookaside buffer). Хотя наличие TLB не меняет принципа построения схемы страничной организации, с точки зрения защиты памяти, необходимо предусмотреть возможность очистки его при переключении с одной программы на другую.

Поиск в таблицах страниц, расположенных в основной памяти, и загрузка TLB может осуществляться либо программным способом, либо

специальными аппаратными средствами. В последнем случае для того, чтобы предотвратить возможность обращения пользовательской программы к таблицам страниц, с которыми она не связана, предусмотрены специальные меры. С этой целью в процессоре предусматривается дополнительный регистр защиты, содержащий описатель (дескриптор) таблицы страниц или базово-граничную пару. База определяет адрес начала таблицы страниц в основной памяти, а граница – длину таблицы страниц соответствующей программы. Загрузка этого регистра защиты разрешена только в привилегированном режиме. Для каждой программы операционная система хранит дескриптор таблицы страниц и устанавливает его в регистр защиты процессора перед запуском соответствующей программы.

Существуют некоторые особенности, присущие простым схемам со страничной организацией памяти. Наиболее важной из них является то, что все программы, которые должны непосредственно связываться друг с другом без вмешательства операционной системы, должны использовать общее пространство виртуальных адресов. Это относится и к самой операционной системе, которая, вообще говоря, должна работать в режиме динамического распределения памяти. Поэтому в некоторых системах пространство виртуальных адресов пользователя укорачивается на размер общих процедур, к которым программы пользователей желают иметь доступ. Общим процедурам должен быть отведен определенный объем пространства виртуальных адресов всех пользователей, чтобы они имели постоянное место в таблицах страниц всех пользователей. В этом случае для обеспечения целостности, секретности и взаимной изоляции выполняющихся программ должны быть предусмотрены различные режимы доступа к страницам, которые реализуются с помощью специальных индикаторов доступа в элементах таблиц страниц.

Следствием такого использования является значительный рост таблиц страниц каждого пользователя. Одно из решений проблемы сокращения длины таблиц основано на введении многоуровневой организации таблиц. Частным случаем многоуровневой организации таблиц является сегментация при страничной организации памяти. Необходимость увеличения адресного пространства пользователя объясняется желанием избежать необходимости перемещения частей программ и данных в пределах адресного пространства, которые обычно приводят к проблемам переименования и серьезным затруднениям в разделении общей информации между многими задачами.

Сегментация памяти. Другой подход к организации памяти опирается на тот факт, что программы обычно разделяются на отдельные области-сегменты. Каждый сегмент представляет собой отдельную логическую единицу информации, содержащую совокупность данных или про-

грамм и расположенную в адресном пространстве пользователя. Сегменты создаются пользователями, которые могут обращаться к ним по символлическому имени. В каждом сегменте устанавливается своя собственная нумерация слов, начиная с нуля. Обычно в подобных системах обмен информацией между пользователями строится на базе сегментов. Поэтому сегменты являются отдельными логическими единицами информации, которые необходимо защищать, и именно на этом уровне вводятся различные режимы доступа к сегментам. Можно выделить два основных типа сегментов: программные сегменты и сегменты данных (сегменты стека являются частным случаем сегментов данных). Поскольку общие программы должны обладать свойством повторной входимости, то из программных сегментов допускается только выборка команд и чтение констант. Запись в программные сегменты может рассматриваться как незаконная и запрещаться системой. Выборка команд из сегментов данных также может считаться незаконной, и любой сегмент данных может быть защищен от обращений по записи или по чтению.

Для реализации сегментации было предложено несколько схем, которые отличаются деталями реализации, но основаны на одних и тех же принципах. В системах с сегментацией памяти каждое слово в адресном пространстве пользователя определяется виртуальным адресом, состоящим из двух частей: старшие разряды адреса рассматриваются как номер сегмента, а младшие – как номер слова внутри сегмента. Наряду с сегментацией может также использоваться страничная организация памяти. В этом случае виртуальный адрес слова состоит из трех частей: старшие разряды адреса определяют номер сегмента, средние – номер страницы внутри сегмента, а младшие – номер слова внутри страницы.

Как и в случае страничной организации, необходимо обеспечить преобразование виртуального адреса в реальный физический адрес основной памяти. С этой целью для каждого пользователя операционная система должна сформировать таблицу сегментов. Каждый элемент таблицы сегментов содержит описатель (дескриптор) сегмента (поля базы, границы и индикаторов режима доступа). При отсутствии страничной организации поле базы определяет адрес начала сегмента в основной памяти, а граница – длину сегмента. При наличии страничной организации поле базы определяет адрес начала таблицы страниц данного сегмента, а граница – число страниц в сегменте. Поле индикаторов режима доступа представляет собой некоторую комбинацию признаков блокировки чтения, записи и выполнения.

Таблицы сегментов различных пользователей операционная система хранит в основной памяти. Для определения расположения таблицы сегментов выполняющейся программы используется специальный регистр

защиты, который загружается операционной системой перед началом ее выполнения. Этот регистр содержит дескриптор таблицы сегментов (базу и границу), причем база содержит адрес начала таблицы сегментов выполняющейся программы, а граница – длину этой таблицы сегментов. Разряды номера сегмента виртуального адреса используются в качестве индекса для поиска в таблице сегментов. Таким образом, наличие базово-граничных пар в дескрипторе таблицы сегментов и элементах таблицы сегментов предотвращает возможность обращения программы пользователя к таблицам сегментов и страниц, с которыми она не связана. Наличие в элементах таблицы сегментов индикаторов режима доступа позволяет осуществить необходимый режим доступа к сегменту со стороны данной программы. Для повышения эффективности схемы используется ассоциативная кэш-память.

Отметим, что в описанной схеме сегментации таблица сегментов с индикаторами доступа предоставляет всем программам, являющимся частями некоторой задачи, одинаковые возможности доступа, т. е. она определяет единственную область (домен) защиты. Однако для создания защищенных подсистем в рамках одной задачи для того, чтобы изменять возможности доступа, когда точка выполнения переходит через различные программы, управляющие ее решением, необходимо связать с каждой задачей множество доменов защиты. Реализация защищенных подсистем требует разработки некоторых специальных аппаратных средств. Рассмотрение таких систем, которые включают в себя кольцевые схемы защиты, а также различного рода мандатные схемы защиты, выходит за рамки данного курса.

МОДУЛЬ II. УНИВЕРСАЛЬНЫЕ МИКРОПРОЦЕССОРЫ

1. Определение понятия «архитектура». Архитектура системы команд. Классификация процессоров CISC и RISC.

Определение понятия «архитектура». Термин «архитектура системы» часто употребляется как в узком, так и в широком смысле этого слова. В узком смысле под архитектурой понимается архитектура набора команд. Архитектура набора команд служит границей между аппаратурой и программным обеспечением и представляет ту часть системы, которая видна программисту или разработчику компиляторов. Следует отметить, что это наиболее частое употребление этого термина. В широком смысле архитектура охватывает понятие организации системы, включающее такие высокоуровневые аспекты разработки компьютера как систему памяти, структуру системной шины, организацию ввода/вывода и т. п.

Применительно к вычислительным системам термин «архитектура» может быть определен как распределение функций, реализуемых системой, между ее уровнями, точнее как определение границ между этими уровнями. Таким образом, архитектура вычислительной системы предполагает многоуровневую организацию. Архитектура первого уровня определяет, какие функции по обработке данных выполняются системой в целом, а какие возлагаются на внешний мир (пользователей, операторов, администраторов баз данных и т. д.). Система взаимодействует с внешним миром через набор интерфейсов: языки (язык оператора, языки программирования, языки описания и манипулирования базой данных, язык управления заданиями) и системные программы (программы-утилиты, программы редактирования, сортировки, сохранения и восстановления информации).

Интерфейсы следующих уровней могут разграничивать определенные уровни внутри программного обеспечения. Например, уровень управления логическими ресурсами может включать реализацию таких функций, как управление базой данных, файлами, виртуальной памятью, сетевой телеобработкой. К уровню управления физическими ресурсами относятся функции управления внешней и оперативной памятью, управления процессами, выполняющимися в системе.

Следующий уровень отражает основную линию разграничения системы, а именно границу между системным программным обеспечением и аппаратурой. Эту идею можно развить и дальше и говорить о распределении функций между отдельными частями физической системы. Например, некоторый интерфейс определяет, какие функции реализуют центральные процессоры, а какие – процессоры ввода/вывода. Архитектура следующего

уровня определяет разграничение функций между процессорами ввода/вывода и контроллерами внешних устройств. В свою очередь можно разграничить функции, реализуемые контроллерами и самими устройствами ввода/вывода (терминалами, модемами, накопителями на магнитных дисках и лентах). Архитектура таких уровней часто называется архитектурой физического ввода/вывода.

Архитектура системы команд. Классификация процессоров (CISC и RISC). Как уже было отмечено, архитектура набора команд служит границей между аппаратурой и программным обеспечением и представляет ту часть системы, которая видна программисту или разработчику компиляторов.

Двумя основными архитектурами набора команд, используемыми компьютерной промышленностью на современном этапе развития вычислительной техники являются архитектуры CISC и RISC. Основоположником CISC-архитектуры можно считать компанию IBM с ее базовой архитектурой / 360, ядро которой используется с 1964 г. и дошло до наших дней, например, в таких современных мейнфреймах как IBM ES/9000.

Лидером в разработке микропроцессоров с полным набором команд (CISC – Complete Instruction Set Computer) считается компания Intel со своей серией x86 и Pentium. Эта архитектура является практическим стандартом для рынка микрокомпьютеров. Для CISC-процессоров характерно: сравнительно небольшое число регистров общего назначения; большое количество машинных команд, некоторые из которых нагружены семантически аналогично операторам высокоуровневых языков программирования и выполняются за много тактов; большое количество методов адресации; большое количество форматов команд различной разрядности; преобладание двухадресного формата команд; наличие команд обработки типа регистр-память.

Основой архитектуры современных рабочих станций и серверов является архитектура компьютера с сокращенным набором команд (RISC - Reduced Instruction Set Computer). Зачатки этой архитектуры уходят своими корнями к компьютерам CDC6600, разработчиков которых (Торнтона, Крэя и др.) можно считать первыми, кто воспользовался упрощением набора команд для построения быстрых вычислительных машин. Эту традицию упрощения архитектуры С. Крэй с успехом применил при создании широко известной серии суперкомпьютеров компании Cray Research. Однако окончательно понятие RISC в современном его понимании сформировалось на базе трех исследовательских проектов компьютеров: процессора 801 компании IBM, процессора RISC университета Беркли и процессора MIPS Стенфордского университета.

Разработка экспериментального проекта компании IBM началась еще в конце 70-х годов, но его результаты никогда не публиковались, и компьютер на его основе в промышленных масштабах не изготавливался. В 1980 г. Д. Паттерсон со своими коллегами из Беркли начали свой проект и изготовили две машины, которые получили названия RISC-I и RISC-II. Главными идеями этих машин было отделение медленной памяти от высокоскоростных регистров и использование регистровых окон. В 1981г. Дж. Хеннесси со своими коллегами опубликовал описание стенфордской машины MIPS, основным аспектом разработки которой была эффективная реализация конвейерной обработки посредством тщательного планирования компилятором загрузки этого конвейера.

Эти три машины имели много общего. Все они придерживались архитектуры, отделяющей команды обработки от команд работы с памятью, и делали упор на эффективную конвейерную обработку. Система команд разрабатывалась таким образом, чтобы выполнение любой команды занимало небольшое количество машинных тактов (предпочтительно один машинный такт). Сама логика выполнения команд с целью повышения производительности ориентировалась на аппаратную, а не на микропрограммную реализацию. Чтобы упростить логику декодирования команд использовались команды фиксированной длины и фиксированного формата.

Среди других особенностей RISC-архитектур следует отметить наличие достаточно большого регистрового файла (в типовых RISC-процессорах реализуются 32 или большее число регистров по сравнению с 8 – 16 регистрами в CISC-архитектурах), что позволяет большему объему данных храниться в регистрах на процессорном кристалле большее время и упрощает работу компилятора по распределению регистров под переменные. Для обработки, как правило, используются трехадресные команды, что помимо упрощения дешифрации дает возможность сохранять большее число переменных в регистрах без их последующей перезагрузки.

Таким образом, отличительными чертами RISC-архитектуры являются:

1) сокращенный набор относительно простых команд;
2) однородный набор регистров универсального назначения;
3) уменьшенный фиксированный формат команды;
4) большинство операций имеет характер регистр-регистр, а обращения к памяти происходят только для выполнения простых операций загрузки в регистры и занесения в память;
5) относительная простота процессора делает возможным размещение на кристалле большего числа регистров;
6) высокая степень конвейеризации вычислений, что позволяет выполнять большинство команд набора за один такт.

Ко времени завершения университетских проектов (1983 – 1984 гг.) обозначился также прорыв в технологии изготовления сверхбольших инте-

гральных схем. Простота архитектуры и ее эффективность, подтвержденная этими проектами, вызвали большой интерес в компьютерной индустрии, и с 1986 г. началась активная промышленная реализация архитектуры RISC. К настоящему времени эта архитектура прочно занимает лидирующие позиции на мировом компьютерном рынке рабочих станций и серверов.

Развитие архитектуры RISC в значительной степени определялось прогрессом в области создания оптимизирующих компиляторов. Именно современная техника компиляции позволяет эффективно использовать преимущества большего регистрового файла, конвейерной организации и большей скорости выполнения команд. Современные компиляторы используют также преимущества другой оптимизационной техники для повышения производительности, обычно применяемой в процессорах RISC: реализацию задержанных переходов и суперскалярной обработки, позволяющей в один и тот же момент времени выдавать на выполнение несколько команд.

Следует отметить, что в последних разработках компании Intel (начиная с Pentium P54C и процессорах следующих поколений), а также ее последователей-конкурентов (AMD и некоторых других) широко используются идеи, реализованные в RISC-микропроцессорах, так что многие различия между CISC и RISC стираются. Однако сложность архитектуры и системы команд x86 остается и является главным фактором, ограничивающим производительность процессоров на ее основе.

2. Методы адресации и типы данных. Типы команд. Команды управления потоком команд

Методы адресации. В машинах с регистрами общего назначения метод (или режим) адресации объектов, с которыми манипулирует команда, может задавать константу, регистр или ячейку памяти. Для обращения к ячейке памяти процессор, прежде всего, должен вычислить действительный или эффективный адрес памяти, который определяется заданным в команде методом адресации.

В таблице 3 представлены основные методы адресации операндов. Адресация непосредственных данных и литеральных констант обычно рассматривается как один из методов адресации памяти (хотя значения данных, к которым в этом случае производятся обращения, являются частью самой команды и обрабатываются в общем потоке команд). Адресация регистров, как правило, рассматривается отдельно. В данном разделе методы адресации, связанные со счетчиком команд (адресация относительно счетчика команд) рассматриваются отдельно. Этот вид адресации используется главным образом для определения программных адресов в командах передачи управления.

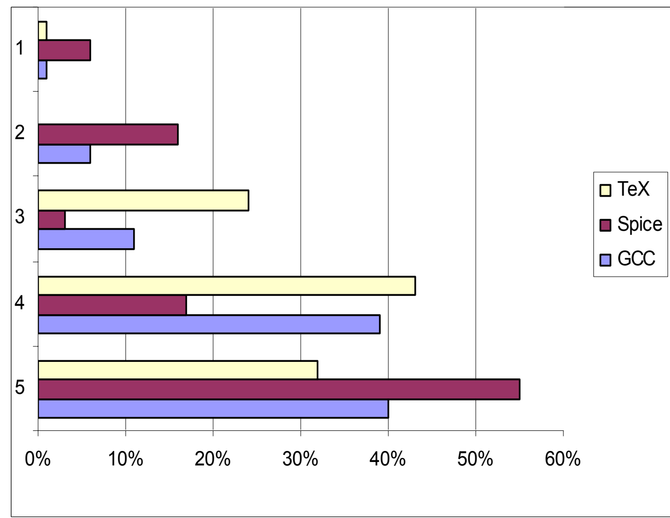
В таблице 3 на примере команды сложения (Add) приведены наиболее употребительные названия методов адресации, хотя при описании архитектуры в документации разные производители используют разные названия для этих методов. В этой таблице знак "(" используется для обозначения оператора присваивания, а буква M обозначает память (Memory)). Таким образом, M[R1] обозначает содержимое ячейки памяти, адрес которой определяется содержимым регистра R1.

Использование сложных методов адресации позволяет существенно сократить количество команд в программе, но при этом значительно увеличивается сложность аппаратуры. Возникает вопрос, а как часто эти методы адресации используются в реальных программах? На рисунке 34 представлены результаты измерений частоты использования различных методов адресации на примере трех популярных программ (компилятора с языка C и GCC, текстового редактора TeX и САПР Spice), выполненных на компьютере VAX.

Таблица 3

Методы адресации

Метод адресации	**Пример команды**	**Смысл команды Использование**
Регистровая	Add R4,R5	R4(R4+R5 Требуемое значение в регистре
Непосредственная или литеральная	Add R4,#3	R4(R4+3 Для задания констант
Базовая со смещением	Add R4,100(R1)	R4(R4+M[100+R1] Для обращения к локальным переменным
Косвенная регистровая	Add R4,(R1)	R4(R4+M[R1] Для обращения по указателю или вычисленному адресу
Индексная	Add R3,(R1+R2)	R3(R3+M[R1+R2] Иногда полезна при работе с массивами: R1 – база, R3 – индекс
Прямая илиабсолютная	Add R1,(1000)	R1(R1+M[1000] Иногда полезна для обращения к статическим данным
Косвенная	Add R1,@(R3)	R1(R1+M[M[R3]] Если R3 – адрес указателя p, то выбирается значение по этому указателю
Автоинкрементная	Add R1,(R2)+	R1(R1+M[R2] R2(R2+d Полезна для прохода в цикле по массиву с шагом: R2 – начало массива В каждом цикле R2 получает приращение d
Автодекрементная	Add R1,(R2)-	R2(R2-d R1(R1+M[R2] Аналогична предыдущей. Обе могут использоваться для реализации стека
Базовая индексная со смещением и масштабированием	Add R1, 100(R2)[R3]	R1(R1+M[100+R2+R3*d] Для индексации массивов



1. Косвенная (1%, 6%, 1%)

2. Базовая индексация со смещением и масштабированием (0%, 16%, 6%)

3. Регистровая (24%, 3%, 11%)

4. Непосредственная (43%, 17%, 39%)

5. Базовая со смещением (32%, 55%, 40%)

Рис. 34. Частота использования различных методов адресации на программах TeX, Spice, GCC

Из рисунка 34 видно, что непосредственная адресация и базовая со смещением доминируют.

При этом основной вопрос, который возникает для метода базовой адресации со смещением, связан с длиной (разрядностью) смещения. Выбор длины смещения, в конечном счете, определяет длину команды. Результаты измерений показали, что в подавляющем большинстве случаев длина смещения не превышает 16 разрядов.

Этот же вопрос важен и для непосредственной адресации. Непосредственная адресация используется при выполнении арифметических операций, операций сравнения, а также для загрузки констант в регистры. Результаты анализа статистики показывают, что в подавляющем числе случаев 16 разрядов оказывается вполне достаточно (хотя для вычисления адресов намного реже используются и более длинные константы).

Важным вопросом построения любой системы команд является оптимальное кодирование команд. Оно определяется количеством регистров и применяемых методов адресации, а также сложностью аппаратуры, необходимой для декодирования. Именно поэтому в современных RISC-архитектурах используются достаточно простые методы адресации, позволяющие резко упростить декодирование команд. Более сложные и редко встречающиеся в реальных программах методы адресации реализуются с помощью дополнительных команд, что приводит к увеличению размера программного кода. Однако такое увеличение длины программы окупается возможностью простого увеличения тактовой частоты RISC-процессоров. Этот процесс мы можем наблюдать сегодня, когда максимальные тактовые частоты практически всех RISC-процессоров (Alpha, R4400, Hyper SPARC и Power2) превышают тактовую частоту, достигнутую процессором Pentium.

Типы команд. Команды традиционного машинного уровня можно разделить на несколько типов, которые представлены ниже в табл. 4.

Таблица 4

Методы адресации

Тип операции	Примеры
Арифметические и логические	Целочисленные арифметические и логические операции: сложение, вычитание, логическое сложение, логическое умножение и т. д.
Пересылки данных	Операции загрузки/записи
Управление потоком команд	Безусловные и условные переходы, вызовы процедур и возвраты
Системные операции	Системные вызовы, команды управления виртуальной памятью и т. д.
Операции с плавающей точкой	Операции сложения, вычитания, умножения и деления над вещественными числами
Десятичные операции	Десятичное сложение, умножение, преобразование форматов и т. д.
Операции над строками	Пересылки, сравнения и поиск строк

Команды управления потоком команд. В английском языке для указания команд безусловного перехода, как правило, используется термин jump, а для команд условного перехода – термин branch, хотя разные поставщики не обязательно придерживаются этой терминологии. Например, компания Intel использует термин jump и для условных, и для безусловных переходов. Можно выделить четыре основных типа команд для управления потоком команд: условные переходы, безусловные переходы, вызовы процедур и возвраты из процедур.

Частота использования этих команд по статистике примерно следующая. В программах доминируют команды условного перехода. Среди указанных команд управления на разных программах частота их использования колеблется от 66 до 78%. Следующие по частоте использования – команды безусловного перехода (от 12 до 18%). Частота переходов на выполнение процедур и возврата из них составляет от 10 до 16%.

При этом примерно 90% команд безусловного перехода выполняются относительно счетчика команд. Для команд перехода адрес перехода должен быть всегда заранее известен. Это не относится к адресам возврата, которые не известны во время компиляции программы и должны определяться во время ее работы. Наиболее простой способ определения адреса перехода заключается в указании его положения относительно текущего значения счетчика команд (с помощью смещения в команде), и такие пере-

ходы называются переходами относительно счетчика команд. Преимуществом такого метода адресации является то, что адреса переходов, как правило, расположены недалеко от текущего адреса выполняемой команды и указание относительно текущего значения счетчика команд требует небольшого количества бит в смещении. Кроме того, использование адресации относительно счетчика команд позволяет программе выполняться в любом месте памяти, независимо от того, куда она была загружена. То есть этот метод адресации позволяет автоматически создавать перемещаемые программы.

Реализация возвратов и переходов по косвенному адресу, в которых адрес не известен во время компиляции программы, требует методов адресации, отличных от адресации относительно счетчика команд. В этом случае адрес перехода должен определяться динамически во время работы программы. Наиболее простой способ заключается в указании регистра для хранения адреса возврата, либо для перехода может разрешаться любой метод адресации для вычисления адреса перехода.

Одним из ключевых вопросов реализации команд перехода состоит в том, насколько далеко целевой адрес перехода находится от самой команды перехода? И на этот вопрос статистика использования команд дает ответ: в подавляющем большинстве случаев переход идет в пределах 3 – 7 команд относительно команды перехода, причем в 75% случаев выполняются переходы в направлении увеличения адреса, т. е. вперед по программе.

Поскольку большинство команд управления потоком команд составляют команды условного перехода, важным вопросом реализации архитектуры является определение условий перехода. Для этого используются три различных подхода. При первом из них в архитектуре процессора предусматривается специальный регистр, разряды которого соответствуют определенным кодам условий. Команды условного перехода проверяют эти условия в процессе своего выполнения. Преимуществом такого подхода является то, что иногда установка кода условия и переход по нему могут быть выполнены без дополнительных потерь времени, что, впрочем, бывает достаточно редко. Недостатками такого подхода является то, что, во-первых, появляются новые состояния машины, за которыми необходимо следить (сохранять при прерывании и восстанавливать при возврате из него). Во-вторых, и это очень важно для современных высокоскоростных конвейерных архитектур, коды условий ограничивают порядок выполнения команд в потоке, поскольку их основное назначение заключается в передаче кода условия команде условного перехода.

Второй метод заключается в простом использовании произвольного регистра (возможно одного выделенного) общего назначения. В этом случае выполняется проверка состояния этого регистра, в который предварительно помещается результат операции сравнения. Недостатком этого под-

хода является необходимость выделения в программе для анализа кодов условий специального регистра.

Третий метод предполагает объединение команды сравнения и перехода в одной команде. Недостатком такого подхода является то, что эта объединенная команда довольно сложна для реализации (в одной команде надо указать и тип условия, и константу для сравнения и адрес перехода). Поэтому в таких машинах часто используется компромиссный вариант, когда для некоторых кодов условий используются такие команды, например, для сравнения с нулем, а для более сложных условий используется регистр условий. Часто для анализа результатов команд сравнения для целочисленных операций и для операций с плавающей точкой используется разная техника, хотя это можно объяснить и тем, что в программах количество переходов по условиям выполнения операций с плавающей точкой значительно меньше общего количества переходов, определяемых результатами работы целочисленной арифметики.

Одним из наиболее заметных свойств большинства программ является преобладание в них сравнений на условие равно/неравно и сравнений с нулем. Поэтому в ряде архитектур такие команды выделяются в отдельный поднабор, особенно при использовании команд типа «сравнить и перейти».

Говорят, что переход выполняется, если истинным является условие, которое проверяет команда условного перехода. В этом случае выполняется переход на адрес, заданный командой перехода. Поэтому все команды безусловного перехода всегда выполняемые. По статистике оказывается, что переходы назад по программе в большинстве случаев используются для организации циклов, причем примерно 60% из них составляют выполняемые переходы. В общем случае поведение команд условного перехода зависит от конкретной прикладной программы, однако иногда сказывается и зависимость от компилятора. Такие зависимости от компилятора возникают вследствие изменений потока управления, выполняемого оптимизирующими компиляторами для ускорения выполнения циклов.

Вызовы процедур и возвраты предполагают передачу управления и возможно сохранение некоторого состояния. Как минимум, необходимо уметь где-то сохранять адрес возврата. Некоторые архитектуры предлагают аппаратные механизмы для сохранения состояния регистров, в других случаях предполагается вставка в программу команд самим компилятором. Имеются два основных вида соглашений относительно сохранения состояния регистров. Сохранение вызывающей (caller saving) программой означает, что вызывающая процедура должна сохранять свои регистры, которые она хочет использовать после возврата в нее. Сохранение вызванной процедурой предполагает, что вызванная процедура должна сохранить регистры, которые она собирается использовать.

3. Конвейеризация и параллелизм. Конвейерная организация обработки данных. Простейшая организация конвейера и оценка его производительности

Конвейеризация и параллелизм. Разработчики архитектуры компьютеров издавна прибегали к методам проектирования, известным под общим названием «совмещение операций», при котором аппаратура компьютера в любой момент времени выполняет одновременно более одной базовой операции. Этот общий метод включает два понятия: параллелизм и конвейеризацию. Хотя у них много общего и их зачастую трудно различать на практике, эти термины отражают два совершенно различных подхода. При параллелизме совмещение операций достигается путем воспроизведения в нескольких копиях аппаратной структуры. Высокая производительность достигается за счет одновременной работы всех элементов структур, осуществляющих решение различных частей задачи.

Конвейеризация (или конвейерная обработка) в общем случае основана на разделении подлежащей исполнению функции на более мелкие части, называемые ступенями, и выделении для каждой из них отдельного блока аппаратуры. Так обработку любой машинной команды можно разделить на несколько этапов (несколько ступеней), организовав передачу данных от одного этапа к следующему. При этом конвейерную обработку можно использовать для совмещения этапов выполнения разных команд. Производительность при этом возрастает благодаря тому, что одновременно на различных ступенях конвейера выполняются несколько команд. Конвейерная обработка такого рода широко применяется во всех современных быстродействующих процессорах.

Простейшая организация конвейера и оценка его производительности. Для иллюстрации основных принципов построения процессоров мы будем использовать простейшую архитектуру, содержащую 32 целочисленных регистра общего назначения (R0,...,R31), 32 регистра плавающей точки (F0,...,F31) и счетчик команд PC. Будем считать, что набор команд нашего процессора включает типичные арифметические и логические операции, операции с плавающей точкой, операции пересылки данных, операции управления потоком команд и системные операции. В арифметических командах используется трехадресный формат, типичный для RISC-процессоров, а для обращения к памяти используются операции загрузки и записи содержимого регистров в память.

Выполнение типичной команды можно разделить на следующие этапы:

1) выборка команды – IF (по адресу, заданному счетчиком команд, из памяти извлекается команда);

2) декодирование команды/выборка операндов из регистров – ID;

3) выполнение операции/вычисление эффективного адреса памяти – EX;

4) обращение к памяти – MEM;

5) запоминание результата – WB.

Работу конвейера можно условно представить в виде сдвинутых во времени схем процессора (рис. 35). Этот рисунок хорошо отражает совмещение во времени выполнения различных этапов команд. Однако чаще для представления работы конвейера используются временные диаграммы (табл. 5), на которых обычно изображаются выполняемые команды, номера тактов и этапы выполнения команд.

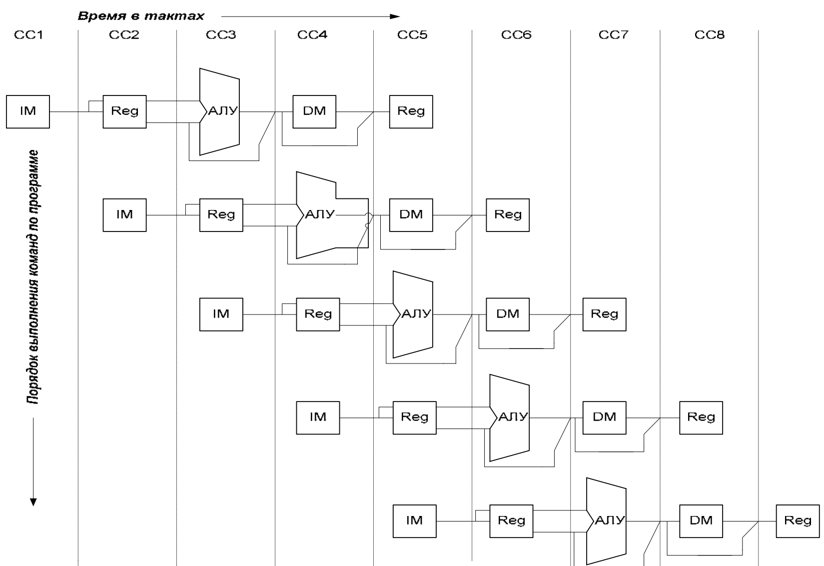


Рис. 35. Представление о работе конвейера

Таблица 5

Диаграмма работы простейшего конвейера

Номер команды	Номер такта								
	1	2	3	4	5	6	7	8	9
Команда i	IF	ID	EX	MEM	WB				
Команда i+1		IF	ID	EX	MEM	WB			
Команда i+2			IF	ID	EX	MEM	WB		
Команда i+3				IF	ID	EX	MEM	WB	
Команда i+4					IF	ID	EX	MEM	WB

Конвейеризация увеличивает пропускную способность процессора (количество команд, завершающихся в единицу времени), но она не сокращает время выполнения отдельной команды. В действительности, она даже несколько увеличивает время выполнения каждой команды из-за накладных расходов, связанных с управлением регистровыми станциями. Однако увеличение пропускной способности означает, что программа будет выполняться быстрее по сравнению с простой неконвейерной схемой.

Тот факт, что время выполнения каждой команды в конвейере не уменьшается, накладывает некоторые ограничения на практическую длину

конвейера. Кроме ограничений, связанных с задержкой конвейера, имеются также ограничения, возникающие в результате несбалансированности задержки на каждой его ступени и из-за накладных расходов на конвейеризацию. Частота синхронизации не может быть выше, а следовательно, такт синхронизации не может быть меньше, чем время, необходимое для работы наиболее медленной ступени конвейера. Накладные расходы на организацию конвейера возникают из-за задержки сигналов в конвейерных регистрах (защелках) и из-за перекосов сигналов синхронизации. Конвейерные регистры к длительности такта добавляют время установки и задержку распространения сигналов.

В качестве примера рассмотрим неконвейерную машину с пятью этапами выполнения операций, которые имеют длительность 50, 50, 60, 50 и 50нс. соответственно (рис. 36). Пусть накладные расходы на организацию конвейерной обработки составляют 5 нс. Тогда среднее время выполнения команды в неконвейерной машине будет равно 260 нс. Если же используется конвейерная организация, длительность такта будет равна длительности самого медленного этапа обработки плюс накладные расходы, т. е. 65нс. Это время соответствует среднему времени выполнения команды в конвейере. Таким образом, ускорение, полученное в результате конвейеризации, будет равно:

– Среднее время выполнения команды в неконвейерном режиме – 260;
– Среднее время выполнения команды в конвейерном режиме – 64.

Конвейеризация эффективна только тогда, когда загрузка конвейера близка к полной, а скорость подачи новых команд и операндов соответствует максимальной производительности конвейера. Если произойдет задержка, то параллельно будет выполняться меньше операций и суммарная производительность снизится. Такие задержки могут возникать в результате возникновения конфликтных ситуаций. В следующих разделах будут рассмотрены различные типы конфликтов, возникающие при выполнении команд в конвейере, и способы их разрешения.

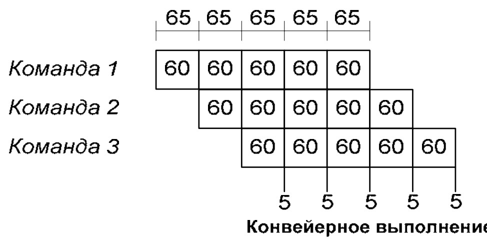


Рис. 36. Эффект конвейеризации при выполнении 3-х команд – четырехкратное ускорение

При реализации конвейерной обработки возникают ситуации, которые препятствуют выполнению очередной команды из потока команд в предназначенном для нее такте. Такие ситуации называются конфликтами. Конфликты снижают реальную производительность конвейера, которая могла бы быть достигнута в идеальном случае. Существуют три класса конфликтов:

1) структурные конфликты, которые возникают из-за конфликтов по ресурсам, когда аппаратные средства не могут поддерживать все возможные комбинации команд в режиме одновременного выполнения с совмещением;

2) конфликты по данным, возникающие в случае, когда выполнение одной команды зависит от результата выполнения предыдущей команды;

3) конфликты по управлению, которые возникают при конвейеризации команд переходов и других команд, которые изменяют значение счетчика команд.

Конфликты в конвейере приводят к необходимости приостановки выполнения команд (pipeline stall). Обычно в простейших конвейерах, если приостанавливается какая-либо команда, то все следующие за ней команды также приостанавливаются. Команды, предшествующие приостановленной, могут продолжать выполняться, но во время приостановки не выбирается ни одна новая команда.

4. Структурные конфликты и способы их минимизации. Конфликты по данным, остановы конвейера и реализация механизма обходов. Сокращение потерь на выполнение команд перехода и минимизация конфликтов по управлению

Структурные конфликты и способы их минимизации. Совмещенный режим выполнения команд в общем случае требует конвейеризации функциональных устройств и дублирования ресурсов для разрешения всех возможных комбинаций команд в конвейере. Если какая-нибудь комбинация команд не может быть принята из-за конфликта по ресурсам, то говорят, что в машине имеется структурный конфликт. Наиболее типичным примером машин, в которых возможно появление структурных конфликтов, являются машины с не полностью конвейерными функциональными устройствами. Время работы такого устройства может составлять несколько тактов синхронизации конвейера. В этом случае последовательные команды, которые используют данное функциональное устройство, не могут поступать в него в каждом такте. Другая возможность появления структурных конфликтов связана с недостаточным дублированием некоторых ресурсов, что препятствует выполнению произвольной последовательности команд в конвейере без его приостановки. Например, машина может иметь только один порт записи в регистровый файл, но при определенных обстоятельствах конвейеру может потребоваться выполнить две записи в регистровый файл в одном такте. Это также приведет к структурному конфликту. Когда последовательность команд наталкивается

на такой конфликт, конвейер приостанавливает выполнение одной из команд до тех пор, пока не станет доступным требуемое устройство.

Структурные конфликты возникают, например, и в машинах, в которых имеется единственный конвейер памяти для команд и данных (рис. 37). В этом случае, когда одна команда содержит обращение к памяти за данными, оно будет конфликтовать с выборкой более поздней команды из памяти. Чтобы разрешить эту ситуацию, можно просто приостановить конвейер на один такт, когда происходит обращение к памяти за данными. Подобная приостановка часто называются «конвейерным пузырем» (pipeline bubble) или просто пузырем, поскольку пузырь проходит по конвейеру, занимая место, но, не выполняя никакой полезной работы.

При всех прочих обстоятельствах, машина без структурных конфликтов будет всегда иметь более низкий CPI (среднее число тактов на выдачу команды). Возникает вопрос: почему разработчики допускают наличие структурных конфликтов? Для этого имеются две причины: снижение стоимости и уменьшение задержки устройства. Конвейеризация всех функциональных устройств может оказаться слишком дорогой. Машины, допускающие два обращения к памяти в одном такте, должны иметь удвоенную пропускную способность памяти, например, путем организации раздельных кэшей для команд и данных. Аналогично, полностью конвейерное устройство деления с плавающей точкой требует огромного количества вентилей. Если структурные конфликты не будут возникать слишком часто, то может быть и не стоит платить за то, чтобы их обойти. Как правило, можно разработать неконвейерное, или не полностью конвейерное устройство, имеющее меньшую общую задержку, чем полностью конвейерное. Например, разработчики устройств с плавающей точкой компьютеров CDC7600 и MIPS R2010 предпочли иметь меньшую задержку выполнения операций вместо полной их конвейеризации.

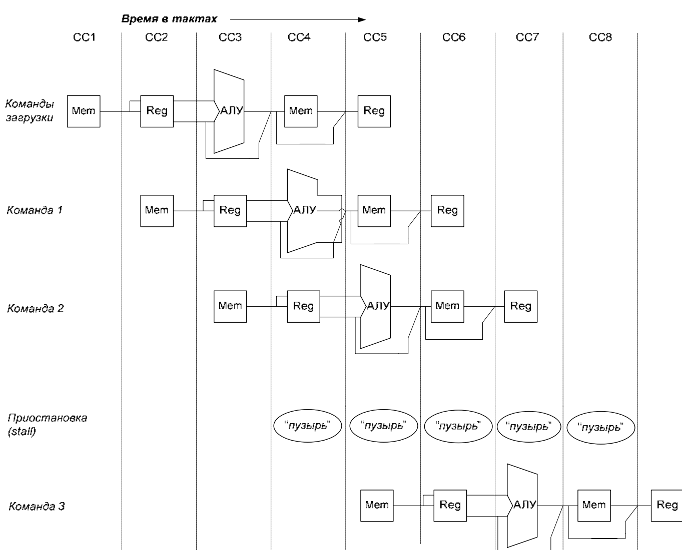


Рис. 37. Пример структурного конфликта при реализации памяти с одним портом

Временная диаграмма этой приостановки представлена ниже в табл. 6.

Таблица 6

Диаграмма работы простейшего конвейера при рассмотренном структурном конфликте

Команда	Номер такта									
	1	2	3	4	5	6	7	8	9	10
Команда загрузки	IF	ID	EX	MEM	WB					
Команда 1		IF	ID	EX	MEM	WB				
Команда 2			IF	ID	EX	MEM	WB			
Команда 3				stall	IF	ID	EX	MEM	WB	
Команда 4						IF	ID	EX	MEM	WB
Команда 5							IF	ID	EX	MEM
Команда 6								IF	ID	EX

Конфликты по данным, остановы конвейера и реализация механизма обходов. Одним из факторов, который оказывает существенное влияние на производительность конвейерных систем, являются межкомандные логические зависимости. Такие зависимости в большой степени ограничивают потенциальный параллелизм смежных операций, обеспечиваемый соответствующими аппаратными средствами обработки. Степень влияния этих зависимостей определяется как архитектурой процессора (в основном, структурой управления конвейером команд и параметрами функциональных устройств), так и характеристиками программ.

Конфликты по данным возникают в том случае, когда применение конвейерной обработки может изменить порядок обращений за операндами так, что этот порядок будет отличаться от порядка, который наблюдается при последовательном выполнении команд на неконвейерной машине. Рассмотрим конвейерное выполнение последовательности команд в табл. 7.

Таблица 7

Совмещение чтения и записи регистров в одном такте

ADD	R1,R2,R3	IF	ID	EX	MEM	WB				
			R			W				
SUB	R4,R1,R5		IF	ID	EX	MEM	WB			
				R			W			
AND	R6,R1,R7			IF	ID	EX	MEM	WB		
					R			W		
OR	R8,R1,R9				IF	ID	EX	MEM	WB	
						R			W	
XOR	R10,R1,R11					IF	ID	EX	MEM	WB
							R			W

В этом примере все команды, следующие за командой ADD, используют результат ее выполнения. Команда ADD записывает результат в ре-

гистр R1, а команда SUB читает это значение. Если не предпринять никаких мер для того, чтобы предотвратить этот конфликт, команда SUB прочитает неправильное значение и попытается его использовать. На самом деле значение, используемое командой SUB, является даже неопределенным: хотя логично предположить, что SUB всегда будет использовать значение R1, которое было присвоено какой-либо командой, предшествовавшей ADD, это не всегда так. Если произойдет прерывание между командами ADD и SUB, то команда ADD завершится, и значение R1 в этой точке будет соответствовать результату ADD. Такое непрогнозируемое поведение очевидно неприемлемо.

Проблема, поставленная в этом примере, может быть разрешена с помощью достаточно простой аппаратной техники, которая называется пересылкой или продвижением данных (data forwarding), обходом (data bypassing), иногда закороткой (short-circuiting). Эта аппаратура работает следующим образом. Результат операции АЛУ с его выходного регистра всегда снова подается назад на входы АЛУ. Если аппаратура обнаруживает, что предыдущая операция АЛУ записывает результат в регистр, соответствующий источнику операнда для следующей операции АЛУ, то логические схемы управления выбирают в качестве входа для АЛУ результат, поступающий по цепи «обхода», а не значение, прочитанное из регистрового файла (рис. 38).

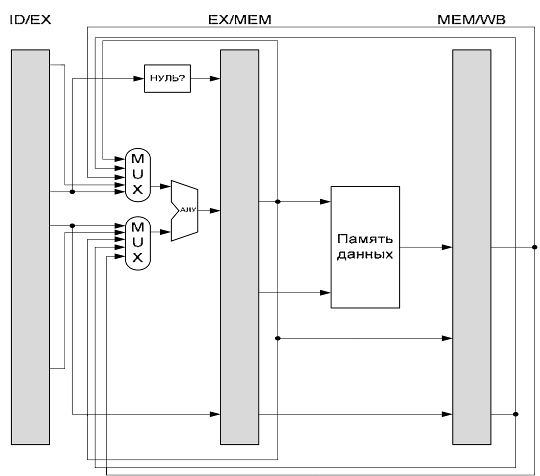


Рис. 38. АЛУ с цепями обхода и ускоренной пересылки

Эта техника «обходов» может быть обобщена для того, чтобы включить передачу результата прямо в то функциональное устройство, которое в нем нуждается: результат с выхода одного устройства «пересылается» на вход другого, а не с выхода некоторого устройства только на его вход.

Классификация конфликтов по данным. Конфликт возникает везде, где имеет место зависимость между командами, и они расположены по отношению друг к другу достаточно близко так, что совмещение

операций, происходящее при конвейеризации, может привести к изменению порядка обращения к операндам. В нашем примере был проиллюстрирован конфликт, происходящий с регистровыми операндами, но для пары команд возможно появление зависимостей при записи или чтении одной и той же ячейки памяти. Однако, если все обращения к памяти выполняются в строгом порядке, то появление такого типа конфликтов предотвращается.

Известны три возможных конфликта по данным в зависимости от порядка операций чтения и записи. Рассмотрим две команды i и j, при этом i предшествует j. Возможны следующие конфликты:

1. RAW (чтение после записи) – j пытается прочитать операнд-источник данных прежде, чем i туда запишет. Таким образом, j может некорректно получить старое значение. Это наиболее общий тип конфликтов, способ их преодоления с помощью механизма «обходов» рассмотрен ранее.

2. WAR (запись после чтения) – j пытается записать результат в приемник прежде, чем он считывается оттуда командой i, так что i может некорректно получить новое значение. Этот тип конфликтов как правило не возникает в системах с централизованным управлением потоком команд, обеспечивающих выполнение команд в порядке их поступления, так как последующая запись всегда выполняется позже, чем предшествующее считывание. Особенно часто конфликты такого рода могут возникать в системах, допускающих выполнение команд не в порядке их расположения в программном коде.

3. WAW (запись после записи) – j пытается записать операнд прежде, чем будет записан результат команды i, т.е. записи заканчиваются в неверном порядке, оставляя в приемнике значение, записанное командой i, а не j. Этот тип конфликтов присутствует только в конвейерах, которые выполняют запись со многих ступеней (или позволяют команде выполняться даже в случае, когда предыдущая приостановлена).

Конфликты по данным, приводящие к приостановке конвейера. К сожалению, не все потенциальные конфликты по данным могут обрабатываться с помощью механизма «обходов». Рассмотрим последовательность команд, приведенную в табл. 8.

Таблица 8

Последовательность команд с приостановкой конвейера

Команда	IF	ID	EX	MEM	WB					
LW R1,32(R6)		IF	ID	EX	MEM	WB				
ADD R4,R1,R7			IF	ID	stall	EX	MEM	WB		
SUB R5,R1,R8				IF	stall	ID	EX	MEM	WB	
AND R6,R1,R7					stall	IF	ID	EX	MEM	WB

Этот случай отличается от последовательности подряд идущих команд АЛУ. Команда загрузки (LW) регистра R1 из памяти имеет задержку, которая не может быть устранена обычной «пересылкой». Вместо этого нам нужна дополнительная аппаратура, называемая аппаратурой внутренних блокировок конвейера (pipeline interlook), чтобы обеспечить корректное выполнение примера. Вообще такого рода аппаратура обнаруживает конфликты и приостанавливает конвейер до тех пор, пока существует конфликт. В этом случае эта аппаратура приостанавливает конвейер начиная с команды, которая хочет использовать данные в то время, когда предыдущая команда, результат которой является операндом для нашей, вырабатывает этот результат. Эта аппаратура вызывает приостановку конвейера или появление «пузыря» точно также, как и в случае структурных конфликтов.

Методика планирования компилятора для устранения конфликтов по данным. Многие типы приостановок конвейера могут происходить достаточно часто. Например, для оператора A = B + C компилятор скорее всего сгенерирует следующую последовательность команд (табл. 9).

Таблица 9

Конвейерное выполнение оператора A = B + C

LW R1,B	IF	ID	EX	MEM	WB				
LW R2,C		IF	ID	EX	MEM	WB			
ADD R3,R1,R2			IF	ID	stall	EX	MEM	WB	
SW A,R3				IF	stall	ID	EX	MEM	WB

Очевидно, выполнение команды ADD должно быть приостановлено до тех пор, пока не станет доступным поступающий из памяти операнд C. Дополнительной задержки выполнения команды SW не произойдет в случае применения цепей обхода для пересылки результата операции АЛУ непосредственно в регистр данных памяти для последующей записи.

Для данного простого примера компилятор никак не может улучшить ситуацию, однако в ряде более общих случаев он может реорганизовать последовательность команд так, чтобы избежать приостановок конвейера. Эта техника, называемая планированием загрузки конвейера (pipeline scheduling) или планированием потока команд (instruction scheduling), использовалась начиная с 60-х годов и стала особой областью интереса в 80-х годах, когда конвейерные машины стали более распространенными.

Пусть, например, имеется последовательность операторов:

a = b + c; d = e - f;

Предполагается, что задержка загрузки из памяти составляет один такт. Сгенерированный код и код не вызывающий остановок конвейера приведены ниже в таблице 10.

Таблица 10

Пример устранения конфликтов компилятором

Неоптимизированная последовательность команд	Оптимизированная последовательность команд
LW Rb,b	LW Rb,b
LW Rc,c	LW Rc,c
ADD Ra,Rb,Rc	LW Re,e
SW a,Ra	ADD Ra,Rb,Rc
LW Re,e	LW Rf,f
LW Rf,f	SW a,Ra
SUB Rd,Re,Rf	SUB Rd,Re,Rf
SW d,Rd	SW d,Rd

В результате устранены обе блокировки (командой LW Rc,c команды ADD Ra,Rb,Rc и командой LW Rf,f команды SUB Rd,Re,Rf). Имеется зависимость между операцией АЛУ и операцией записи в память, но структура конвейера допускает пересылку результата с помощью цепей «обхода». Заметим, что использование разных регистров для первого и второго операторов было достаточно важным для реализации такого правильного планирования. В частности, если переменная e была бы загружена в тот же самый регистр, что b или c, такое планирование не было бы корректным. В общем случае планирование конвейера может требовать увеличенного количества регистров. Такое увеличение может оказаться особенно существенным для машин, которые могут выдавать на выполнение несколько команд в одном такте.

Многие современные компиляторы используют технику планирования команд для улучшения производительности конвейера. В простейшем алгоритме компилятор просто планирует распределение команд в одном и том же базовом блоке. Базовый блок представляет собой линейный участок последовательности программного кода, в котором отсутствуют команды перехода, за исключением начала и конца участка (переходы внутрь этого участка тоже должны отсутствовать). Планирование такой последовательности команд осуществляется достаточно просто, поскольку компилятор знает, что каждая команда в блоке будет выполняться, если выполняется первая из них, и можно просто построить граф зависимостей этих команд и упорядочить их так, чтобы минимизировать приостановки конвейера. Для простых конвейеров стратегия планирования на основе базовых блоков вполне удовлетворительна. Однако когда конвейеризация становится более интенсивной и действительные задержки конвейера растут, требуются более сложные алгоритмы планирования.

К счастью, существуют аппаратные методы, позволяющие изменить порядок выполнения команд программы так, чтобы минимизировать приостановки конвейера. Эти методы получили общее название методов ди-

намической оптимизации (в англоязычной литературе в последнее время часто применяются также термины «out-of-order execution» – неупорядоченное выполнение и «out-of-order issue» – неупорядоченная выдача). Основными средствами динамической оптимизации являются:

1. Размещение схемы обнаружения конфликтов в возможно более низкой точке конвейера команд так, чтобы позволить команде продвигаться по конвейеру до тех пор, пока ей реально не потребуется операнд, являющийся также результатом логически более ранней, но еще не завершившейся команды. Альтернативным подходом является централизованное обнаружение конфликтов на одной из ранних ступеней конвейера.

2. Буферизация команд, ожидающих разрешения конфликта, и выдача последующих, логически не связанных команд, в "обход" буфера. В этом случае команды могут выдаваться на выполнение не в том порядке, в котором они расположены в программе, однако аппаратура обнаружения и устранения конфликтов между логически связанными командами обеспечивает получение результатов в соответствии с заданной программой.

3. Соответствующая организация коммутирующих магистралей, обеспечивающая засылку результата операции непосредственно в буфер, хранящий логически зависимую команду, задержанную из-за конфликта, или непосредственно на вход функционального устройства до того, как этот результат будет записан в регистровый файл или в память (short-circuiting, data forwarding, data bypassing – методы, которые были рассмотрены ранее).

Еще одним аппаратным методом минимизации конфликтов по данным является метод переименования регистров (register renaming). Он получил свое название от широко применяющегося в компиляторах метода переименования – метода размещения данных, способствующего сокращению числа зависимостей и тем самым увеличению производительности при отображении необходимых исходной программе объектов (например, переменных) на аппаратные ресурсы (например, ячейки памяти и регистры).

При аппаратной реализации метода переименования регистров выделяются логические регистры, обращение к которым выполняется с помощью соответствующих полей команды, и физические регистры, которые размещаются в аппаратном регистровом файле процессора. Номера логических регистров динамически отображаются на номера физических регистров посредством таблиц отображения, которые обновляются после декодирования каждой команды. Каждый новый результат записывается в новый физический регистр. Однако предыдущее значение каждого логического регистра сохраняется и может быть восстановлено в случае, если выполнение команды должно быть прервано из-за возникновения исключительной ситуации или неправильного предсказания направления условного перехода.

В процессе выполнения программы генерируется множество временных регистровых результатов. Эти временные значения записываются в регистровые файлы вместе с постоянными значениями. Временное значение становится новым постоянным значением, когда завершается выполнение команды (фиксируется ее результат). В свою очередь, завершение выполнения команды происходит, когда все предыдущие команды успешно завершились в заданном программой порядке.

Программист имеет дело только с логическими регистрами. Реализация физических регистров от него скрыта. Как уже отмечалось, номера логических регистров ставятся в соответствие номерам физических регистров. Отображение реализуется с помощью таблиц отображения, которые обновляются после декодирования каждой команды. Каждый новый результат записывается в физический регистр. Однако до тех пор, пока не завершится выполнение соответствующей команды, значение в этом физическом регистре рассматривается как временное.

Метод переименования регистров упрощает контроль зависимостей по данным. В машине, которая может выполнять команды не в порядке их расположения в программе, номера логических регистров могут стать двусмысленными, поскольку один и тот же регистр может быть назначен последовательно для хранения различных значений. Но поскольку номера физических регистров уникально идентифицируют каждый результат, все неоднозначности устраняются.

Сокращение потерь на выполнение команд перехода и минимизация конфликтов по управлению. Конфликты по управлению могут вызывать даже большие потери производительности конвейера, чем конфликты по данным. Когда выполняется команда условного перехода, она может либо изменить, либо не изменить значение счетчика команд. Если команда условного перехода заменяет счетчик команд значением адреса, вычисленного в команде, то переход называется выполняемым; в противном случае, он называется невыполняемым.

Простейший метод работы с условными переходами заключается в приостановке конвейера, как только обнаружена команда условного перехода до тех пор, пока она не достигнет ступени конвейера, которая вычисляет новое значение счетчика команд (табл. 11). Такие приостановки конвейера из-за конфликтов по управлению должны реализовываться иначе, чем приостановки из-за конфликтов по данным, поскольку выборка команды, следующей за командой условного перехода, должна быть выполнена как можно быстрее, как только мы узнаем окончательное направление команды условного перехода.

Таблица 11

Приостановка конвейера при выполнении команды условного перехода

Команды перехода	IF	ID	EX	MEM	WB					
Следующая команда		IF	stall	stall	IF	ID	EX	MEM	WB	
Следующая команда +1			stall	stall	stall	IF	ID	EX	MEM	WB
Следующая команда +2				stall	stall	stall	IF	ID	EX	MEM
Следующая команда +3					stall	stall	stall	IF	ID	EX
Следующая команда +4						stall	stall	stall	IF	ID
Следующая команда +5							stall	stall	stall	IF

Например, если конвейер будет приостановлен на три такта на каждой команде условного перехода, то это может существенно отразиться на производительности машины. При частоте команд условного перехода в программах, равной 30% и идеальном CPI, равным 1, машина с приостановками условных переходов достигает примерно только половины ускорения, получаемого за счет конвейерной организации. Таким образом, снижение потерь от условных переходов становится критическим вопросом. Число тактов, теряемых при приостановках из-за условных переходов, может быть уменьшено двумя способами:

1. Обнаружением является ли условный переход выполняемым или невыполняемым на более ранних ступенях конвейера.

2. Более ранним вычислением значения счетчика команд для выполняемого перехода (т. е. вычислением целевого адреса перехода).

В некоторых машинах конфликты из-за условных переходов являются даже еще более дорогостоящими по количеству тактов, чем в нашем примере, поскольку время на оценку условия перехода и вычисление адреса перехода может быть даже большим. Например, машина с раздельными ступенями декодирования и выборки команд возможно будет иметь задержку условного перехода (длительность конфликта по управлению), которая по крайней мере на один такт длиннее. Многие компьютеры VAX имеют задержки условных переходов в четыре и более тактов, а большие машины с глубокими конвейерами имеют потери по условным переходам, равные шести или семи тактам. В общем случае, чем глубина конвейера больше, тем больше потери на командах условного перехода, исчисляемые в тактах. Конечно эффект снижения относительной производительности при этом зависит от общего CPI машины. Машины с высоким CPI могут иметь условные переходы большей длительности, поскольку процент производительности машины, которая будет потеряна из-за условных переходов, меньше.

Снижение потерь на выполнение команд условного перехода. Имеется несколько методов сокращения приостановок конвейера, возникающих из-за задержек выполнения условных переходов. Проанализируем четыре простые схемы, используемые во время компиляции. В этих схемах прогнозирование направления перехода выполняется статически, т. е. прогнозируемое направление перехода фиксируется для каждой команды

условного перехода на все время выполнения программы. Далее рассмотрим вопрос о правильности предсказания направления перехода компиляторами, поскольку все эти схемы основаны на такой технологии.

Метод выжидания

Простейшая схема обработки команд условного перехода заключается в замораживании или подавлении операций в конвейере, путем блокировки выполнения любой команды, следующей за командой условного перехода, до тех пор, пока не станет известным направление перехода. В таблице 11 приведен именно такой подход. Привлекательность такого решения заключается в его простоте.

Метод возврата

Вместо метода выжидания можно использовать более сложную схему, суть которой состоит в том, чтобы прогнозировать условный переход как невыполняемый. При этом аппаратура должна просто продолжать выполнение программы, как если бы условный переход вовсе не выполнялся. В этом случае необходимо позаботиться о том, чтобы не изменить состояние машины до тех пор, пока направление перехода не станет окончательно известным. В некоторых машинах эта схема с невыполняемыми по прогнозу условными переходами реализована путем продолжения выборки команд, как если бы условный переход был обычной командой. Поведение конвейера выглядит так, как будто ничего необычного не происходит. Однако, если условный переход на самом деле выполняется, то необходимо просто очистить конвейер от команд, выбранных вслед за командой условного перехода и заново повторить выборку команд (табл. 12).

Таблица 12

Диаграмма работы модернизированного конвейера

Невыполняемый условный переход	IF	ID	EX	MEM	WB				
Команда i+1		IF	ID	EX	MEM	WB			
Команда i+2			IF	ID	EX	MEM	WB		
Команда i+3				IF	ID	EX	MEM	WB	
Команда i+4					IF	ID	EX	MEM	WB
Выполняемый условный переход	IF	ID	EX	MEM	WB				
Команда i+1		IF	ID	EX	MEM	WB			
Команда i+2			stall	IF	ID	EX	MEM	WB	
Команда i+3				stall	IF	ID	EX	MEM	WB
Команда i+4					stall	IF	ID	EX	MEM

Альтернативная схема прогнозирует переход как выполняемый. Как только команда условного перехода декодирована и вычислен целевой адрес перехода, мы предполагаем, что переход выполняемый, и осуществляем выборку команд и их выполнение, начиная с целевого адреса. Если мы не знаем целевой адрес перехода раньше, чем узнаем окончательное направление перехода, у этого подхода нет никаких преимуществ. Если бы условие перехода зависело от непосредственно предшествующей команды,

то произошла бы приостановка конвейера из-за конфликта по данным для регистра, который является условием перехода, и мы бы узнали сначала целевой адрес. В таких случаях прогнозировать переход как выполняемый было бы выгодно. Дополнительно в некоторых машинах (особенно в машинах с устанавливаемыми по умолчанию кодами условий или более мощным (а потому и более медленным) набором условий перехода) целевой адрес перехода известен раньше окончательного направления перехода, и схема прогноза перехода как выполняемого имеет смысл.

Задержанные переходы

Четвертая схема, которая используется в некоторых машинах, называется «задержанным переходом». В задержанном переходе такт выполнения с задержкой перехода длиною n есть:

```
команда условного перехода ─────────────┐
   следующая команда 1                   │
   следующая команда 2                   │
   …                                     │
   следующая команда n                   │
   целевой адрес при выполняемом переходе◄┘
```

Команды 1 – n находятся в слотах (временных интервалах) задержанного перехода. Задача программного обеспечения заключается в том, чтобы сделать команды, следующие за командой перехода, действительными и полезными. Аппаратура гарантирует реальное выполнение этих команд перед выполнением собственно перехода. Здесь используются несколько приемов оптимизации.

На рисунке 39 показаны три случая, при которых может планироваться задержанный переход. В верхней части рисунка для каждого случая показана исходная последовательность команд, а в нижней части – последовательность команд, полученная в результате планирования. В случае (a) слот задержки заполняется независимой командой, находящейся перед командой условного перехода. Это наилучший выбор. Стратегии (b) и (c) используются, если применение стратегии (a) невозможно.

В последовательностях команд для случаев (b) и (c) использование содержимого регистра R1 в качестве условия перехода препятствует перемещению команды ADD (которая записывает результат в регистр R1) за команду перехода. В случае (b) слот задержки заполняется командой, находящейся по целевому адресу команды перехода. Обычно такую команду приходится копировать, поскольку к ней возможны обращения и из других частей программы. Стратегии (b) отдается предпочтение, когда с высокой вероятностью переход является выполняемым, например, если это переход на начало цикла.

Наконец, слот задержки может заполняться командой, находящейся между командой невыполняемого перехода и командой, находящейся по целевому адресу, как в случае (c). Чтобы подобная оптимизация была законной, необходимо, чтобы можно было все-таки выполнить команду

SUB, если переход пойдет не по прогнозируемому направлению. При этом мы предполагаем, что команда SUB выполнит ненужную работу, но вся программа при этом будет выполняться корректно. Это, например, может быть в случае, если регистр R4 используется только для временного хранения промежуточных результатов вычислений, когда переход выполняется не по прогнозируемому направлению.

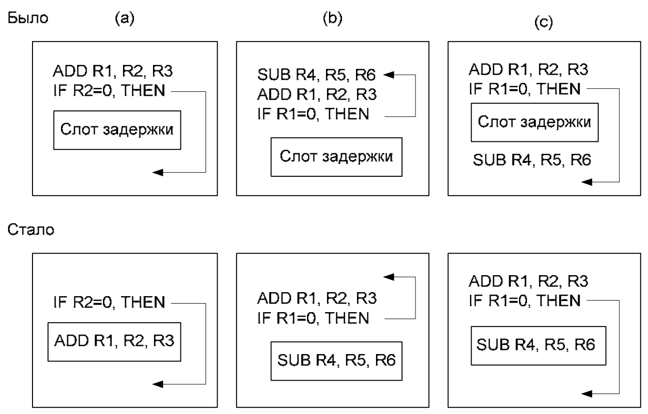


Рис. 39. Требования к переставляемым командам
при планировании задержанного перехода

Ниже в табл. 13 показаны различные ограничения для всех этих схем планирования задержанных условных переходов, а также ситуации, в которых они дают выигрыш. Компилятор должен соблюдать требования при подборе подходящей команды для заполнения слота задержки. Если такой команды не находится, слот задержки должен заполняться пустой операцией.

Таблица 13

Ограничения схем планирования задержанных условных переходов

Рассматриваемый случай	Требования	Когда увеличивается производительность
(a)	Команда условного перехода не должна зависеть от переставляемой команды	Всегда
(b)	Выполнение переставляемой команды должно быть корректным, даже если переход не выполняется. Может потребоваться копирование команды	Когда переход выполняется. Может увеличивать размер программы в случае копирования команды
(c)	Выполнение переставляемой команды должно быть корректным, даже если переход выполняется	Когда переход не выполняется

Планирование задержанных переходов осложняется наличием ограничений на команды, размещение которых планируется в слотах задержки и необходимостью предсказывать во время компиляции, будет ли условный переход выполняемым или нет. Рисунок 40 дает общее представление об эффективности планирования переходов для простейшего конвейера с одним слотом задержки перехода при использовании простого алгоритма планирования.

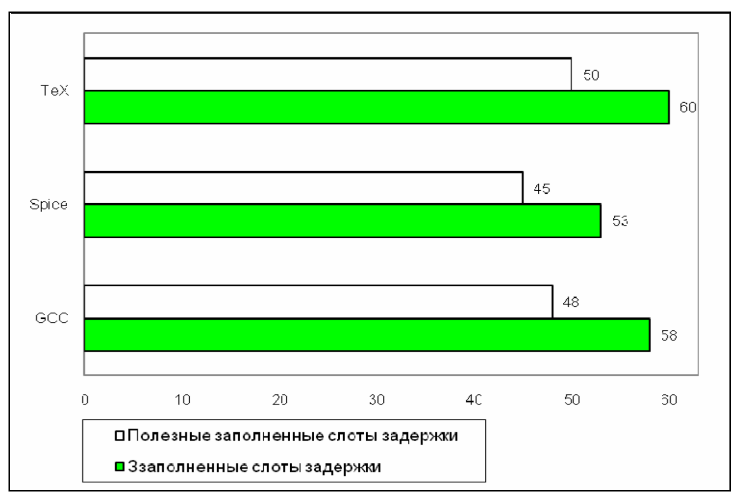


Рис. 40. Частота заполнения одного слота задержки условного перехода

Этот рисунок показывает, что больше половины слотов задержки переходов оказываются заполненными. При этом почти 80% заполненных слотов оказываются полезными для выполнения программы. Это может показаться удивительным, поскольку условные переходы являются выполняемыми примерно в 53% случаев. Высокий процент использования заполненных слотов объясняется тем, что примерно половина из них заполняется командами, предшествовавшими команде условного перехода (стратегия (а)), выполнение которых необходимо независимо от того, выполняется ли переход, или нет.

Имеются небольшие дополнительные затраты аппаратуры на реализацию задержанных переходов. Из-за задержанного эффекта условных переходов, для корректного восстановления состояния в случае появления прерывания нужны несколько счетчиков команд (один плюс длина задержки).

Статическое прогнозирование условных переходов

Имеются два основных метода, которые можно использовать для статического предсказания переходов: метод исследования структуры программы и метод использования информации о профиле выполнения программы, который собран в результате предварительных запусков программы. Использование структуры программы достаточно просто: в качестве исходной точки можно предположить, например, что все идущие назад по программе переходы являются выполняемыми, а идущие вперед по программе – невыполняемыми. Однако эта схема не очень эффективна для большинства программ. Основываясь только на структуре программы просто трудно сделать лучший прогноз.

Альтернативная техника для предсказания переходов основана на информации о профиле выполнения программы, собранной во время предыдущих прогонов. Ключевым моментом, который делает этот подход заслуживающим внимания, является то, что поведение переходов при выполнении программы часто повторяется, т. е. каждый отдельный переход в

программе часто оказывается смещенным в одну из сторон: он либо выполняемый, либо невыполняемый. Проведенные многими авторами исследования показывают достаточно успешное предсказание переходов с использованием этой стратегии.

5. Проблемы реализации точного прерывания в конвейере. Обработка многотактных операций и механизмы обходов в длинных конвейерах

Проблемы реализации точного прерывания в конвейере. Обработка прерываний в конвейерной машине оказывается более сложной из-за того, что совмещенное выполнение команд затрудняет определение возможности безопасного изменения состояния машины произвольной командой. В конвейерной машине команда выполняется по этапам, и ее завершение осуществляется через несколько тактов после выдачи для выполнения. Еще в процессе выполнения отдельных этапов команда может изменить состояние машины. Тем временем возникшее прерывание может вынудить машину прервать выполнение еще не завершенных команд.

Как и в неконвейерных машинах двумя основными проблемами при реализации прерываний являются: (1) прерывания возникают в процессе выполнения некоторой команды; (2) необходим механизм возврата из прерывания для продолжения выполнения программы. Например, для нашего простейшего конвейера прерывание по отсутствию страницы виртуальной памяти при выборке данных не может произойти до этапа выборки из памяти (MEM). В момент возникновения этого прерывания в процессе обработки уже будут находиться несколько команд. Поскольку подобное прерывание должно обеспечить возврат для продолжения программы и требует переключения на другой процесс (операционную систему), необходимо надежно очистить конвейер и сохранить состояние машины таким, чтобы повторное выполнение команды после возврата из прерывания осуществлялось при корректном состоянии машины. Обычно это реализуется путем сохранения адреса команды (PC), вызвавшей прерывание. Если выбранная после возврата из прерывания команда не является командой перехода, то сохраняется обычная последовательность выборки и обработки команд в конвейере. Если же это команда перехода, то мы должны оценить условие перехода и в зависимости от выбранного направления начать выборку либо по целевому адресу команды перехода, либо следующей за переходом команды. Когда происходит прерывание, для корректного сохранения состояния машины необходимо выполнить следующие шаги:

1. В последовательность команд, поступающих на обработку в конвейер, принудительно вставить команду перехода на прерывание.

2. Пока выполняется команда перехода на прерывание, погасить все требования записи, выставленные командой, вызвавшей прерывание, а также всеми следующими за ней в конвейере командами. Эти действия позволяют предотвратить все изменения состояния машины командами, которые не завершились к моменту начала обработки прерывания.

3. После передачи управления подпрограмме обработки прерываний операционной системы, она немедленно должна сохранить значение адреса команды (PC), вызвавшей прерывание. Это значение будет использоваться позже для организации возврата из прерывания.

Если используются механизмы задержанных переходов, состояние машины уже невозможно восстановить с помощью одного счетчика команд, поскольку в процессе восстановления команды в конвейере могут оказаться вовсе не последовательными. В частности, если команда, вызвавшая прерывание, находилась в слоте задержки перехода и переход был выполненным, то необходимо заново повторить выполнение команд из слота задержки плюс команду, находящуюся по целевому адресу команды перехода. Сама команда перехода уже выполнилась и ее повторения не требуется. При этом адреса команд из слота задержки перехода и целевой адрес команды перехода естественно не являются последовательными. Поэтому необходимо сохранять и восстанавливать несколько счетчиков команд, число которых на единицу превышает длину слота задержки. Это выполняется на третьем шаге обработки прерывания.

После обработки прерывания специальные команды осуществляют возврат из прерывания путем перезагрузки счетчиков команд и инициализации потока команд. Если конвейер может быть остановлен так, что команды, непосредственно предшествовавшие вызвавшей прерывание команде, завершаются, а следовавшие за ней могут быть заново запущены для выполнения, то говорят, что конвейер обеспечивает точное прерывание. В идеале команда, вызывающая прерывание, не должна менять состояние машины, и для корректной обработки некоторых типов прерываний требуется, чтобы команда, вызывающая прерывание, не имела никаких побочных эффектов. Для других типов прерываний, например, для прерываний по исключительным ситуациям плавающей точки, вызывающая прерывание команда на некоторых машинах записывает свои результаты еще до того момента, когда прерывание может быть обработано. В этих случаях аппаратура должна быть готовой для восстановления операндов-источников, даже если местоположение результата команды совпадает с местоположением одного из операндов-источников.

Поддержка точных прерываний во многих системах является обязательным требованием, а в некоторых системах была бы весьма желательной, поскольку она упрощает интерфейс операционной системы. Как минимум в машинах со страничной организацией памяти или с реализацией

арифметической обработки в соответствии со стандартом IEEE средства обработки прерываний должны обеспечивать точное прерывание либо целиком с помощью аппаратуры, либо с помощью некоторой поддержки со стороны программных средств.

Необходимость реализации в машине точных прерываний иногда оспаривается из-за некоторых проблем, которые осложняют повторный запуск команд. Повторный запуск сложен из-за того, что команды могут изменить состояние машины еще до того, как они гарантировано завершают свое выполнение (иногда гарантированное завершение команды называется фиксацией команды или фиксацией результатов выполнения команды). Поскольку команды в конвейере могут быть взаимозависимыми, блокировка изменения состояния машины может оказаться непрактичной, если конвейер продолжает работать. Таким образом, по мере увеличения степени конвейеризации машины возникает необходимость отката любого изменения состояния, выполненного до фиксации команды. К счастью, в простых конвейерах, подобных рассмотренному, эти проблемы не возникают. В таблице 14 показаны ступени рассмотренного конвейера и причины прерываний, которые могут возникнуть на соответствующих ступенях при выполнении команд.

Таблица 14

Причины прерываний в простейшем конвейере

Ступень конвейера	Причина прерывания
IF	Ошибка при обращении к странице памяти при выборке команды; невыровненное обращение к памяти; нарушение защиты памяти
ID	Неопределенный или запрещенный код операции
EX	Арифметическое прерывание
MEM	Ошибка при обращении к странице памяти при выборке данных; невыровненное обращение к памяти; нарушение защиты памяти
WB	Отсутствует

Обработка многотактных операций и механизмы обходов в длинных конвейерах. В рассмотренном нами конвейере стадия выполнения команды (EX) составляла всего один такт, что вполне приемлемо для целочисленных операций. Однако для большинства операций плавающей точки было бы непрактично требовать, чтобы все они выполнялись за один или даже за два такта. Это привело бы к существенному увеличению такта синхронизации конвейера, либо к сверхмерному увеличению количества оборудования (объема логических схем) для реализации устройств плавающей точки. Проще всего представить, что команды плавающей точки используют тот же самый конвейер, что и целочисленные команды, но с двумя важными изменениями. Во-первых, такт EX может повторяться

многократно столько раз, сколько необходимо для выполнения операции. Во-вторых, в процессоре может быть несколько функциональных устройств, реализующих операции плавающей точки. При этом могут возникать приостановки конвейера, если выданная для выполнения команда либо вызывает структурный конфликт по функциональному устройству, которое она использует, либо существует конфликт по данным.

Допустим, что в нашей реализации процессора имеются четыре отдельных функциональных устройства (рис. 41):

1. Основное целочисленное устройство.

2. Устройство умножения целочисленных операндов и операндов с плавающей точкой.

3. Устройство сложения с плавающей точкой.

4. Устройство деления целочисленных операндов и операндов с плавающей точкой.

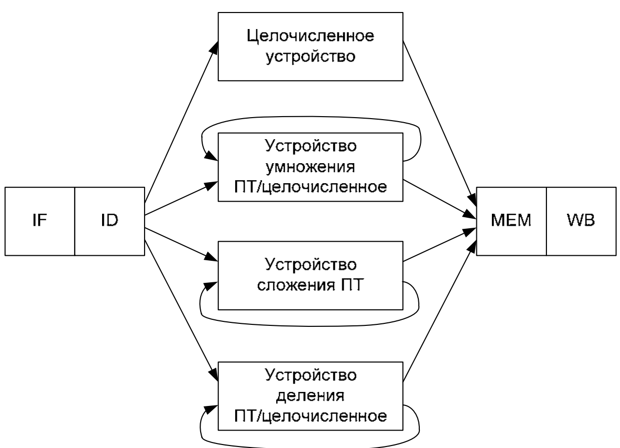


Рис. 41. Конвейер с дополнительными функциональными устройствами

Целочисленное устройство обрабатывает все команды загрузки и записи в память при работе с двумя наборами регистров (целочисленных и с плавающей точкой), все целочисленные операции (за исключением команд умножения и деления) и все команды переходов. Если предположить, что стадии выполнения других функциональных устройств неконвейерные, то рис. 41 показывает структуру такого конвейера. Поскольку стадия EX является неконвейерной, никакая команда, использующая функциональное устройство, не может быть выдана для выполнения до тех пор, пока предыдущая команда не покинет ступень EX. Более того, если команда не может поступить на ступень EX, весь конвейер за этой командой будет приостановлен.

В действительности промежуточные результаты возможно не используются циклически ступенью EX, как это показано на рис. 41, и ступень EX имеет задержки длительностью более одного такта. Мы можем

обобщить структуру конвейера плавающей точки, допустив конвейеризацию некоторых ступеней и параллельное выполнение нескольких операций. Чтобы описать работу такого конвейера, мы должны определить задержки функциональных устройств, а также скорость инициаций или скорость повторения операций. Это скорость, с которой новые операции данного типа могут поступать в функциональное устройство. Например, предположим, что имеют место следующие задержки функциональных устройств и скорости повторения операций (табл. 15).

Таблица 15

Задержки функциональных устройств и скорости повторения операций

Функциональное устройство	Задержка	Скорость повторения
Целочисленное АЛУ	1	1
Сложение с ПТ	4	2
Умножение с ПТ (и целочисленное)	6	3
Деление с ПТ (и целочисленное)	15	15

На рисунке 42 представлена структура подобного конвейера. Ее реализация требует введения конвейерной регистровой станции EX1/EX2 и модификации связей между регистрами ID/EX и EX/MEM.

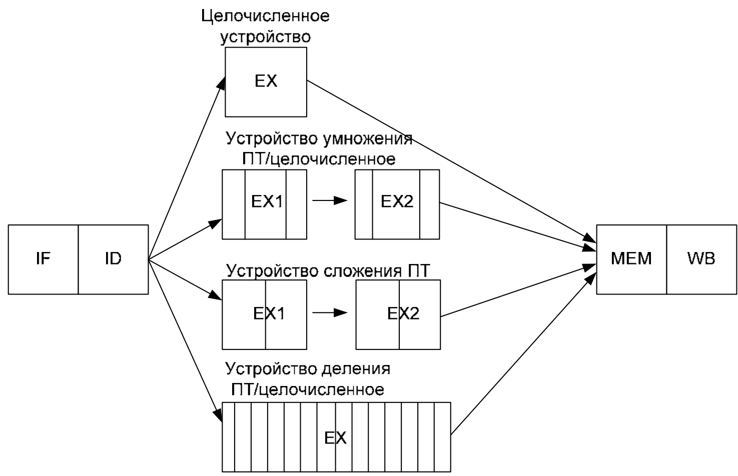


Рис. 42. Конвейер с многоступенчатыми функциональными устройствами

Конфликты и ускоренные пересылки в длинных конвейерах. Имеется несколько различных аспектов обнаружения конфликтов и организации ускоренной пересылки данных в конвейерах, подобных представленному на рис. 42:

1. Поскольку устройства не являются полностью конвейерными, в данной схеме возможны структурные конфликты. Эти ситуации необходимо обнаруживать и приостанавливать выдачу команд.

2. Поскольку устройства имеют разные времена выполнения, количество записей в регистровый файл в каждом такте может быть больше 1.

3. Возможны конфликты типа WAW, поскольку команды больше не поступают на ступень WB в порядке их выдачи для выполнения. Заметим, что конфликты типа WAR невозможны, поскольку чтение регистров всегда осуществляется на ступени ID.

4. Команды могут завершаться не в том порядке, в котором они были выданы для выполнения, что вызывает проблемы с реализацией прерываний.

Прежде чем представить общее решение для реализации схем обнаружения конфликтов, рассмотрим вторую и третью проблемы.

Если предположить, что файл регистров с ПТ имеет только один порт записи, то последовательность операций с ПТ, а также операция загрузки ПТ совместно с операциями ПТ может вызвать конфликты по порту записи в регистровый файл. Рассмотрим последовательность команд, представленную в табл. 16. В такте 10 все три команды достигнут ступени WB и должны произвести запись в регистровый файл. При наличии только одного порта записи в регистровый файл машина должна обеспечить последовательное завершение команд. Этот единственный регистровый порт является источником структурных конфликтов. Чтобы решить эту проблему, можно увеличить количество портов в регистровом файле, но такое решение может оказаться неприемлемым, поскольку эти дополнительные порты записи скорее всего будут редко использоваться. Однако в установившемся состоянии максимальное количество необходимых портов записи равно 1. Поэтому в реальных машинах разработчики предпочитают отслеживать обращения к порту записи в регистры и рассматривать одновременное к нему обращение как структурный конфликт.

Таблица 16

Пример конфликта по записи в регистровый файл

Команда	Номер такта									
	1	2	3	4	5	6	7	8	9	10
MULTD F0,F4,F6	IF	ID	EX11	EX12	EX13	EX21	EX22	EX23	MEM	WB
...		IF	ID	EX	MEM	WB				
ADDD F2,F4,F6			IF	ID	EX11	EX12	EX21	EX22	MEM	WB
...				IF	ID	EX	MEM	WB		
...					IF	ID	EX	MEM	WB	
LD F8,0(R2)						IF	ID	EX	MEM	WB

Имеется два способа для обхода этого конфликта. Первый заключается в отслеживании использования порта записи на ступени ID конвейера и приостановке выдачи команды как при структурном конфликте. Альтернативная схема предполагает приостановку конфликтующей команды, когда она пытается попасть на ступень MEM конвейера. Преимуществом такой схемы является то, что она не требует обнаружения конфликта до входа на ступень MEM, где это легче сделать. Однако подобная реализация усложняет управление конвейером, поскольку приостановки в этом случае могут возникать в двух разных местах конвейера.

Другой проблемой является возможность конфликтов типа WAW. Можно рассмотреть тот же пример, что приведен в таблице 16. Если бы команда LD была выдана на один такт раньше и имела в качестве местоположения результата регистр F2, то возник бы конфликт типа WAW, поскольку эта команда выполняла бы запись в регистр F2 на один такт раньше команды ADDD. Имеются два способа обработки этого конфликта типа WAW. Первый подход заключается в задержке выдачи команды загрузки до момента передачи команды ADDD на ступень MEM. Второй подход заключается в подавлении результата операции сложения при обнаружении конфликта и изменении управления таким образом, чтобы команда сложения не записывала свой результат. Тогда команда LD может выдаваться для выполнения сразу же. Поскольку такой конфликт является редким, обе схемы будут работать достаточно хорошо. В любом случае конфликт может быть обнаружен на ранней стадии ID, когда команда LD выдается для выполнения. Тогда приостановка команды LD или установка блокировки записи результата командой ADDD реализуются достаточно просто.

Таким образом, для обнаружения возможных конфликтов необходимо рассматривать конфликты между командами ПТ, а также конфликты между командами ПТ и целочисленными командами. За исключением команд загрузки/записи с ПТ и команд пересылки данных между регистрами ПТ и целочисленными регистрами, команды ПТ и целочисленные команды достаточно хорошо разделены, и все целочисленные команды работают с целочисленными регистрами, а команды ПТ – с регистрами ПТ. Таким образом, для обнаружения конфликтов между целочисленными командами и командами ПТ необходимо рассматривать только команды загрузки/записи с ПТ и команды пересылки регистров ПТ. Это упрощение управления конвейером является дополнительным преимуществом поддержания отдельных регистровых файлов для хранения целочисленных данных и данных с ПТ. (Главное преимущество заключается в удвоении общего количества регистров и увеличении пропускной способности без увеличения числа портов в каждом наборе). Если предположить, что конвейер выполняет обнаружение всех конфликтов на стадии ID, перед выдачей команды для выполнения в функциональные устройства должны быть выполнены три проверки:

1. Проверка наличия структурных конфликтов. Ожидание освобождения функционального устройства и порта записи в регистры, если он потребуется.

2. Проверка наличия конфликтов по данным типа RAW. Ожидание до тех пор, пока регистры-источники операндов указаны в качестве регистров результата на конвейерных станциях ID/EX (которая соответствует команде, выданной в предыдущем такте), EX1/EX2 или EX/MEM.

3. Проверка наличия конфликтов типа WAW. Проверка того, что команды, находящиеся на конвейерных станциях EX1 и EX2, не имеют в ка-

честве месторасположения результата регистр результата выдаваемой для выполнения команды. В противном случае выдача команды, находящейся на ступени ID, приостанавливается.

Хотя логика обнаружения конфликтов для многотактных операций ПТ несколько более сложная, концептуально она не отличается от такой же логики для целочисленного конвейера. То же самое касается логики для ускоренной пересылки данных. Логика ускоренной пересылки данных может быть реализована с помощью проверки того, что указанный на конвейерных станциях EX/MEM и MEM/WB регистр результата является регистром операнда команды ПТ. Если происходит такое совпадение, для пересылки данных разрешается прием по соответствующему входу мультиплексора. Многотактные операции ПТ создают также новые проблемы для механизма прерывания.

Поддержка точных прерываний в длинных конвейерах. Другая проблема, связанная с реализацией команд с большим временем выполнения, может быть проиллюстрирована с помощью следующей последовательности команд:

```
DIVF F0,F2,F4
ADDF F10,F10,F8
SUBF F12,F12,F14
```

Эта последовательность команд выглядит очень просто. В ней отсутствуют какие-либо зависимости. Однако она приводит к появлению новых проблем из-за того, что выданная раньше команда может завершиться после команды, выданной для выполнения позже. В данном примере можно ожидать, что команды ADDF и SUBF завершаться раньше, чем завершится команда DIVF. Этот эффект является типичным для конвейеров команд с большим временем выполнения и называется внеочередным завершением команд (out-of-order completion). Тогда, например, если команда DIVF вызовет арифметическое прерывание после завершения команды ADDF, мы не сможем реализовать точное прерывание на уровне аппаратуры. В действительности, поскольку команда ADDF меняет значение одного из своих операндов, невозможно даже с помощью программных средств восстановить состояние, которое было перед выполнением команды DIVF.

Имеются четыре возможных подхода для работы в условиях внеочередного завершения команд. Первый из них просто игнорирует проблему и предлагает механизмы неточного прерывания. Этот подход использовался в 60-х и 70-х годах и все еще применяется в некоторых суперкомпьютерах, в которых некоторые классы прерываний запрещены или обрабатываются аппаратурой без остановки конвейера. Такой подход трудно использовать в современных машинах при наличии концепции виртуальной памяти и стандарта на операции с плавающей точкой IEEE, которые требуют реали-

зации точного прерывания путем комбинации аппаратных и программных средств. В некоторых машинах эта проблема решается путем введения двух режимов выполнения команд: быстрого, но с возможно не точными прерываниями, и медленного, гарантирующего реализацию точных прерываний.

Второй подход заключается в буферизации результатов операции до момента завершения выполнения всех команд, предшествовавших данной. В некоторых машинах используется этот подход, но он становится все более дорогостоящим, если отличия во времени выполнения разных команд велики, поскольку становится большим количество результатов, которые необходимо буферизовать. Более того, результаты из этой буферизованной очереди необходимо пересылать для обеспечения продолжения выдачи новых команд. Это требует большого количества схем сравнения и многовходовых мультиплексоров. Имеются две вариации этого основного подхода. Первая называется буфером истории (history file), использовавшемся в машине CYBER 180/990. Буфер истории отслеживает первоначальные значения регистров. Если возникает прерывание и состояние машины необходимо откатить назад до точки, предшествовавшей некоторым завершившимся вне очереди командам, то первоначальное значение регистров может быть восстановлено из этого буфера истории. Подобная методика использовалась также при реализации автоинкрементной и автодекрементной адресации в машинах типа VAX. Другой подход называется буфером будущего (future file). Этот буфер хранит новые значения регистров. Когда все предшествующие команды завершены, основной регистровый файл обновляется значениями из этого буфера. При прерывании основной регистровый файл хранит точные значения регистров, что упрощает организацию прерывания. В следующей главе будут рассмотрены некоторые расширения этой идеи.

Третий используемый метод заключается в том, чтобы разрешить в ряде случаев неточные прерывания, но при этом сохранить достаточно информации, чтобы подпрограмма обработки прерывания могла выполнить точную последовательность прерывания. Это предполагает наличие информации о находившихся в конвейере командах и их адресов. Тогда после обработки прерывания, программное обеспечение завершает выполнение всех команд, предшествовавших последней завершившейся команде, а затем последовательность может быть запущена заново. Рассмотрим следующий наихудший случай:

Команда 1 – длинная команда, которая в конце концов вызывает прерывание.

Команда 2, ... , Команда n-1 – последовательность команд, выполнение которых не завершилось.

Команда n – команда, выполнение которой завершилось.

Имея значения адресов всех команд в конвейере и адрес возврата из прерывания, программное обеспечение может определить состояние команды 1 и команды n. Поскольку команда n завершила выполнение, хотелось бы продолжить выполнение с команды n+1. После обработки прерывания программное обеспечение должно смоделировать выполнение команд с 1 по n-1. Тогда можно осуществить возврат из прерывания на команду n+1. Наибольшая неприятность такого подхода связана с усложнением подпрограммы обработки прерывания. Но для простых конвейеров, подобных рассмотренному нами, имеются и упрощения. Если команды с 2 по n все являются целочисленными, то мы просто знаем, что в случае завершения выполнения команды n, все команды с 2 по n-1 также завершили выполнение. Таким образом, необходимо обрабатывать только операцию с плавающей точкой. Чтобы сделать эту схему работающей, количество операций ПТ, выполняющихся с совмещением, может быть ограничено. Например, если допускается совмещение только двух операций, то только прерванная команда должна завершаться программными средствами. Это ограничение может снизить потенциальную пропускную способность, если конвейеры плавающей точки являются достаточно длинными или если имеется значительное количество функциональных устройств. Такой подход использовался в архитектуре SPARC, позволяющей совмещать выполнение целочисленных операций с операциями плавающей точки.

Четвертый метод представляет собой гибридную схему, которая позволяет продолжать выдачу команд, только если известно, что все команды, предшествовавшие выдаваемой, будут завершены без прерывания. Это гарантирует, что в случае возникновения прерывания ни одна следующая за ней команда не будет завершена, а все предшествующие будут завершены. Иногда это означает необходимость приостановки машины для поддержки точных прерываний. Чтобы эта схема работала, необходимо, чтобы функциональные устройства плавающей точки определяли возможность появления прерывания на самой ранней стадии выполнения команд так, чтобы предотвратить завершение выполнения следующих команд.

6. Параллелизм на уровне выполнения команд, планирование загрузки конвейера и методика разворачивания циклов. Конвейерная суперскалярная обработка

Параллелизм на уровне выполнения команд, планирование загрузки конвейера и методика разворачивания циклов. Мы рассмотрели средства конвейеризации, которые обеспечивают совмещенный режим выполнения команд, когда они являются независимыми друг от друга. Это потенциальное совмещение выполнения команд называется параллелизмом на уровне команд. В данной главе мы рассмотрим ряд

методов развития идей конвейеризации, основанных на увеличении степени параллелизма, используемой при выполнении команд. Мы начнем с рассмотрения методов, позволяющих снизить влияние конфликтов по данным и по управлению, а затем вернемся к теме расширения возможностей процессора по использованию параллелизма, заложенного в программах. Затем мы обсудим современные технологии компиляторов, используемые для увеличения степени параллелизма уровня команд.

Для начала запишем выражение, определяющее среднее количество тактов для выполнения команды в конвейере:

CPI конвейера = CPI идеального конвейера +

+ Приостановки из-за структурных конфликтов +

+ Приостановки из-за конфликтов типа RAW +

+ Приостановки из-за конфликтов типа WAR +

+ Приостановки из-за конфликтов типа WAW +

+ Приостановки из-за конфликтов по управлению.

CPI идеального конвейера есть не что иное, как максимальная пропускная способность, достижимая при реализации. Уменьшая каждое из слагаемых в правой части выражения, мы минимизируем общий CPI конвейера и таким образом увеличиваем пропускную способность команд. Это выражение позволяет также охарактеризовать различные методы, которые будут рассмотрены в этой главе, по тому компоненту общего CPI, который соответствующий метод уменьшает. В таблице 17 перечислены некоторые методы, которые будут рассмотрены далее, и их воздействие на величину CPI.

Таблица 17

Методы уменьшения CPI

Метод	**Снижает**
Разворачивание циклов	Приостановки по управлению
Базовое планирование конвейера	Приостановки RAW
Динамической планирование с централизованной схемой управления	Приостановки RAW
Динамическое планирование с переименованием регистров	Приостановки WAR и WAW
Динамическое прогнозирование переходов	Приостановки по управлению
Выдача нескольких команд в одном такте	Идеальный CPI
Анализ зависимостей компилятором	Идеальный CPI и приостановки по данным
Программная конвейеризация и планирование трасс	Идеальный CPI и приостановки по данным
Выполнение по предположению	Все приостановки по данным и управлению
Динамическое устранение неоднозначности памяти	Приостановки RAW, связанные с памятью

Данные методы основываются на следующих концепциях.

Параллелизм уровня команд: зависимости и конфликты по данным. Все рассматриваемые методы используют параллелизм, заложенный в последовательности команд. Как мы установили выше этот тип параллелизма называется параллелизмом уровня команд или ILP. Степень параллелизма, доступная внутри базового блока (линейной последовательности команд, переходы из вне которой разрешены только на ее вход, а переходы внутри которой разрешены только на ее выход) достаточно мала. Например, средняя частота переходов в целочисленных программах составляет около 16%. Это означает, что в среднем между двумя переходами выполняются примерно пять команд. Поскольку эти пять команд возможно взаимозависимые, то степень перекрытия, которую мы можем использовать внутри базового блока, возможно будет меньше чем пять. Чтобы получить существенное улучшение производительности, мы должны использовать параллелизм уровня команд одновременно для нескольких базовых блоков.

Самый простой и общий способ увеличения степени параллелизма, доступного на уровне команд, является использование параллелизма между итерациями цикла. Этот тип параллелизма часто называется параллелизмом уровня итеративного цикла. Ниже приведен простой пример цикла, выполняющего сложение двух 1000-элементных векторов, который является полностью параллельным:

```
for (i = 1; i <= 1000; i = i + 1)
x[i] = x[i] + y[i];
```

Каждая итерация цикла может перекрываться с любой другой итерацией, хотя внутри каждой итерации цикла практическая возможность перекрытия небольшая.

Имеется несколько методов для превращения такого параллелизма уровня цикла в параллелизм уровня команд. Эти методы основаны главным образом на разворачивании цикла либо статически, используя компилятор, либо динамически с помощью аппаратуры. Ниже в этом разделе мы рассмотрим подробный пример разворачивания цикла.

Важным альтернативным методом использования параллелизма уровня команд является использование векторных команд. По существу векторная команда оперирует с последовательностью элементов данных. Например, приведенная выше последовательность на типичной векторной машине может быть выполнена с помощью четырех команд: двух команд загрузки векторов x и y из памяти, одной команды сложения двух векторов и одной команды записи вектора-результата. Конечно, эти команды могут быть конвейеризованными и иметь относительно большие задержки выполнения, но эти задержки могут перекрываться. Векторные команды и векторные машины заслуживают отдельного рассмотрения, которое выходит за рамки данного курса. Хотя разработка идей векторной обработки предшествовала появлению большинства методов использования паралле-

лизма, которые рассматриваются в этой главе, машины, использующие параллелизм уровня команд постепенно заменяют машины, базирующиеся на векторной обработке. Более детально причины этого сдвига технологии будут рассмотрены позже.

Параллелизм уровня цикла: концепции и методы. Параллелизм уровня цикла обычно анализируется на уровне исходного текста программы или близкого к нему, в то время как анализ параллелизма уровня команд главным образом выполняется, когда команды уже сгенерированы компилятором. Анализ на уровне циклов включает определение того, какие зависимости существуют между операндами в цикле в пределах одной итерации цикла. Теперь мы будем рассматривать только зависимости по данным, которые возникают, когда операнд записывается в некоторой точке и считывается в некоторой более поздней точке. Мы обсудим коротко зависимости по именам. Анализ параллелизма уровня цикла фокусируется на определении того, зависят ли по данным обращения к данным в последующей итерации от значений данных, вырабатываемых в более ранней итерации.

Рассмотрим следующий цикл:

```
for (i=1; i<=100; i=i+1) {
A[i+1] = A[i] + C[i]; /* S1 */
B[i+1] = B[i] + A[i+1];} /*S2*/
}
```

Предположим, что A, B и C представляют собой отдельные, неперекрывающиеся массивы. (На практике иногда массивы могут быть теми же самыми или перекрываться. Поскольку массивы могут передаваться в качестве параметров некоторой процедуре, которая содержит этот цикл, определение того, перекрываются ли массивы или они совпадают, требует изощренного, межпроцедурного анализа программы). Какие зависимости по данным имеют место между операторами этого цикла?

Имеются две различных зависимости:

1. S1 использует значение, вычисляемое оператором S1 на более ранней итерации, поскольку итерация i вычисляет A[i+1], которое считывается в итерации i+1. То же самое справедливо для оператора S2 для B[i] и B[i+1].

2. S2 использует значение A[i+1], вычисляемое оператором S1 в той же самой итерации.

Эти две зависимости отличаются друг от друга и имеют различный эффект. Чтобы увидеть, чем они отличаются, предположим, что в каждый момент времени существует только одна из этих зависимостей. Рассмотрим зависимость оператора S1 от более ранней итерации S1. Эта зависимость (loop-carried dependence) означает, что между различными итерациями цикла существует зависимость по данным. Более того, поскольку

оператор S1 зависит от самого себя, последовательные итерации оператора S1 должны выполняться упорядочено.

Вторая зависимость (S2 зависит от S1) не передается от итерации к итерации. Таким образом, если бы это была единственная зависимость, несколько итераций цикла могли бы выполняться параллельно, при условии, что каждая пара операторов в итерации поддерживается в заданном порядке.

Имеется третий тип зависимостей по данным, который возникает в циклах, как показано в следующем примере.

Рассмотрим цикл:

```
for (i=1; i<=100; i=i+1) {
A[i] = A[i] + B[i]; /* S1 */
B[i+1] = C[i] + D[i]; /* S2 */
}
```

Оператор S1 использует значение, которое присваивается оператором S2 в предыдущей итерации, так что имеет место зависимость между S2 и S1 между итерациями.

Несмотря на эту зависимость, этот цикл может быть сделан параллельным. Как и в более раннем цикле эта зависимость не циклическая: ни один из операторов не зависит сам от себя и хотя S1 зависит от S2, S2 не зависит от S1. Цикл является параллельным, если только отсутствует циклическая зависимость.

Хотя в вышеприведенном цикле отсутствуют циклические зависимости, чтобы выявить параллелизм, он должен быть преобразован в другую структуру. Здесь следует сделать два важных замечания:

1. Зависимость от S1 к S2 отсутствует. Если бы она была, то в зависимостях появился бы цикл и цикл не был бы параллельным. Вследствие отсутствия других зависимостей, перестановка двух операторов не будет влиять на выполнение оператора S2.

2. В первой итерации цикла оператор S1 зависит от значения B[1], вычисляемого перед началом цикла.

Эти два замечания позволяют нам заменить выше приведенный цикл следующей последовательностью:

```
A[1] = A[1] + B[1];
for (i=1; i<=99; i=i+1) {
B[i+1] = C[i] + D[i];
A[i+1] = A[i+1] + B[i+1];
}
B[101] = C[100] + D[100];
```

Теперь итерации цикла могут выполняться с перекрытием, при условии, что операторы в каждой итерации выполняются в заданном порядке.

Имеется множество такого рода преобразований, которые реструктурируют цикл для выявления параллелизма.

Основное внимание в оставшейся части этой главы сосредоточено на методах выявления параллелизма уровня команд. Зависимости по данным в откомпилированных программах представляют собой ограничения, которые оказывают влияние на то, какая степень параллелизма может быть использована. Вопрос заключается в том, чтобы подойти к этому пределу путем минимизации действительных конфликтов и связанных с ними приостановок конвейера. Методы, которые мы изучаем становятся все более изощренными в стремлении использования всего доступного параллелизма при поддержании истинных зависимостей по данным в коде программы. Как компилятор, так и аппаратура здесь играют свою роль: компилятор старается устранить или минимизировать зависимости, в то время как аппаратура старается предотвратить превращение зависимостей в приостановки конвейера.

Основы планирования загрузки конвейера и разворачивание циклов. Для поддержания максимальной загрузки конвейера должен использоваться параллелизм уровня команд, основанный на выявлении последовательностей несвязанных команд, которые могут выполняться в конвейере с совмещением. Чтобы избежать приостановки конвейера зависимая команда должна быть отделена от исходной команды на расстояние в тактах, равное задержке конвейера для этой исходной команды. Способность компилятора выполнять подобное планирование зависит как от степени параллелизма уровня команд, доступного в программе, так и от задержки функциональных устройств в конвейере. В рамках этой лекции мы будем предполагать задержки, показанные в табл. 18, если только явно не установлены другие задержки. Мы предполагаем, что условные переходы имеют задержку в один такт, так что команда следующая за командой перехода не может быть определена в течение одного такта после команды условного перехода. Мы предполагаем, что функциональные устройства полностью конвейеризованы или дублированы (столько раз, какова глубина конвейера), так что операция любого типа может выдаваться для выполнения в каждом такте и структурные конфликты отсут-ствуют.

Таблица 18

Основные задержки между командами

Команда, вырабатывающая результат	Команда, использующая результат	Задержка в тактах
Операция АЛУ с ПТ	Другая операция АЛУ с ПТ	3
Операция АЛУ с ПТ	Запись двойного слова	2
Загрузка двойного слова	Другая операция АЛУ с ПТ	1
Загрузка двойного слова	Запись двойного слова	0

В данном коротком разделе мы рассмотрим вопрос о том, каким образом компилятор может увеличить степень параллелизма уровня команд путем разворачивания циклов. Для иллюстрации этих методов мы будем использовать простой цикл, который добавляет скалярную величину к вектору в памяти; это параллельный цикл, поскольку зависимость между итерациями цикла отсутствует. Мы предполагаем, что первоначально в регистре R1 находится адрес последнего элемента вектора (например, элемент с наибольшим адресом), а в регистре F2 – скалярная величина, которая должна добавляться к каждому элементу вектора. Программа для машины, не рассчитанная на использование конвейера, будет выглядеть примерно так:

```
Loop: LD F0,0(R1) ;F0=элемент вектора
ADDD F4,F0,F2 ;добавляет скаляр из F2
SD 0(R1),F4 ;запись результата
SUBI R1,R1,#8 ;пересчитать указатель
;8 байт (в двойном слове)
BNEZ R1, Loop ;переход R1!=нулю
```

Для упрощения мы предполагаем, что массив начинается с ячейки 0. Если бы он находился в любом другом месте, цикл потребовал бы наличия одной дополнительной целочисленной команды для выполнения сравнения с регистром R1.

Рассмотрим работу этого цикла при выполнении на простом конвейере с задержками, показанными в таблице 18.

Если не делать никакого планирования, работа цикла будет выглядеть следующим образом:

```
Такт выдачи
Loop: LD F0,0(R1) 1
приостановка 2
ADDD F4,F0,F2 3
приостановка 4
приостановка 5
SD 0(R1),F4 6
SUBI R1,R1,#8 7
BNEZ R1,Loop 8
приостановка 9
```

Для его выполнения потребуется 9 тактов на итерацию: одна приостановка для команды LD, две для команды ADDD, и одна для задержанного перехода. Мы можем спланировать цикл так, чтобы получить

```
Loop: LD F0,0(R1) 1
приостановка 2
ADDD F4,F0,F2 3
SUBI R1,R1,#8 4
BNEZ R1,Loop; задержанный переход 5
```

SD 8(R1),F4; команда изменяется, когда 6

;меняется местами с командой SUB1

Время выполнения уменьшилось с 9 до 6 тактов.

Заметим, что для планирования задержанного перехода компилятор должен определить, что он может поменять местами команды SUB1 и SD путем изменения адреса в команде записи SD: Адрес был равен 0(R1), а теперь равен 8(R1). Это не тривиальная задача, поскольку большинство компиляторов будут видеть, что команда SD зависит от SUB1, и откажутся от такой перестановки мест. Более изощренный компилятор смог бы рассчитать отношения и выполнить перестановку. Цепочка зависимостей от команды LD к команде ADDD и далее к команде SD определяет количество тактов, необходимое для данного цикла.

В вышеприведенном примере мы завершаем одну итерацию цикла и выполняем запись одного элемента вектора каждые 6 тактов, но действительная работа по обработке элемента вектора отнимает только 3 из этих 6 тактов (загрузка, сложение и запись). Оставшиеся 3 такта составляют накладные расходы на выполнение цикла (команды SUB1, BNEZ и приостановка). Чтобы устранить эти три такта нам нужно иметь больше операций в цикле относительно числа команд, связанных с накладными расходами. Одним из наиболее простых методов увеличения числа команд по отношению к команде условного перехода и команд, связанных с накладными расходами, является разворачивание цикла. Такое разворачивание выполняется путем многократной репликации (повторения) тела цикла и коррекции соответствующего кода конца цикла.

Разворачивание циклов может также использоваться для улучшения планирования. В этом случае, мы можем устранить приостановку, связанную с задержкой команды загрузки путем создания дополнительных независимых команд в теле цикла. Затем компилятор может планировать эти команды для помещения в слот задержки команды загрузки. Если при разворачивании цикла мы просто реплицируем команды, то результирующие зависимости по именам могут помешать нам эффективно спланировать цикл. Таким образом, для разных итераций хотелось бы использовать различные регистры, что увеличивает требуемое число регистров.

Представим теперь этот цикл развернутым так, что имеется четыре копии тела цикла, предполагая, что R1 первоначально кратен 4. Устраним при этом любые очевидные излишние вычисления и не будем пользоваться повторно никакими регистрами.

Ниже приведен результат, полученный путем слияния команд SUB1 и выбрасывания ненужных операций BNEZ, которые дублируются при разворачивании цикла.

Loop: LD F0,0(R1)

ADDD F4,F0,F2

```
SD 0(R1),F4 ;выбрасывается SUB1 и BNEZ
LD F6,-8(R1)
ADDD F8,F6,F2
SD -8(R1),F8 ;выбрасывается SUB1 и BNEZ
LD F10,-16(R1)
ADDD F12,F10,F2
SD -16(R1),F12 ;выбрасывается SUB1 и BNEZ
LD F14,-24(R1)
ADDD F16,F14,F2
SD -24(R1),F16
SUB1 R1,R1,#32
BNEZ R1, Loop
```

Мы ликвидировали три условных перехода и три операции декрементирования R1. Адреса команд загрузки и записи были скорректированы так, чтобы позволить слить команды SUB1 в одну команду по регистру R1. При отсутствии планирования за каждой командой здесь следует зависимая команда и это будет приводить к приостановкам конвейера. Этот цикл будет выполняться за 27 тактов (на каждую команду LD потребуется 2 такта, на каждую команду ADDD – 3, на условный переход – 2 и на все другие команды 1 такт) или по 6.8 такта на каждый из четырех элементов. Хотя эта развернутая версия в такой редакции медленнее, чем оптимизированная версия исходного цикла, после оптимизации самого развернутого цикла ситуация изменится. Обычно разворачивание циклов выполняется на более ранних стадиях процесса компиляции, так что избыточные вычисления могут быть выявлены и устранены оптимизатором.

В реальных программах мы обычно не знаем верхней границы цикла. Предположим, что она равна n и мы хотели бы развернуть цикл так, чтобы иметь k копий тела цикла. Вместо единственного развернутого цикла мы генерируем пару циклов. Первый из них выполняется (n mod k) раз и имеет тело первоначального цикла. Развернутая версия цикла окружается внешним циклом, который выполняется (n div k) раз.

В вышеприведенном примере разворачивание цикла увеличивает производительность этого цикла путем устранения команд, связанных с накладными расходами цикла, хотя оно заметно увеличивает размер программного кода. Насколько увеличится производительность, если цикл будет оптимизироваться?

Ниже представлен развернутый цикл из предыдущего примера после оптимизации.

```
Loop: LD F0,0(R1)
LD F6,-8(R1)
LD F10,-16(R1)
LD F14,-24(R1)
```

```
ADDD F4,F0,F2
ADDD F8,F6,F2
ADDD F12,F10,F2
ADDD F16,F14,F2
SD 0(R1),F4
SD -8(R1),F8
SD -16(R1),F12
SUB1 R1,R1,#32
BNEZ R1, Loop
SD 8(R1),F16 ; 8 - 32 = -24
```

Время выполнения развернутого цикла снизилось до 14 тактов или до 3.5 тактов на элемент, по сравнению с 6.8 тактов на элемент до оптимизации, и по сравнению с 6 тактами при оптимизации без разворачивания цикла.

Выигрыш от оптимизации развернутого цикла даже больше, чем от оптимизации первоначального цикла. Это произошло потому, что разворачивание цикла выявило больше вычислений, которые могут быть оптимизированы для минимизации приостановок конвейера; приведенный выше программный код выполняется без приостановок. При подобной оптимизации цикла необходимо осознавать, что команды загрузки и записи являются независимыми и могут чередоваться. Анализ зависимостей по данным позволяет нам определить, являются ли команды загрузки и записи независимыми.

Разворачивание циклов представляет собой простой, но полезный метод увеличения размера линейного кодового фрагмента, который может эффективно оптимизироваться. Это преобразование полезно на множестве машин от простых конвейеров, подобных рассмотренному ранее, до суперскалярных конвейеров, которые обеспечивают выдачу для выполнения более одной команды в такте. В следующем разделе рассмотрены методы, которые используются аппаратными средствами для динамического планирования загрузки конвейера и сокращения приостановок из-за конфликтов типа RAW, аналогичные рассмотренным выше методам компиляции.

Конвейерная суперскалярная обработка. Методы минимизации приостановок работы конвейера из-за наличия в программах логических зависимостей по данным и по управлению, рассмотренные выше, были нацелены на достижение идеального CPI (среднего количества тактов на выполнение команды в конвейере), равного 1. Чтобы еще больше повысить производительность процессора необходимо сделать CPI меньшим, чем 1. Однако этого нельзя добиться, если в одном такте выдается на выполнение только одна команда. Следовательно необходима параллельная выдача нескольких команд в каждом такте. Существуют два типа подобного рода машин: суперскалярные машины и VLIW-машины (машины с очень длин-

ным командным словом). Суперскалярные машины могут выдавать на выполнение в каждом такте переменное число команд, и работа их конвейеров может планироваться как статически с помощью компилятора, так и с помощью аппаратных средств динамической оптимизации. В отличие от суперскалярных машин, VLIW-машины выдают на выполнение фиксированное количество команд, которые сформатированы либо как одна большая команда, либо как пакет команд фиксированного формата. Планирование работы VLIW-машины всегда осуществляется компилятором. Подробнее этот вид машин рассмотрен далее в 10 разделе.

Суперскалярные машины используют параллелизм на уровне команд путем посылки нескольких команд из обычного потока команд в несколько функциональных устройств. Дополнительно, чтобы снять ограничения последовательного выполнения команд, эти машины используют механизмы внеочередной выдачи и внеочередного завершения команд, прогнозирование переходов, кэши целевых адресов переходов и условное (по предположению) выполнение команд. Возросшая сложность, реализуемая этими механизмами, создает также проблемы реализации точного прерывания.

В типичной суперскалярной машине аппаратура может осуществлять выдачу от одной до восьми команд в одном такте. Обычно эти команды должны быть независимыми и удовлетворять некоторым ограничениям, например таким, что в каждом такте не может выдаваться более одной команды обращения к памяти. Если какая-либо команда в потоке команд является логически зависимой или не удовлетворяет критериям выдачи, на выполнение будут выданы только команды, предшествующие данной. Поэтому скорость выдачи команд в суперскалярных машинах является переменной. Это отличает их от VLIW-машин, в которых полную ответственность за формирование пакета команд, которые могут выдаваться одновременно, несет компилятор, а аппаратура в динамике не принимает никаких решений относительно выдачи нескольких команд.

Предположим, что машина может выдавать на выполнение две команды в одном такте. Одной из таких команд может быть команда загрузки регистров из памяти, записи регистров в память, команда переходов, операции целочисленного АЛУ, а другой может быть любая операция плавающей точки. Параллельная выдача целочисленной операции и операции с плавающей точкой намного проще, чем выдача двух произвольных команд. В реальных системах (например, в микропроцессорах PA7100, hyperSPARC, Pentium и др.) применяется именно такой подход. В более мощных микропроцессорах (например, MIPS R10000, UltraSPARC, PowerPC 620 и др.) реализована выдача до четырех команд в одном такте.

Выдача двух команд в каждом такте требует одновременной выборки и декодирования по крайней мере 64 бит. Чтобы упростить декодирова-

ние можно потребовать, чтобы команды располагались в памяти парами и были выровнены по 64-битовым границам. В противном случае необходимо анализировать команды в процессе выборки и, возможно, менять их местами в момент пересылки в целочисленное устройство и в устройство ПТ. При этом возникают дополнительные требования к схемам обнаружения конфликтов. В любом случае вторая команда может выдаваться, только если может быть выдана на выполнение первая команда. Аппаратура принимает такие решения в динамике, обеспечивая выдачу только первой команды, если условия для одновременной выдачи двух команд не соблюдаются. В таблице 19 представлена диаграмма работы подобного конвейера в идеальном случае, когда в каждом такте на выполнение выдается пара команд.

Таблица 19

Основные задержки между командами

Тип команды	Ступень конвейера							
Целочисленная команда	IF	ID	EX	MEM	WB			
Команда ПТ	IF	ID	EX	MEM	WB			
Целочисленная команда		IF	ID	EX	MEM	WB		
КомандаПТ		IF	ID	EX	MEM	WB		
Целочисленная команда			IF	ID	EX	MEM	WB	
Команда ПТ			IF	ID	EX	MEM	WB	
Целочисленная команда				IF	ID	EX	MEM	WB
Команда ПТ				IF	ID	EX	MEM	WB

Такой конвейер позволяет существенно увеличить скорость выдачи команд. Однако чтобы он смог так работать, необходимо иметь либо полностью конвейеризованные устройства плавающей точки, либо соответствующее число независимых функциональных устройств. В противном случае устройство плавающей точки станет узким горлом и эффект, достигнутый за счет выдачи в каждом такте пары команд, сведется к минимуму.

При параллельной выдаче двух операций (одной целочисленной команды и одной команды ПТ) потребность в дополнительной аппаратуре, помимо обычной логики обнаружения конфликтов, минимальна: целочисленные операции и операции ПТ используют разные наборы регистров и разные функциональные устройства. Более того, усиление ограничений на выдачу команд, которые можно рассматривать как специфические структурные конфликты (поскольку выдаваться на выполнение могут только определенные пары команд), обнаружение которых требует только анализа кодов операций. Единственная сложность возникает, только если команды представляют собой команды загрузки, записи и пересылки чисел с плавающей точкой. Эти команды создают конфликты по портам регистров

ПТ, а также могут приводить к новым конфликтам типа RAW, когда операция ПТ, которая могла бы быть выдана в том же такте, является зависимой от первой команды в паре.

Проблема регистровых портов может быть решена, например, путем реализации отдельной выдачи команд загрузки, записи и пересылки с ПТ. В случае составления ими пары с обычной операцией ПТ ситуацию можно рассматривать как структурный конфликт. Такую схему легко реализовать, но она будет иметь существенное воздействие на общую производительность. Конфликт подобного типа может быть устранен посредством реализации в регистровом файле двух дополнительных портов (для выборки и записи).

Если пара команд состоит из одной команды загрузки с ПТ и одной операции с ПТ, которая от нее зависит, необходимо обнаруживать подобный конфликт и блокировать выдачу операции с ПТ. За исключением этого случая, все другие конфликты естественно могут возникать, как и в обычной машине, обеспечивающей выдачу одной команды в каждом такте. Для предотвращения ненужных приостановок могут, правда потребоваться дополнительные цепи обхода.

Другой проблемой, которая может ограничить эффективность суперскалярной обработки, является задержка загрузки данных из памяти. В нашем примере простого конвейера команды загрузки имели задержку в один такт, что не позволяло следующей команде воспользоваться результатом команды загрузки без приостановки. В суперскалярном конвейере результат команды загрузки не может быть использован в том же самом и в следующем такте. Это означает, что следующие три команды не могут использовать результат команды загрузки без приостановки. Задержка перехода также становится длиною в три команды, поскольку команда перехода должна быть первой в паре команд. Чтобы эффективно использовать параллелизм, доступный на суперскалярной машине, нужны более сложные методы планирования потока команд, используемые компилятором или аппаратными средствами, а также более сложные схемы декодирования команд.

Рассмотрим, например, что дает разворачивание циклов и планирование потока команд для суперскалярного конвейера. Ниже представлен цикл, который мы уже разворачивали и планировали его выполнение на простом конвейере.

```
Loop: LD F0,0(R1) ;F0=элемент вектора
ADDD F4,F0,F2 ;добавление скалярной величины из F2
SD 0(R1),F4 ;запись результата
SUBI R1,R1,#8 ;декрементирование указателя
;8 байт на двойное слово
BNEZ R1,Loop ;переход R1!=нулю
```

Чтобы спланировать этот цикл для работы без задержек, необходимо его развернуть и сделать пять копий тела цикла. После такого разворачи-

вания цикл будет содержать по пять команд LD, ADDD, и SD, а также одну команду SUBI и один условный переход BNEZ. Развернутая и оптимизированная программа этого цикла дана ниже в табл. 20.

Таблица 20

Развернутая и оптимизированная программа цикла

Целочисленная команда	Команда ПТ	Номер такта
Loop: LD F0,0(R1)		1
LD F8,-8(R1)		2
LD F10,-16(R1)		3
LD F14,-24(R1)		4
LD F18,-32(R1)	ADDD F4,F0,F2	5
SD 0(R1),F4	ADDD F8,F6,F2	6
SD -8(R1),F8	ADDD F12,F10,F2	7
SD -16(R1),F12	ADDD F16,F14,F2	8
SD -24(R1),F16	ADDD F20,F18,F2	9
SUBI R1,R1,#40		10
BNEZ R1,Loop		11
SD -32(R1),F20		12

Этот развернутый суперскалярный цикл теперь работает со скоростью 12 тактов на итерацию, или 2.4 такта на один элемент (по сравнению с 3.5 тактами для оптимизированного развернутого цикла на обычном конвейере). В этом примере производительность суперскалярного конвейера ограничена существующим соотношением целочисленных операций и операций ПТ, но команд ПТ не достаточно для поддержания полной загрузки конвейера ПТ. Первоначальный оптимизированный неразвернутый цикл выполнялся со скоростью 6 тактов на итерацию, вычисляющую один элемент. Мы получили таким образом ускорение в 2.5 раза, больше половины которого произошло за счет разворачивания цикла. Чистое ускорение за счет суперскалярной обработки дало улучшение примерно в 1.5 раза.

В лучшем случае такой суперскалярный конвейер позволит выбирать две команды и выдавать их на выполнение, если первая из них является целочисленной, а вторая – с плавающей точкой. Если это условие не соблюдается, что легко проверить, то команды выдаются последовательно. Это показывает два главных преимущества суперскалярной машины по сравнению с WLIW-машиной. Во-первых, малое воздействие на плотность кода, поскольку машина сама определяет, может ли быть выдана следующая команда, и нам не надо следить за тем, чтобы команды соответствовали возможностям выдачи. Во-вторых, на таких машинах могут работать неоптимизированные программы, или программы, откомпилированные в расчете на более старую реализацию. Конечно, такие программы не могут работать очень хорошо. Один из способов улучшить ситуацию заключается в использовании аппаратных средств динамической оптимизации.

В общем случае в суперскалярной системе команды могут выполняться параллельно и возможно не в порядке, предписанном программой.

Если не предпринимать никаких мер, такое неупорядоченное выполнение команд и наличие множества функциональных устройств с разными временами выполнения операций могут приводить к дополнительным трудностям. Например, при выполнении некоторой длинной команды с плавающей точкой (команды деления или вычисления квадратного корня) может возникнуть исключительная ситуация уже после того, как завершилось выполнение более быстрой операции, выданной после этой длинной команды. Для того, чтобы поддерживать модель точных прерываний, аппаратура должна гарантировать корректное состояние процессора при прерывании для организации последующего возврата.

Обычно в машинах с неупорядоченным выполнением команд предусматриваются дополнительные буферные схемы, гарантирующие завершение выполнения команд в строгом порядке, предписанном программой. Такие схемы представляют собой некоторый буфер «истории», т. е. аппаратную очередь, в которую при выдаче попадают команды и текущие значения регистров результата этих команд в заданном программой порядке.

В момент выдачи команды на выполнение она помещается в конец этой очереди, организованной в виде буфера FIFO (первый вошел – первый вышел). Единственный способ для команды достичь головы этой очереди – завершение выполнения всех предшествующих ей операций. При неупорядоченном выполнении некоторая команда может завершить свое выполнение, но все еще будет находиться в середине очереди. Команда покидает очередь, когда она достигает головы очереди и ее выполнение завершается в соответствующем функциональном устройстве. Если команда находится в голове очереди, но ее выполнение в функциональном устройстве не закончено, она очередь не покидает. Такой механизм может поддерживать модель точных прерываний, поскольку вся необходимая информация хранится в буфере и позволяет скорректировать состояние процессора в любой момент времени.

Этот же буфер «истории» позволяет реализовать и условное (speculative) выполнение команд (выполнение по предположению), следующих за командами условного перехода. Это особенно важно для повышения производительности суперскалярных архитектур. Статистика показывает, что на каждые шесть обычных команд в программах приходится в среднем одна команда перехода. Если задерживать выполнение следующих за командой перехода команд, потери на конвейеризацию могут оказаться просто неприемлемыми. Например, при выдаче четырех команд в одном такте в среднем в каждом втором такте выполняется команда перехода. Механизм условного выполнения команд, следующих за командой перехода, позволяет решить эту проблему. Это условное выполнение обычно связано с последовательным выполнением команд из заранее предсказанной ветви ко-

манды перехода. Устройство управления выдает команду условного перехода, прогнозирует направление перехода и продолжает выдавать команды из этой предсказанной ветви программы.

Если прогноз оказался верным, выдача команд так и будет продолжаться без приостановок. Однако если прогноз был ошибочным, устройство управления приостанавливает выполнение условно выданных команд и, если необходимо, использует информацию из буфера истории для ликвидации всех последствий выполнения условно выданных команд. Затем начинается выборка команд из правильной ветви программы. Таким образом, аппаратура, подобная буферу, истории позволяет не только решить проблемы с реализацией точного прерывания, но и обеспечивает увеличение производительности суперскалярных архитектур.

7. Зависимости. Классификация зависимостей и их применение. Устранение зависимостей по данным и механизмы динамического планирования

Зависимости. Классификация зависимостей и их применение. Чтобы точно определить, что мы понимаем под параллелизмом уровня цикла и параллелизмом уровня команд, а также для количественного определения степени доступного параллелизма, мы должны определить, что такое параллельные команды и параллельные циклы. Начнем с объяснения того, что такое пара параллельных команд. Две команды являются параллельными, если они могут выполняться в конвейере одновременно без приостановок, предполагая, что конвейер имеет достаточно ресурсов (структурные конфликты отсутствуют).

Поэтому, если между двумя командами существует взаимозависимость, то они не являются параллельными. Имеется три типа зависимостей: зависимости по данным, зависимости по именам и зависимости по управлению. Команда j зависит по данным от команды i, если имеет место любое из следующих условий:

1) команда i вырабатывает результат, который использует команда j;

2) команда j является зависимой по данным от команды k, а команда k является зависимой по данным от команды i.

Второе условие просто означает, что одна команда зависит от другой, если между этими двумя командами имеется цепочка зависимостей первого типа. Эта цепочка зависимостей может быть длиною во всю программу.

Если две команды являются зависимыми по данным, они не могут выполняться одновременно или полностью совмещено. Зависимость по данным предполагает, что между двумя командами имеется цепочка из одного или нескольких конфликтов типа RAW. Одновременное выполнение

таких команд требует создания машины с внутренними схемами блокировок конвейера, обеспечивающих обнаружение конфликтов и уменьшение времени приостановок или полное устранение перекрытия. В машине без внутренних блокировок, которые базируются на программном планировании работы конвейера компилятором, компилятор не может спланировать зависимые команды так, чтобы они полностью совмещались, поскольку в противном случае программа не будет выполняться правильно. Наличие зависимостей по данным в последовательности команд отражает зависимость по данным в исходном тексте программы, на основании которого она генерировалась. Эффект первоначальной зависимости по данным должен сохраняться.

Зависимости являются свойством программ. Приведет ли данная зависимость к обнаруживаемому конфликту и вызовет ли данный конфликт реальную приостановку конвейера, зависит от организации конвейера. Действительно, многие методы, рассматриваемые в этой главе, позволяют обойти конфликты или обойти необходимость приостановки конвейера в случае возникновения конфликта, при сохранении зависимости. Важность зависимостей по данным заключается в том, что именно они устанавливают верхнюю границу степени параллелизма, который вероятно может быть использован. Наличие зависимостей по данным означает также, что результаты должны вычисляться в определенном порядке, поскольку более поздняя команда зависит от результата предыдущей.

Данные могут передаваться от команды к команде либо через регистры, либо через ячейки памяти. Когда данные передаются через регистры, обнаружение зависимостей значительно упрощается, поскольку имена регистров зафиксированы в командах (хотя этот процесс становится более сложным, если вмешиваются условные переходы). Зависимости по данным, которые передаются через ячейки памяти, обнаружить значительно сложнее, поскольку два адреса могут относиться к одной и той же ячейке памяти, но внешне выглядят по разному (например, 100(R4) и 20(R6) могут определять один и тот же адрес). Кроме того, эффективный адрес команды загрузки или записи может меняться от одного выполнения команды к другому (так что 20(R4) и 20(R4) будут определять разные адреса), еще больше усложняя обнаружение зависимости. В этой главе мы рассмотрим как аппаратные, так и программные методы обнаружения зависимостей по данным, которые связаны с ячейками памяти. Методы компиляции для обнаружения таких зависимостей являются очень важными при выявлении параллелизма уровня цикла.

Вторым типом зависимостей в программах являются зависимости по именам. Зависимости по именам возникают когда две команды используют одно и то же имя (либо регистра, либо ячейки памяти), но при отсутствии

передачи данных между командами. Имеется два типа зависимости имен между командой i, которая предшествует команде j в программе:

1. Антизависимость между командой i и командой j возникает тогда, когда команда j записывает в регистр или ячейку памяти, который(ую) команда i считывает и команда i выполняется первой. Антизависимость соответствует конфликту типа WAR, и обнаружение конфликтов типа WAR означает упорядочивание выполнения пары команд с антизависимостью.

2. Зависимость по выходу возникает когда команда i и команда j записывают результат в один и тот же регистр или в одну и ту же ячейку памяти. Порядок выполнения этих команд должен сохраняться. Зависимости по выходу сохраняются путем обнаружения конфликтов типа WAW.

Как антизависимости, так и зависимости по выходу являются зависимостями по именам, в отличие от истинных зависимостей по данным, поскольку в них отсутствует передача данных от одной команды к другой. Это означает, что команды, связанные зависимостью по именам, могут выполняться одновременно или могут быть переупорядочены, если имя (номер регистра или адрес ячейки памяти), используемое в командах изменяется так, что команды не конфликтуют. Это переименование может быть выполнено более просто для регистровых операндов и называется переименованием регистров (register renaming). Переименование регистров может выполняться либо статически компилятором, или динамически аппаратными средствами.

В качестве примера рассмотрим следующую последовательность команд:

```
ADD R1,R2,R3
SUB R2,R3,R4
AND R5,R1,R2
OR R1,R3,R4
```

В этой последовательности имеется антизависимость по регистру R2 между командами ADD и SUB, которая может привести к конфликту типа WAR. Ее можно устранить путем переименования регистра результата команды SUB, например, на R6 и изменения всех последующих команд, которые используют результат команды вычитания, для использования этого регистра R6 (в данном случае это только последний операнд в команде AND). Использование R1 в команде OR приводит как к зависимости по выходу с командой ADD, так и к антизависимости между командами ADD и AND. Обе зависимости могут быть устранены путем замены регистра результата либо команды ADD, либо команды OR. В первом случае должна измениться каждая команда, которая использует результат команды ADD прежде чем команда OR запишет в регистр R1 (а именно, второй операнд команды AND в данном примере). Во втором случае при замене регистра результата команды OR, все последующие команды, использующие ее ре-

зультат, должны также измениться. Альтернативой переименованию в процессе компиляции является аппаратное переименование регистров, которое может быть использовано в ситуациях, когда возникают условные переходы, которые возможно сложны или невозможны для анализа компилятором; в следующем разделе эта методика обсуждается более подробно.

Последним типом зависимостей являются зависимости по управлению. Зависимости по управлению определяют порядок команд по отношению к команде условного перехода так, что команды, не являющиеся командами перехода, выполняются только когда они должны выполняться. Каждая команда в программе является зависимой по управлению от некоторого набора условных переходов и, в общем случае, эти зависимости по управлению должны сохраняться. Одним из наиболее простых примеров зависимости по управлению является зависимость операторов, находящихся в части «then» оператора условного перехода if. Например, в последовательности кода:

```
if p1 {
S1;
};
if p2 {
S2;
}
```

S1 является зависимым по управлению от p1, а S2 зависит по управлению от p2 и не зависит от p1.

Имеются два ограничения, связанные с зависимостями по управлению:

1. Команда, которая зависит по управлению от условного перехода, не может быть в результате перемещения поставлена перед командой условного перехода так, что ее выполнение более не управлялось бы этим условным переходом. Например, мы не можем взять команду из части «then» оператора if и поставить ее перед оператором if.

2. Команда, которая не является зависимой по управлению от команды условного перехода, не может быть поставлена после команды условного перехода так, что ее выполнение станет управляться этим условным переходом. Например, мы не можем взять оператор, стоящий перед оператором if и перенести его в часть «then» условного оператора.

Следующий пример иллюстрирует эти два ограничения:

```
ADD R1,R2,R3
BEQZ R12,skipnext
SUB R4,R5,R6
skipnext: OR R7,R1,R9
MULT R13,R1,R4
```

В этой последовательности команд имеются следующие зависимости по управлению (предполагается, что переходы не задерживаются). Команда SUB зависит по управлению от команды BEQZ, поскольку изменение порядка следования этих команд изменит и результат вычислений. Если поставить команду SUB перед командой условного перехода, результат команды MULT не будет тем же самым, что и в случае, когда условный переход является выполняемым. Аналогично, команда ADD не может быть поставлена после команды условного перехода, поскольку это приведет к изменению результата команды MULT, когда переход является выполняемым. Команда OR не является зависимой по управлению от условного перехода, поскольку она выполняется независимо от того, является ли переход выполняемым или нет. Поскольку команда OR не зависит по данным от предшествующих команд и не зависит по управлению от команды условного перехода, она может быть поставлена перед командой условного перехода, что не приведет к изменению значения, вычисляемого этой последовательностью команд. Конечно это предполагает, что команда OR не вызывает никаких побочных эффектов (например, возникновения исключительной ситуации). Если мы хотим переупорядочить команды с подобными потенциальными побочными эффектами, требуется дополнительный анализ компилятором или дополнительная аппаратная поддержка.

Обычно зависимости по управлению сохраняются посредством двух свойств простых конвейеров, подобных рассмотренным в предыдущей главе. Во-первых, команды выполняются в порядке, предписанном программой. Это гарантирует, что команда, стоящая перед командой условного перехода, выполняется перед переходом; таким образом, команда ADD в выше приведенной последовательности будет выполняться перед условным переходом. Во-вторых, средства обнаружения конфликтов по управлению или конфликтов условных переходов гарантируют, что команда, зависимая по управлению от условного перехода, не будет выполняться до тех пор, пока не известно направление условного перехода. В частности команда SUB не будет выполняться до тех пор, пока машина не определит, что условный переход является невыполняемым.

Хотя сохранение зависимостей по управлению является полезным и простым способом обеспечения корректности программы, сама по себе зависимость по управлению не является фундаментальным ограничением производительности. Возможно мы были бы рады выполнять команды, которые не должны выполняться, тем самым нарушая зависимости по управлению, если бы могли это делать не нарушая корректность программы. Зависимость по управлению не является критическим свойством, которое должно сохраняться. В действительности, двумя свойствами, которые являются критичными с точки зрения корректности программы и которые обычно сохра-

няются посредством зависимостей по управлению, являются поведение исключительных ситуаций (exception behavior) и поток данных (data flow).

Сохранение поведения исключительных ситуаций означает, что любые изменения в порядке выполнения команд не должны менять условия возникновения исключительных ситуаций в программе. Часто это требование можно смягчить: переупорядочивание выполнения команд не должно приводить к появлению в программе новых исключительных ситуаций. Простой пример показывает как поддержка зависимостей по управлению может сохранить эти ситуации. Рассмотрим кодовую последовательность:

```
BEQZ R2,L1
LW R1,0(R2)
L1:
```

В данном случае, если мы игнорируем зависимость по управлению и ставим команду загрузки перед командой условного перехода, команда загрузки может вызвать исключительную ситуацию по защите памяти. Заметим, что здесь зависимость по данным, которая препятствует перестановке команд BEQZ и LW, отсутствует, это только зависимость по управлению. Подобная ситуация может возникнуть и при выполнении операции с ПТ, которая может вызвать исключительную ситуацию. В любом случае, если переход выполняется, то исключительная ситуация не возникнет, если команда не ставится выше команды условного перехода. Чтобы разрешить переупорядочивание команд, мы хотели бы как раз игнорировать исключительную ситуацию, если переход не выполняется.

Вторым свойством, сохраняемым с помощью поддержки зависимостей по управлению, является поток данных. Поток данных представляет собой действительный поток данных между командами, которые вырабатывают результаты, и командами, которые эти результаты используют. Условные переходы делают поток данных динамическим, поскольку они позволяют данным для конкретной команды поступать из многих точек (источников). Рассмотрим следующую последовательность команд:

```
ADD R1,R2,R3
BEQZ R4,L
SUB R1,R5,R6
L: OR R7,R1,R8
```

В этом примере значение R1, используемое командой OR, зависит от того, выполняется или не выполняется условный переход. Одной зависимости по данным не достаточно для сохранения корректности программы, поскольку она имеет дело только со статическим порядком чтения и записи. Таким образом, хотя команда OR зависит по данным как от команды ADD, так и от команды SUB, этого недостаточно для корректного выпол-

нения. Когда выполняются команды, должен сохраняться поток данных: если переход не выполняется, то команда OR должна использовать значение R1, вычисленное командой SUB, а если переход выполняется – значение R1, вычисленное командой ADD. Перестановка команды SUB на место перед командой условного перехода не меняет статической зависимости, но она определенно повлияет на поток данных и таким образом приведет к некорректному выполнению. При сохранении зависимости по управлению команды SUB от условного перехода, мы предотвращаем незаконное изменение потока данных. Выполнение команд по предположению и условные команды, которые помогают решить проблему исключительных ситуаций, позволяют также изменить зависимость по управлению, поддерживая при этом правильный поток данных.

Иногда мы можем определить, что устранение зависимости по управлению, не может повлиять на поведение исключительных ситуаций, либо на поток данных. Рассмотрим слегка модифицированную последовательность команд:

```
ADD R1,R2,R3
BEQZ R12,skipnext
SUB R4,R5,R6
ADD R5,R4,R9
skipnext: OR R7,R8,R9
```

Предположим, что мы знаем, что регистр результата команды SUB (R4) не используется после команды, помеченной меткой skipnext. (Свойство, определяющее, будет ли значение использоваться последующими командами, называется живучестью (liveness) и мы вскоре определим его более формально). Если бы регистр R4 не использовался, то изменение значения R4 прямо перед выполнением условного перехода не повлияло бы на поток данных. Таким образом, если бы регистр R4 не использовался и команда SUB не могла выработать исключительную ситуацию, мы могли бы поместить команду SUB на место перед командой условного перехода, поскольку на результат программы это изменение не влияет. Если переход выполняется, команда SUB выполнится и будет бесполезна, но она не повлияет на результат программы. Этот тип планирования кода иногда называется планированием по предположению (speculation), поскольку компилятор в основном делает ставку на исход условного перехода; в данном случае предполагается, что условный переход обычно является невыполняемым. Механизмы задержанных переходов, которые мы рассматривали в предыдущей главе, могут использоваться для уменьшения простоев, возникающих по вине условных переходов, и иногда позволяют использовать планирование по предположению для оптимизации задержек переходов.

Зависимости по управлению сохраняются путем реализации схемы обнаружения конфликта по управлению, которая приводит к приостановке конвейера по управлению. Приостановки по управлению могут устраняться или уменьшаться множеством аппаратных и программных методов. Например, задержанные переходы могут уменьшать приостановки, возникающие в результате конфликтов по управлению. Другие методы уменьшения приостановок, вызванных конфликтами по управлению, включают разворачивание циклов, преобразование условных переходов в условно выполняемые команды и планирование по предположению, выполняемое с помощью компилятора или аппаратуры.

Устранение зависимостей по данным и механизмы динамического планирования

Основная идея динамической оптимизации. Главным ограничением методов конвейерной обработки, которые мы рассматривали ранее, является выдача для выполнения команд строго в порядке, предписанном программой: если выполнение какой-либо команды в конвейере приостанавливалось, следующие за ней команды также приостанавливались. Таким образом, при наличии зависимости между двумя близко рас положенными в конвейере командами возникала приостановка обработки многих команд. Но если имеется несколько функциональных устройств, многие из них могут оказаться незагруженными. Если команда j зависит от длинной команды i, выполняющейся в конвейере, то все команды, следующие за командой j должны приостановиться до тех пор, пока команда i не завершится и не начнет выполняться команда j. Например, рассмотрим следующую последовательность команд:

```
DIVD F0,F2,F4
ADDD F10,F0,F8
SUBD F8,F8,F14
```

Команда SUBD не может выполняться из-за того, что зависимость между командами DIVD и ADDD привела к приостановке конвейера. Однако команда SUBD не имеет никаких зависимостей от команд в конвейере. Это ограничение производительности, которое может быть устранено снятием требования о выполнении команд в строгом порядке.

В рассмотренном нами конвейере структурные конфликты и конфликты по данным проверялись во время стадии декодирования команды (ID). Если команда могла нормально выполняться, она выдавалась с этой ступени конвейера в следующие. Чтобы позволить начать выполнение команды SUBD из предыдущего примера, необходимо разделить процесс выдачи на две части: проверку наличия структурных конфликтов и ожида-

ние отсутствия конфликта по данным. Когда мы выдаем команду для выполнения, мы можем осуществлять проверку наличия структурных конфликтов; таким образом, мы все еще используем упорядоченную выдачу команд. Однако мы хотим начать выполнение команды как только станут доступными ее операнды. Таким образом, конвейер будет осуществлять неупорядоченное выполнение команд, которое означает и неупорядоченное завершение команд.

Неупорядоченное завершение команд создает основные трудности при обработке исключительных ситуаций. В рассматриваемых в данном разделе машинах с динамическим планированием потока команд прерывания будут неточными, поскольку команды могут завершиться до того, как выполнение более ранней выданной команды вызовет исключительную ситуацию. Таким образом, очень трудно повторить запуск после прерывания. Вместо того, чтобы рассматривать эти проблемы в данном разделе, мы обсудим возможные решения для реализации точных прерываний позже в контексте машин, использующих планирование по предположению.

Чтобы реализовать неупорядоченное выполнение команд, мы расщепляем ступень ID на две ступени:

1. Выдача – декодирование команд, проверка структурных конфликтов.
2. Чтение операндов – ожидание отсутствия конфликтов по данным и последующее чтение операндов.

Затем, как и в рассмотренном нами конвейере, следует ступень EX. Поскольку выполнение команд ПТ может потребовать нескольких тактов в зависимости от типа операции, мы должны знать, когда команда начинает выполняться и когда заканчивается. Это позволяет нескольким командам выполняться в один и тот же момент времени. В дополнение к этим изменениям структуры конвейера мы изменим и структуру функциональных устройств, варьируя количество устройств, задержку операций и степень конвейеризации функциональных устройств так, чтобы лучше использовать эти методы конвейеризации.

Динамическая оптимизация с централизованной схемой обнаружения конфликтов. В конвейере с динамическим планированием выполнения команд все команды проходят через ступень выдачи строго в порядке, предписанном программой (упорядоченная выдача). Однако они могут приостанавливаться и обходить друг друга на второй ступени (ступени чтения операндов) и тем самым поступать на ступени выполнения нерпорядочено. Централизованная схема обнаружения конфликтов представляет собой метод, допускающий неупорядоченное выполнение команд при наличии достаточных ресурсов и отсутствии зависимостей по данным. Впервые подобная схема была применена в компьютере CDC 6600.

Прежде чем начать обсуждение возможности применения подобных схем, важно заметить, что конфликты типа WAR, отсутствующие в простых конвейерах, могут появиться при неупорядоченном выполнении команд. В ранее приведенном примере регистром результата для команды SUBD является регистр R8, который одновременно является источником операнда для команды ADDD. Поэтому здесь между командами ADDD и SUBD имеет место антизависимость: если конвейер выполнит команду SUBD раньше команды ADDD, он нарушит эту антизависимость. Этот конфликт WAR можно обойти, если выполнить два правила: (1) читать регистры только во время стадии чтения операндов и (2) поставить в очередь операцию ADDD вместе с копией ее операндов. Чтобы избежать нарушений зависимостей по выходу, конфликты типа WAW (например, это могло произойти, если бы регистром результата команды SUBD была бы регистр F10) все еще должны обнаруживаться. Конфликты типа WAW могут быть устранены с помощью приостановки выдачи команды, регистр результата которой совпадает с уже используемым в конвейере.

Задачей централизованной схемы обнаружения конфликтов является поддержание выполнения команд со скоростью одна команда за такт (при отсутствии структурных конфликтов) посредством как можно более раннего начала выполнения команд. Таким образом, когда команда в начале очереди приостанавливается, другие команды могут выдаваться и выполняться, если они не зависят от уже выполняющейся или приостановленной команды. Централизованная схема несет полную ответственность за выдачу и выполнение команд, включая обнаружение конфликтов. Подобное неупорядоченное выполнение команд требует одновременного нахождения нескольких команд на стадии выполнения. Этого можно достигнуть двумя способами: реализацией в процессоре либо множества неконвейерных функциональных устройств, либо путем конвейеризации всех функциональных устройств. Обе эти возможности по сути эквивалентны с точки зрения организации управления. Поэтому предположим, что в машине имеется несколько неконвейерных функциональных устройств.

Машина CDC 6600 имела 16 отдельных функциональных устройств (4 устройства для операций с плавающей точкой, 5 устройств для организации обращений к основной памяти и 7 устройств для целочисленных операций). В нашем случае централизованная схема обнаружения конфликтов имеет смысл только для устройства плавающей точки. Предположим, что имеются два умножителя, один сложитель, одно устройство деления и одно целочисленное устройство для всех операций обращения к памяти, переходов и целочисленных операций. Хотя устройств в этом примере гораздо меньше, чем в CDC 6600, он достаточно мощный для де-

монстрации основных принципов работы. Поскольку как наша машина, так и CDC 6600 являются машинами с операциями регистр-регистр (операциями загрузки/записи), в обеих машинах методика практически одинаковая. На рисунке 43 показана подобная машина.

Каждая команда проходит через централизованную схему обнаружения конфликтов, которая определяет зависимости по данным; этот шаг соответствует стадии выдачи команд и заменяет часть стадии ID в нашем конвейере. Эти зависимости определяют затем моменты времени, когда команда может читать свои операнды и начинать выполнение операции. Если централизованная схема решает, что команда не может немедленно выполняться, она следит за всеми изменениями в аппаратуре и решает, когда команда сможет выполняться. Эта же централизованная схема определяет также когда команда может записать результат в свой регистр результата. Таким образом, все схемы обнаружения и разрешения конфликтов здесь выполняются устройством центрального управления.

Каждая команда проходит четыре стадии своего выполнения. (Поскольку в данный момент мы интересуемся операциями плавающей точки, мы не рассматриваем стадию обращения к памяти). Рассмотрим эти стадии сначала неформально, а затем детально рассмотрим, как централизованная схема поддерживает необходимую информацию, которая определяет обработку при переходе с одной стадии на другую.

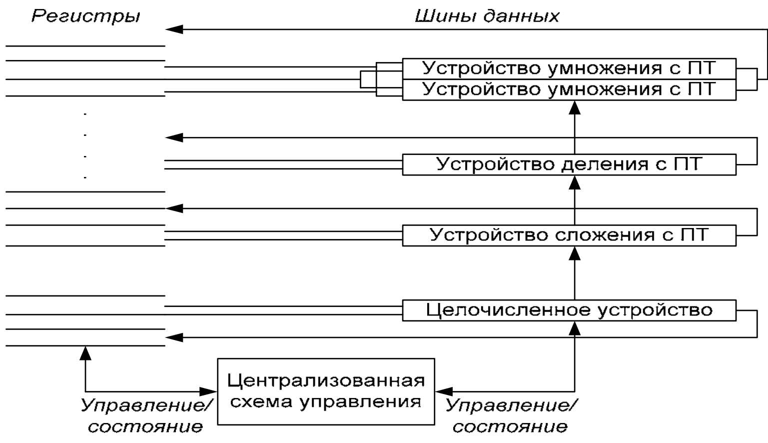


Рис 43. Централизованная схема управления

Следующие четыре стадии заменяют стадии ID, EX и WB в стандартном конвейере:

1. *Выдача*. Если функциональное устройство, необходимое для выполнения команды, свободно и никакая другая выполняющаяся команда не использует тот же самый регистр результата, централизованная схема выдает команду в функциональное устройство и обновляет свою внутреннюю

структуру данных. Поскольку никакое другое работающее функциональное устройство не может записать результат в регистр результата нашей команды, мы гарантируем, что конфликты типа WAW не могут появляться. Если существует структурный конфликт или конфликт типа WAW, выдача команды блокируется и никакие следующие команды не будут выдаваться на выполнение до тех пор, пока эти конфликты существуют. Эта стадия заменяет часть стадии ID в нашем конвейере.

2. *Чтение операндов*. Централизованная схема следит за возможностью выборки источников операндов для соответствующей команды. Операнд-источник доступен, если отсутствует выполняющаяся команда, которая записывает результат в этот регистр или если в данный момент времени в регистр, содержащий операнд, выполняется запись из работающего функционального устройства. Если операнды-источники доступны, централизованная схема сообщает функциональному устройству о необходимости чтения операндов из регистров и начале выполнения операции. Централизованная схема разрешает конфликты RAW на этой стадии динамически, и команды могут посылаться для выполнения не в порядке, предписанном программой. Эта стадия, совместно со стадией выдачи, завершает работу стадии ID простого конвейера.

3. *Выполнение*. Функциональное устройство начинает выполнение операции после получения операндов. Когда результат готов оно уведомляет централизованную схему управления о том, что оно завершило выполнение операции. Эта стадия заменяет стадию EX и занимает несколько тактов в рассмотренном ранее конвейере.

4. *Запись результата*. Когда централизованная схема управления узнает о том, что функциональное устройство завершило выполнение операции, она проверяет существование конфликта типа WAR. Если этот конфликт типа WAR не существует, централизованная схема управления сообщает функциональному устройству о необходимости записи результата в регистр назначения. Эта стадия заменяет стадию WB в простом конвейере.

Основываясь на своей собственной структуре данных, централизованная схема управления управляет продвижением команды с одной ступени на другую, взаимодействуя с функциональными устройствами. Но имеется небольшое усложнение: в регистровом файле имеется только ограниченное число магистралей для операндов-источников и магистралей для записи результата. Централизованная схема управления должна гарантировать, что количество функциональных устройств, которым разрешено продолжать работу на ступенях 2 и 4 не превышает числа доступных шин. Не будем вдаваться в дальнейшие подробности и упомянем лишь, что CDC 6600 решала эту проблему путем объединения 16 функциональных устройств

друг с другом в четыре группы и поддержки для каждой группы устройств набора шин, называемых магистралями данных (data trunks). Только одно устройство в группе могло читать операнды или записывать свой результат в течение одного такта.

Общая структура регистров состояния устройства централизованного управления состоит из 3-х частей:

1) *состояние команды* – показывает каждый из четырех этапов выполнения команды;

2) *состояние функциональных устройств* – имеются 9 полей, описывающих состояние каждого функционального устройства:

a. Занятость – показывает, занято устройство или свободно.

b. Op – выполняемая в устройстве операция.

c. Fi – регистр результата.

d. Fj, Fk – регистры-источники операндов.

e. Qj, Qk – функциональные устройства, вырабатывающие результат для записи в регистры.

f. Rj, Rk – признаки готовности операндов в регистрах Fj, Fk.

3) *состояние регистров резу*льтата – показывает функциональное устройство, которое будет записывать в каждый из регистров. Это поле устанавливается в ноль, если отсутствуют команды, записывающие результат в данный регистр.

Другой подход к динамическому планированию – алгоритм Томасуло. Другой подход к параллельному выполнению команд при наличии конфликтов был использован в устройстве плавающей точки в машине IBM 360/91. Эта схема приписывается Р. Томасуло и названа его именем. Разработка IBM 360/91 была завершена спустя три года после выпуска CDC 6600, прежде чем кэш-память появилась в коммерческих машинах. Задачей IBM было достижение высокой производительности на операциях с плавающей точкой, используя набор команд и компиляторы, разработанные для всего семейства 360, а не только для приложений с интенсивным использованием плавающей точки. Архитектура 360 предусматривала только четыре регистра плавающей точки двойной точности, что ограничивало эффективность планирования кода компилятором. Этот факт был другой мотивацией подхода Томасуло. Наконец, машина IBM 360/91 имела большое время обращения к памяти и большие задержки выполнения операций плавающей точки, преодолеть которые и был призван разработанный Томасуло алгоритм. В конце раздела мы увидим, что алгоритм Томасуло может также поддерживать совмещенное выполнение нескольких итераций цикла.

Мы поясним этот алгоритм на примере устройства ПТ. Основное различие между нашим конвейером ПТ и конвейером машины IBM/360 за-

ключается в наличии в последней машине команд типа регистр-память. Поскольку алгоритм Томасуло использует функциональное устройство загрузки, не требуется значительных изменений, чтобы добавить режимы адресации регистр-память; основное добавление – другая шина. IBM 360/91 имела также конвейерные функциональные устройства, а не несколько функциональных устройств. Единственное отличие между ними заключается в том, что конвейерное функциональное устройство может начинать выполнение только одной операции в каждом такте. Поскольку реально отсутствуют фундаментальные отличия, мы описываем алгоритм, как если бы имели место несколько функциональных устройств. IBM 360/91 могла выполнять одновременно три операции сложения ПТ и две операции умножения ПТ. Кроме того, в процессе выполнения могли находиться до 6 операций загрузки ПТ, или обращений к памяти, и до трех операций записи ПТ. Для реализации этих функций использовались буфера данных загрузки и буфера данных записи. Хотя мы не будем обсуждать устройства загрузки и записи, необходимо добавить буфера для операндов.

Схема Томасуло имеет много общего со схемой централизованного управления CDC 6600, однако имеются и существенные отличия. Во-первых, обнаружение конфликтов и управление выполнением являются распределенными – станции резервирования (reservation stations) в каждом функциональном устройстве определяют, когда команда может начать выполняться в данном функциональном устройстве. В CDC 6600 эта функция централизована. Во-вторых, результаты операций посылаются прямо в функциональные устройства, а не проходят через регистры. В IBM 360/91 имеется общая шина результатов операций (которая называется общей шиной данных (common data bus – CDB)), которая позволяет производить одновременную загрузку всех устройств, ожидающих операнда. CDC 6600 записывает результаты в регистры, за которые ожидающие функциональные устройства могут соперничать. Кроме того, CDC 6600 имеет несколько шин завершения операций (две в устройстве ПТ), а IBM 360/91 – только одну.

На рисунке 44 представлена основная структура устройства ПТ на базе алгоритма Томасуло. Станции резервирования хранят команды, которые выданы и ожидают выполнения в соответствующем функциональном устройстве, а также информацию, требующуюся для управления командой, когда ее выполнение началось в функциональном устройстве. Буфера загрузки и записи хранят данные поступающие из памяти и записываемые в память. Регистры ПТ соединены с функциональными устройствами парой шин и одной шиной с буферами записи. Все результаты из функциональных устройств и из памяти посылаются на общую шину данных, которая

связана со входами всех устройств за исключением буфера загрузки. Все буфера и станции резервирования имеют поля тегов, используемых для управления конфликтами.

Прежде чем описывать детали станций резервирования и алгоритм, рассмотрим все стадии выполнения команды. В отличие от централизованной схемы управления, имеется всего три стадии:

1. *Выдача*. Берет команду из очереди команд ПТ. Если операция является операцией ПТ, выдает ее при наличии свободной станции резервирования и посылает операнды на станцию резервирования, если они находятся в регистрах. Если операция является операцией загрузки или записи, она может выдаваться при наличии свободного буфера. При отсутствии свободной станции резервирования или свободного буфера возникает структурный конфликт и команда приостанавливается до тех пор, пока не освободится станция резервирования или буфер.

2. *Выполнение*. Если один или более операндов команды не доступны по каким либо причинам, контролируется состояние CDB и ожидается завершение вычисления значений нужного регистра. На этой стадии выполняется контроль конфликтов типа RAW. Когда оба операнда доступны, выполняется операция.

3. *Запись результата*. Когда становится доступным результат, он записывается на CDB и оттуда в регистры и любое функциональное устройство, ожидающее этот результат.

Хотя эти шаги в основном похожи на аналогичные шаги в централизованной схеме управления, имеются три важных отличия. Во-первых, отсутствует контроль конфликтов типа WAW и WAR – они устраняются как побочный эффект алгоритма. Во-вторых, для трансляции результатов используется CDB, а не схема ожидания готовности регистров. В-третьих, устройства загрузки и записи рассматриваются как основные функциональные устройства.

Структуры данных, используемые для обнаружения и устранения конфликтов, связаны со станциями резервирования, регистровым файлом и буферами загрузки и записи. Хотя с разными объектами связана разная информация, все устройства, за исключением буферов загрузки, содержат в каждой строке поле тега. Это поле тега представляет собой четырехбитовое значение, которое обозначает одну из пяти станций резервирования или один из шести буферов загрузки. Поле тега используется для описания того, какое функциональное устройства будет поставлять результат, нужный в качестве источника операнда. Неиспользуемые значения, такие как ноль, показывают, что операнд уже доступен. Важно помнить, что теги в схеме Томасуло ссылаются на буфера или устройства, которые будут по-

ставлять результат; когда команда выдается в станцию резервирования номера регистров исключаются из рассмотрения.

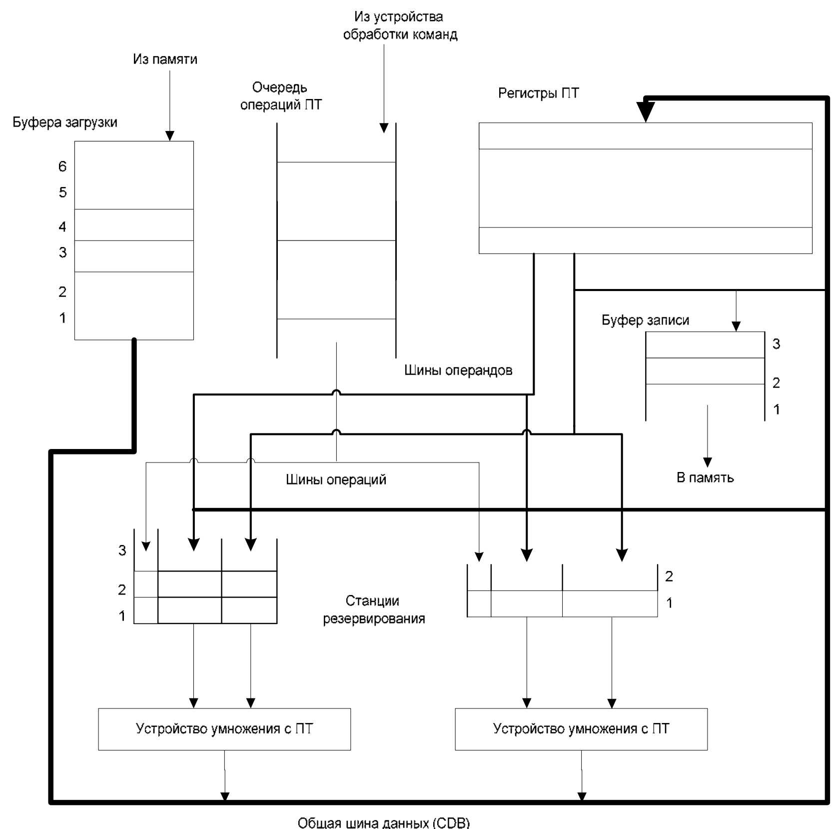


Рис. 44. Структура устройства вычислений с ПТ на основе алгоритма Томасуло

Каждая станция резервирования содержит шесть полей:

1) Op – операция, которая должна выполняться над источниками операндов S1 и S2;

2) Qj,Qk – станции резервирования, которые будут вырабатывать соответствующий операнд-источник; нулевое значение показывает, что операнд-источник уже доступен в Vj или Vk, или не является обязательным;

3) Vj,Vk – значение операндов-источников. Заметим, что для каждого операнда являются действительными только одно из полей либо поле V, либо поле Q;

4) Занято. Показывает, что данная станция резервирования и ее соответствующее функциональное устройство заняты.

Регистровый файл и буфер записи имеют поле Qi, где Qi – номер функционального устройства, которое будет вырабатывать значение, которое надо записать в регистр или память. Если значение Qi равно нулю, то это означает, что ни одна текущая активная команда не вычисляет результат для данного регистра или буфера. Для регистра это означает, что значение определяется содержимым регистра.

В каждом из буферов загрузки и записи требуется поле занятости, показывающее, когда соответствующий буфер становится доступным благодаря завершению загрузки или записи, назначенных на этот буфер. Буфер записи имеет также поле V для хранения значения, которое должно быть записано в память.

Проанализируем работу алгоритма для следующей последовательности команд:

1. LF F6,34(R2)
2. LF F2,45(R3)
3. MULTD F0,F2,F4
4. SUBD F8,F6,F2
5. DIVD F10,F0,F6
6. ADDD F6,F8,F2

Имеются два важных отличия от централизованной схемы управления. Во-первых, значение операнда хранится в станции резервирования в одном из полей V как только оно становится доступным; оно не считывается из регистрового файла во время выдачи команды. Во-вторых, за счет первого отличия сокращается число структурных конфликтов по сравнению с централизованной схемой управления.

Большие преимущества схемы Томасуло заключаются в распределении логики обнаружения конфликтов и устранении приостановок, связанных с конфликтами типа WAW и WAR. Первое преимущество возникает из-за наличия распределенных станций резервирования и использования CDB. Если несколько команд ожидают один и тот же результат и каждая команда уже имеет свой другой операнд, то команды могут выдаваться одновременно посредством трансляции по CDB. В централизованной схеме управления ожидающие команды должны читать свои операнды из регистров когда станут доступными регистровые шины.

Конфликты типа WAW и WAR устраняются путем переименования регистров используя станции резервирования. Например, в нашей кодовой последовательности мы выдали на выполнение как команду DIVD, так и команду ADDD, даже хотя имелся конфликт типа WAR по регистру F6. Конфликт устраняется одним из двух способов. Если команда, поставляющая значение для команды DIVD, завершилась, тогда Vk будет хранить результат, позволяя DIVD выполняться независимо от команды ADDD. С другой стороны, если выполнение команды LF не завершилось, то Qk будет указывать на LOAD1 и команда DIVD будет независимой от ADDD. Таким образом, в любом случае команда ADDD может быть выдана и начать выполняться. Любое использование результата команды MULTD будет указывать на станцию резервирования, позволяя ADDD завершить и записать свое значение в регистры без воздействия DIVD.

Чтобы понять полную мощность устранения конфликтов типа WAW и WAR посредством динамического переименования регистров мы должны рассмотреть цикл. Рассмотрим следующую простую последовательность команд для умножения элементов вектора на скалярную величину, находящуюся в регистре F2:

```
Loop: LD F0,0(R1)
MULTD F4,F0,F2
SD 0(R1),F4
SUBI R1,R1,#8
BNEZ R1,Loop; условный переход при R1 /=0
```

Со стратегией выполняемого перехода использование станций резервирования позволит сразу же продолжить выполнение нескольких итераций этого цикла. Это преимущество дается без статического разворачивания цикла программными средствами: в действительности цикл разворачивается динамически аппаратурой. В архитектуре 360 наличие всего 4 регистров ПТ сильно ограничивало бы использование статического разворачивания цикла. (Вскоре мы увидим, что при статическом разворачивании и планировании выполнения цикла для обхода взаимных блокировок требуется значительно большее число регистров). Алгоритм Томасуло поддерживает выполнение с перекрытием нескольких копий одного и того же цикла при наличии лишь небольшого числа регистров, используемых программой.

Предположим, что мы выдали для выполнения все команды двух последовательных итераций цикла, но еще не завершилось выполнение ни одной операции загрузки/записи в память. Проигнорируем операции целочисленного АЛУ и предположим, что условный переход был спрогнозирован как выполняемый. Когда система достигла такого состояния могут поддерживаться две копии цикла с CPI близким к единице при условии, что операции умножения могут завершиться за четыре такта. Если мы игнорируем накладные расходы цикла, которые не снижены в этой схеме, достигнутый уровень производительности будет соответствовать тому, который мы могли бы достигнуть посредством статического разворачивания и планирования цикла компилятором при условии наличия достаточного числа регистров.

В этом примере демонстрируется дополнительный элемент, который является важным для того, чтобы алгоритм Томасуло работал. Команда загрузки из второй итерации цикла может легко закончиться раньше команды записи из первой итерации, хотя нормальный последовательный порядок отличается. Загрузка и запись могут надежно (безопасно) выполняться в различном порядке при условии, что загрузка и запись обращаются к разным адресам. Это контролируется путем проверки адресов в буфере записи каждый раз при выдаче команды загрузки. Если адрес команды загрузки соответствует одному из адресов в буфере записи мы должны оста-

новиться и подождать до тех пор, пока буфер записи не получит требуемое значение; затем мы можем к нему обращаться или выбирать значение из памяти. Это динамическое сравнение адресов является альтернативой технике, которая может использоваться компилятором при перестановке команд загрузки и записи.

Динамическая схема Томасуло может достигать очень высокой производительности при условии того, что стоимость переходов может поддерживаться небольшой. Главный недостаток этого подхода заключается в сложности схемы Томасуло, которая требует для своей реализации очень большого объема аппаратуры. Особенно это касается большого числа устройств ассоциативной памяти, которая должна работать с высокой скоростью, а также сложной логики управления. Наконец, увеличение производительности ограничивается наличием одной шины завершения (CDB). Хотя дополнительные шины CDB могут быть добавлены, каждая CDB должна взаимодействовать со всей аппаратурой конвейера, включая станции резервирования. В частности, аппаратуру ассоциативного сравнения необходимо дублировать на каждой станции для каждой CDB.

В схеме Томасуло комбинируются две различных методики: методика переименования регистров и буферизация операндов-источников из регистрового файла. Буферизация источников операндов разрешает конфликты типа WAR, которые возникают, когда операнды доступны в регистрах. Как мы увидим позже, возможно также устранять конфликты типа WAR посредством переименования регистра вместе с буферизацией результата до тех пор, пока остаются обращения к старой версии регистра; этот подход будет использоваться, когда мы будем обсуждать аппаратное выполнение по предположению.

Схема Томасуло является привлекательной, если разработчик вынужден делать конвейерную архитектуру, для которой трудно выполнить планирование кода или реализовать большое хранилище регистров. С другой стороны, преимущество подхода Томасуло возможно ощущается меньше, чем увеличение стоимости реализации, по сравнению с методами планирования загрузки конвейера средствами компилятора в машинах, ориентированных на выдачу для выполнения только одной команды в такте. Однако по мере того, как машины становятся все более агрессивными в своих возможностях выдачи команд и разработчики сталкиваются с вопросами производительности кода, который трудно планировать (большинство кодов для нечисловых расчетов), методика типа переименования регистров и динамического планирования будет становиться все более важной. Также эти методы являются одним из важных компонентов большинства схем для реализации аппаратного выполнения по предположению.

Ключевыми компонентами увеличения параллелизма уровня команд в алгоритме Томасуло являются динамическое планирование, переименова-

ние регистров и динамическое устранение неоднозначности обращений к памяти. Трудно оценить значение каждого из этих свойств по отдельности.

Динамической аппаратной технике планирования загрузки конвейера при наличии зависимостей по данным соответствует и динамическая техника для эффективной обработки переходов. Эта техника используется для двух целей: для прогнозирования того, будет ли переход выполняемым, и для возможно более раннего нахождения целевой команды перехода. Эта техника называется аппаратным прогнозированием переходов.

8. Аппаратное прогнозирование направления переходов и снижение потерь на организацию переходов. Буфер прогнозирования переходов. Корреляционая схема. Другие методы сокращения приостановок по управлению

Аппаратное прогнозирование направления переходов и снижение потерь на организацию переходов. Буфера прогнозирования условных переходов. Простейшей схемой динамического прогнозирования направления условных переходов является буфер прогнозирования условных переходов (branch-prediction buffer) или таблица «истории» условных переходов (branch history table). Буфер прогнозирования условных переходов представляет собой небольшую память, адресуемую с помощью младших разрядов адреса команды перехода. Каждая ячейка этой памяти содержит один бит, который говорит о том, был ли предыдущий переход выполняемым или нет. Это простейший вид такого рода буфера. В нем отсутствуют теги, и он оказывается полезным только для сокращения задержки перехода в случае, если эта задержка больше, чем время, необходимое для вычисления значения целевого адреса перехода. В действительности мы не знаем, является ли прогноз корректным (этот бит в соответствующую ячейку буфера могла установить совсем другая команда перехода, которая имела то же самое значение младших разрядов адреса). Но это не имеет значения. Прогноз – это только предположение, которое рассматривается как корректное, и выборка команд начинается по прогнозируемому направлению. Если же предположение окажется неверным, бит прогноза инвертируется. Конечно, такой буфер можно рассматривать как кэш-память, каждое обращение к которой является попаданием, и производительность буфера зависит от того, насколько часто прогноз применялся и насколько он оказался точным.

Однако простая однобитовая схема прогноза имеет недостаточную производительность. Рассмотрим, например, команду условного перехода в цикле, которая являлась выполняемым переходом последовательно девять раз подряд, а затем однажды невыполняемым. Направление перехода будет неправильно предсказываться при первой и при последней итерации цикла. Неправильный прогноз последней итерации цикла неизбежен, по-

скольку бит прогноза будет говорить, что переход «выполняемый» (переход был девять раз подряд выполняемым). Неправильный прогноз на первой итерации происходит из-за того, что бит прогноза инвертируется при предыдущем выполнении последней итерации цикла, поскольку в этой итерации переход был невыполняемым. Таким образом, точность прогноза для перехода, который выполнялся в 90% случаев, составила только 80% (2 некорректных прогноза и 8 корректных). В общем случае, для команд условного перехода, используемых для организации циклов, переход является выполняемым много раз подряд, а затем один раз оказывается невыполняемым. Поэтому однобитовая схема прогнозирования будет неправильно предсказывать направление перехода дважды (при первой и при последней итерации).

Для исправления этого положения часто используется схема двухбитового прогноза. В двухбитовой схеме прогноз должен быть сделан неверно дважды, прежде чем он изменится на противоположное значение. На рис. 45 представлена диаграмма состояний двухбитовой схемы прогнозирования, направления перехода.

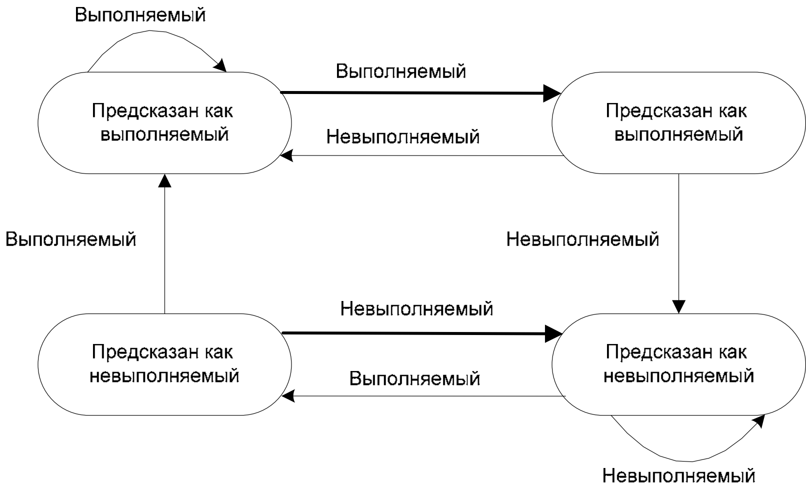


Рис. 45. Диаграмма состояния двухбитовой схемы прогнозирования

Двухбитовая схема прогнозирования в действительности является частным случаем более общей схемы, которая в каждой строке буфера прогнозирования имеет n-битовый счетчик. Этот счетчик может принимать значения от 0 до 2^n - 1. Тогда схема прогноза будет следующей:

1. Если значение счетчика больше или равно 2^n-1 (точка на середине интервала), то переход прогнозируется как выполняемый. Если направление перехода предсказано правильно, к значению счетчика добавляется единица (если только оно не достигло максимальной величины); если прогноз был неверным, из значения счетчика вычитается единица.

2. Если значение счетчика меньше, чем 2^n-1, то переход прогнозируется как невыполняемый. Если направление перехода предсказано правильно, из

значения счетчика вычитается единица (если только не достигнуто значение 0); если прогноз был неверным, к значению счетчика добавляется единица.

Исследования n-битовых схем прогнозирования показали, что двухбитовая схема работает почти также хорошо, и поэтому в большинстве систем применяются двухбитовые схемы прогноза, а не n-битовые.

Буфер прогнозирования переходов может быть реализован в виде небольшой специальной кэш-памяти, доступ к которой осуществляется с помощью адреса команды во время стадии выборки команды в конвейере (IF), или как пара битов, связанных с каждым блоком кэш-памяти команд и выбираемых с каждой командой. Если команда декодируется как команда перехода, и если переход спрогнозирован как выполняемый, выборка команд начинается с целевого адреса как только станет известным новое значение счетчика команд. В противном случае продолжается последовательная выборка и выполнение команд. Если прогноз оказался неверным, значение битов прогноза меняется в соответствии с рис. 46. Хотя эта схема полезна для большинства конвейеров, рассмотренный нами простейший конвейер выясняет примерно за одно и то же время оба вопроса: является ли переход выполняемым и каков целевой адрес перехода. Предполагается отсутствие конфликта при обращении к регистру, определенному в команде условного перехода. Напомним, что для простейшего конвейера это справедливо, поскольку условный переход выполняет сравнение содержимого регистра с нулем во время стадии ID, во время которой вычисляется также и эффективный адрес. Таким образом, эта схема не помогает в случае простых конвейеров, подобных рассмотренному ранее.

Как уже упоминалось, точность двухбитовой схемы прогнозирования зависит от того, насколько часто прогноз каждого перехода является правильным и насколько часто строка в буфере прогнозирования соответствует выполняемой команде перехода. Если строка не соответствует данной команде перехода, прогноз в любом случае делается, поскольку все равно никакая другая информация не доступна. Даже если эта строка соответствует совсем другой команде перехода, прогноз может быть удачным.

Какую точность можно ожидать от буфера прогнозирования переходов на реальных приложениях при использовании 2 бит на каждую строку буфера? Для набора оценочных тестов SPEC-89 буфер прогнозирования переходов с 4096 строками дает точность прогноза от 99% до 82%, т.е. процент неудачных прогнозов составляет от 1% до 18% (см. рис. 46). Следует отметить, что буфер емкостью 4К строк считается очень большим. Буферы меньшего объема дадут худшие результаты.

Однако одного знания точности прогноза не достаточно для того, чтобы определить воздействие переходов на производительность машины,

даже если известны время выполнения перехода и потери при неудачном прогнозе. Необходимо учитывать частоту переходов в программе, поскольку важность правильного прогноза больше в программах с большей частотой переходов. Например, многие целочисленные программы имеют большую частоту переходов, чем значительно более простые для прогнозирования программы плавающей точки.

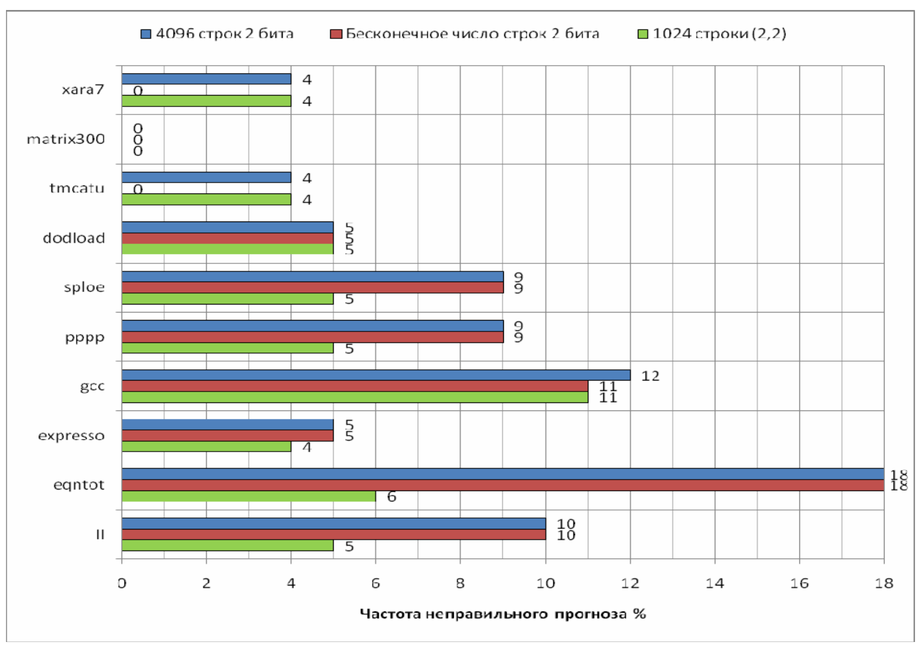


Рис. 46. Сравнение качества 2-битового прогноза

Поскольку главной задачей является использование максимально доступной степени параллелизма программы, точность прогноза направления переходов становится очень важной. Как видно из рис. 46, точность схемы прогнозирования для целочисленных программ, которые обычно имеют более высокую частоту переходов, меньше, чем для научных программ с плавающей точкой, в которых интенсивно используются циклы. Можно решать эту проблему двумя способами: увеличением размера буфера и увеличением точности схемы, которая используется для выполнения каждого отдельного прогноза. Буфер с 4К строками уже достаточно большой и, как показывает рис. 46, работает практически также, что и буфер бесконечного размера. Из этого рисунка становится также ясно, что коэффициент попаданий буфера не является лимитирующим фактором. Как мы упоминали выше, увеличение числа бит в схеме прогноза также имеет малый эффект.

Рассмотренные двухбитовые схемы прогнозирования используют информацию о недавнем поведении команды условного перехода для прогноза будущего поведения этой команды. Вероятно, можно улучшить точ-

ность прогноза, если учитывать не только поведение того перехода, который мы пытаемся предсказать, но рассматривать также и недавнее поведение других команд перехода. Рассмотрим, например, небольшой фрагмент из текста программы eqntott тестового пакета SPEC92 (это наихудший случай для двухбитовой схемы прогноза):

```
if (aa==2)
aa=0;
if (bb==2)
bb=0;
if (aa!=bb) {
```

Ниже приведен текст сгенерированной программы (предполагается, что aa и bb размещены в регистрах R1 и R2):

```
SUBI R3,R1,#2
BNEZ R3,L1 ; переход b1 (aa!=2)
ADD R1,R0,R0 ; aa=0
L1: SUBI R3,R2,#2
BNEZ R3,L2 ; переход b2 (bb!=2)
ADD R2,R0,R0 ; bb=0
L2: SUB R3,R1,R2 ; R3=aa-bb
BEQZ R3,L3 ; branch b3 (aa==bb).
...
L3:
```

Пометим команды перехода как b1, b2 и b3. Можно заметить, что поведение перехода b3 коррелирует с переходами b1 и b2. Ясно, что если оба перехода b1 и b2 являются невыполняемыми (т. е. оба условия if оцениваются как истинные и обеим переменным aa и bb присвоено значение 0), то переход b3 будет выполняемым, поскольку aa и bb очевидно равны. Схема прогнозирования, которая для предсказания направления перехода использует только прошлое поведение того же перехода никогда этого не учтет.

Схемы прогнозирования, которые для предсказания направления перехода используют поведение других команд перехода, называются коррелированными или двухуровневыми схемами прогнозирования. Схема прогнозирования называется прогнозом (1,1), если она использует поведение одного последнего перехода для выбора из пары однобитовых схем прогнозирования на каждый переход. В общем случае схема прогнозирования (m,n) использует поведение последних m переходов для выбора из 2^m схем прогнозирования, каждая из которых представляет собой n-битовую схему прогнозирования для каждого отдельного перехода. Привлекательность такого типа коррелируемых схем прогнозирования переходов заключается в том, что они могут давать больший процент успешного прогнозирования,

чем обычная двухбитовая схема, и требуют очень небольшого объема дополнительной аппаратуры. Простота аппаратной схемы определяется тем, что глобальная история последних m переходов может быть записана в m-битовом сдвиговом регистре, каждый разряд которого запоминает, был ли переход выполняемым или нет. Тогда буфер прогнозирования переходов может индексироваться конкатенацией (объединением) младших разрядов адреса перехода с m-битовой глобальной историей. Например, на рис. 47 показана схема прогнозирования (2,2) и организация выборки битов прогноза.

Рис. 47. Буфер прогнозирования переходов (2,2)

В этой реализации имеется тонкий эффект: поскольку буфер прогнозирования не является кэш-памятью, счетчики, индексируемые единственным значением глобальной схемы прогнозирования, могут в действительности в некоторый момент времени соответствовать разным командам перехода; это не отличается от того, что мы видели и раньше: прогноз может не соответствовать текущему переходу. На рисунке 47 с целью упрощения понимания буфер изображен как двумерный объект. В действительности он может быть реализован просто как линейный массив двухбитовой памяти; индексация выполняется путем конкатенации битов глобальной истории и соответствующим числом бит, требуемых от адреса перехода.

Насколько лучше схемы с корреляцией переходов работают по сравнению со стандартной двухбитовой схемой? Чтобы их справедливо сравнить, нужно сопоставить схемы прогнозирования, использующие одинаковое число бит состояния. Число бит в схеме прогнозирования (m,n) равно

2^m * n * количество строк, выбираемых с помощью адреса перехода. Например, двухбитовая схема прогнозирования без глобальной истории есть просто схема (0,2).

Раньше мы рассматривали схему прогнозирования с 4К строками, выбираемыми адресом перехода. Таким образом, общее количество бит равно: 2^0 * 2 * 4K = 8K.

Схема на рис. 47. имеет 2^2 * 2 * 16 = 128 бит.

Чтобы сравнить производительность схемы коррелированного прогнозирования с простой двухбитовой схемой прогнозирования, производительность которой была представлена на рис. 45, нужно определить количество строк в схеме коррелированного прогнозирования.

Таким образом, мы должны определить количество строк, выбираемых командой перехода в схеме прогнозирования (2,2), которая содержит 8К бит в буфере прогнозирования.

Мы знаем, что

2^2 * 2 * количество строк, выбираемых командой перехода = 8К

Поэтому количество строк, выбираемых командой перехода = 1К.

На рисунке 46 представлены результаты сравнения простой двухбитовой схемы прогнозирования с 4К строками и схемы прогнозирования (2,2) с 1К строками. Как можно видеть, эта последняя схема прогнозирования не только превосходит простую двухбитовую схему прогнозирования с тем же самым количеством бит состояния, но часто превосходит даже двухбитовую схему прогнозирования с неограниченным (бесконечным) количеством строк. Имеется широкий спектр корреляционных схем прогнозирования, среди которых схемы (0,2) и (2,2) являются наиболее интересными.

Дальнейшее уменьшение приостановок по управлению: буфера целевых адресов переходов. Рассмотрим ситуацию, при которой на стадии выборки команд находится команда перехода (на следующей стадии будет осуществляться ее дешифрация). Тогда, чтобы сократить потери, необходимо знать, по какому адресу выбирать следующую команду. Это означает, что надо выяснить, что еще недешифрированная команда в самом деле является командой перехода, и чему равно следующее значение счетчика адресов команд. Если все это мы будем знать, то потери на команду перехода могут быть сведены к нулю. Специальный аппаратный кэш прогнозирования переходов, который хранит прогнозируемый адрес следующей команды, называется буфером целевых адресов переходов (branch-target buffer). Каждая строка этого буфера включает программный адрес команды перехода, прогнозируемый адрес следующей команды и предысторию команды перехода (рис. 48).

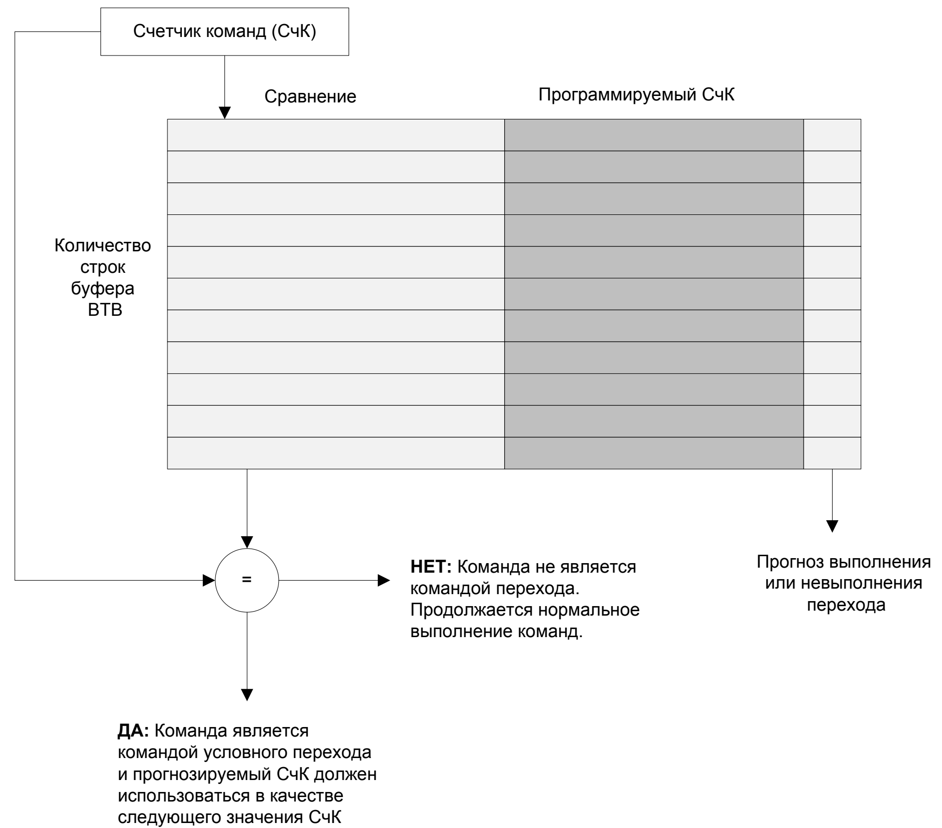


Рис. 48. Буфер целевых адресов переходов

Биты предыстории представляют собой информацию о выполнении или невыполнении условий перехода данной команды в прошлом. Обращение к буферу целевых адресов перехода (сравнение с полями программных адресов команд перехода) производится с помощью текущего значения счетчика команд на этапе выборки очередной команды. Если обнаружено совпадение (попадание в терминах кэш-памяти), то по предыстории команды прогнозируется выполнение или невыполнение условий команды перехода, и немедленно производится выборка и дешифрация команд из прогнозируемой ветви программы. Считается, что предыстория перехода, содержащая информацию о двух предшествующих случаях выполнения этой команды, позволяет прогнозировать развитие событий с вполне достаточной вероятностью.

Существуют и некоторые вариации этого метода. Основной их смысл заключается в том, чтобы хранить в процессоре одну или несколько команд из прогнозируемой ветви перехода. Этот метод может применяться как в совокупности с буфером целевых адресов перехода, так и без него, и имеет два преимущества. Во-первых, он позволяет выполнять обращения к буферу целевых адресов перехода в течение более длительного времени, а не только в течение времени последовательной выборки команд. Это по-

зволяет реализовать буфер большего объема. Во-вторых, буферизация самих целевых команд позволяет использовать дополнительный метод оптимизации, который называется свертыванием переходов (branch folding). Свертывание переходов может использоваться для реализации нулевого времени выполнения самих команд безусловного перехода, а в некоторых случаях и нулевого времени выполнения условных переходов. Рассмотрим буфер целевых адресов перехода, который буферизует команды из прогнозируемой ветви. Пусть к нему выполняется обращение по адресу команды безусловного перехода. Единственной задачей этой команды безусловного перехода является замена текущего значения счетчика команд. В этом случае, когда буфер адресов регистрирует попадание и показывает, что переход безусловный, конвейер просто может заменить команду, которая выбирается из кэш-памяти (это и есть сама команда безусловного перехода), на команду из буфера. В некоторых случаях таким образом удается убрать потери для команд условного перехода, если код условия установлен заранее.

Еще одним методом уменьшения потерь на переходы является метод прогнозирования косвенных переходов, а именно переходов, адрес назначения которых меняется в процессе выполнения программы (в run-time). Компиляторы языков высокого уровня будут генерировать такие переходы для реализации косвенного вызова процедур, операторов select или case и вычисляемых операторов goto в Фортране. Однако подавляющее большинство косвенных переходов возникает в процессе выполнения программы при организации возврата из процедур. Например, для тестовых пакетов SPEC возвраты из процедур в среднем составляют 85% общего числа косвенных переходов.

Хотя возвраты из процедур могут прогнозироваться с помощью буфера целевых адресов переходов, точность такого метода прогнозирования может оказаться низкой, если процедура вызывается из нескольких мест программы или вызовы процедуры из одного места программы не локализуются по времени. Чтобы преодолеть эту проблему, была предложена концепция небольшого буфера адресов возврата, работающего как стек. Эта структура кэширует последние адреса возврата: во время вызова процедуры адрес возврата вталкивается в стек, а во время возврата он оттуда извлекается. Если этот кэш достаточно большой (например, настолько большой, чтобы обеспечить максимальную глубину вложенности вызовов), он будет прекрасно прогнозировать возвраты. На рисунке 49 показано исполнение такого буфера возвратов, содержащего от 1 до 16 строк (элементов) для нескольких тестов SPEC.

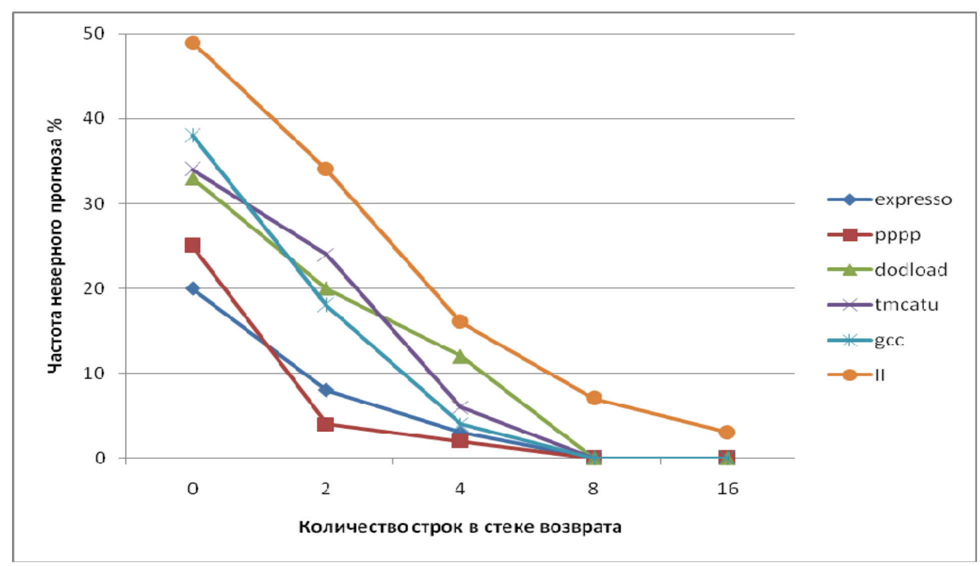


Рис. 49. Точность прогноза для адресов возврата

Точность прогноза в данном случае есть доля адресов возврата, предсказанных правильно. Поскольку глубина вызовов процедур обычно не большая, за некоторыми исключениями даже небольшой буфер работает достаточно хорошо. В среднем возвраты составляют 81% общего числа косвенных переходов для этих шести тестов.

Схемы прогнозирования условных переходов ограничены как точностью прогноза, так и потерями в случае неправильного прогноза. Как мы видели, типичные схемы прогнозирования достигают точности прогноза в диапазоне от 80 до 95% в зависимости от типа программы и размера буфера. Кроме увеличения точности схемы прогнозирования, можно пытаться уменьшить потери при неверном прогнозе. Обычно это делается путем выборки команд по обоим ветвям (по предсказанному и по непредсказанному направлению). Это требует, чтобы система памяти была двухпортовой, включала кэш-память с расслоением, или осуществляла выборку по одному из направлений, а затем по другому (как это делается в IBM POWER-2). Хотя подобная организация увеличивает стоимость системы, возможно, это единственный способ снижения потерь на условные переходы ниже определенного уровня. Другое альтернативное решение, которое используется в некоторых машинах, заключается в кэшировании адресов или команд из нескольких направлений (ветвей) в целевом буфере.

9. Одновременная выдача нескольких команд для выполнения и динамическое планирование

Методы минимизации приостановок работы конвейера из-за наличия в программах логических зависимостей по данным и по управлению были нацелены на достижение идеального CPI (среднего количества тактов на

выполнение команды в конвейере), равного 1. Чтобы еще больше повысить производительность процессора необходимо сделать CPI меньшим, чем 1. Однако этого нельзя добиться, если в одном такте выдается на выполнение только одна команда. Следовательно, необходима параллельная выдача нескольких команд в каждом такте. Существуют два типа подобного рода машин: суперскалярные машины и VLIW-машины (машины с очень длинным командным словом). Суперскалярные машины могут выдавать на выполнение в каждом такте переменное число команд, и работа их конвейеров может планироваться как статически с помощью компилятора, так и с помощью аппаратных средств динамической оптимизации. В отличие от суперскалярных машин, VLIW-машины выдают на выполнение фиксированное количество команд, которые сформатированы либо как одна большая команда, либо как пакет команд фиксированного формата. Планирование работы VLIW-машины всегда осуществляется компилятором.

Суперскалярные машины используют параллелизм на уровне команд путем посылки нескольких команд из обычного потока команд в несколько функциональных устройств. Дополнительно, чтобы снять ограничения последовательного выполнения команд, эти машины используют механизмы внеочередной выдачи и внеочередного завершения команд, прогнозирование переходов, кэши целевых адресов переходов и условное (по предположению) выполнение команд. Возросшая сложность, реализуемая этими механизмами, создает также проблемы реализации точного прерывания.

В типичной суперскалярной машине аппаратура может осуществлять выдачу от одной до восьми команд в одном такте. Обычно эти команды должны быть независимыми и удовлетворять некоторым ограничениям, например таким, что в каждом такте не может выдаваться более одной команды обращения к памяти. Если какая-либо команда в потоке команд является логически зависимой или не удовлетворяет критериям выдачи, на выполнение будут выданы только команды, предшествующие данной. Поэтому скорость выдачи команд в суперскалярных машинах является переменной. Это отличает их от VLIW-машин, в которых полную ответственность за формирование пакета команд, которые могут выдаваться одновременно, несет компилятор, а аппаратура в динамике не принимает никаких решений относительно выдачи нескольких команд.

Предположим, что машина может выдавать на выполнение две команды в одном такте. Одной из таких команд может быть команда загрузки регистров из памяти, записи регистров в память, команда переходов, операции целочисленного АЛУ, а другой может быть любая операция плавающей точки. Параллельная выдача целочисленной операции и операции с плавающей точкой намного проще, чем выдача двух произвольных команд. В реальных системах (например, в микропроцессорах PA7100, hyperSPARC,

Pentium и др.) применяется именно такой подход. В более мощных микропроцессорах (например, MIPS R10000, UltraSPARC, PowerPC 620 и др.) реализована выдача до четырех команд в одном такте.

Выдача двух команд в каждом такте требует одновременной выборки и декодирования по крайней мере 64 бит. Чтобы упростить декодирование можно потребовать, чтобы команды располагались в памяти парами и были выровнены по 64-битовым границам. В противном случае необходимо анализировать команды в процессе выборки и, возможно, менять их местами в момент пересылки в целочисленное устройство и в устройство ПТ. При этом возникают дополнительные требования к схемам обнаружения конфликтов. В любом случае вторая команда может выдаваться, только если может быть выдана на выполнение первая команда. Аппаратура принимает такие решения в динамике, обеспечивая выдачу только первой команды, если условия для одновременной выдачи двух команд не соблюдаются. В таблице 19 представлена диаграмма работы подобного конвейера в идеальном случае, когда в каждом такте на выполнение выдается пара команд.

Таблица 21

Работа суперскалярного конвейера

Тип команды	Ступень конвейера							
Целочисленная команда	IF	ID	EX	MEM	WB			
Команда ПТ	IF	ID	EX	MEM	WB			
Целочисленная команда		IF	ID	EX	MEM	WB		
Команда ПТ		IF	ID	EX	MEM	WB		
Целочисленная команда			IF	ID	EX	MEM	WB	
Команда ПТ			IF	ID	EX	MEM	WB	
Целочисленная команда				IF	ID	EX	MEM	WB
Команда ПТ				IF	ID	EX	MEM	WB

Такой конвейер позволяет существенно увеличить скорость выдачи команд. Однако чтобы он смог так работать, необходимо иметь либо полностью конвейеризованные устройства плавающей точки, либо соответствующее число независимых функциональных устройств. В противном случае устройство плавающей точки станет узким горлом и эффект, достигнутый за счет выдачи в каждом такте пары команд, сведется к минимуму.

При параллельной выдаче двух операций (одной целочисленной команды и одной команды ПТ) потребность в дополнительной аппаратуре, помимо обычной логики обнаружения конфликтов, минимальна: целочисленные операции и операции ПТ используют разные наборы регистров и разные функциональные устройства. Более того, усиление ограничений на выдачу команд, которые можно рассматривать как специфические структурные конфликты (поскольку выдаваться на выполнение могут только определенные пары команд), обнаружение которых требует только анализа кодов операций. Единственная сложность возникает, только если команды

представляют собой команды загрузки, записи и пересылки чисел с плавающей точкой. Эти команды создают конфликты по портам регистров ПТ, а также могут приводить к новым конфликтам типа RAW, когда операция ПТ, которая могла бы быть выдана в том же такте, является зависимой от первой команды в паре.

Проблема регистровых портов может быть решена, например, путем реализации отдельной выдачи команд загрузки, записи и пересылки с ПТ. В случае составления ими пары с обычной операцией ПТ ситуацию можно рассматривать как структурный конфликт. Такую схему легко реализовать, но она будет иметь существенное воздействие на общую производительность. Конфликт подобного типа может быть устранен посредством реализации в регистровом файле двух дополнительных портов (для выборки и записи).

Если пара команд состоит из одной команды загрузки с ПТ и одной операции с ПТ, которая от нее зависит, необходимо обнаруживать подобный конфликт и блокировать выдачу операции с ПТ. За исключением этого случая, все другие конфликты естественно могут возникать, как и в обычной машине, обеспечивающей выдачу одной команды в каждом такте. Для предотвращения ненужных приостановок могут потребоваться дополнительные цепи обхода.

Другой проблемой, которая может ограничить эффективность суперскалярной обработки, является задержка загрузки данных из памяти. В нашем примере простого конвейера команды загрузки имели задержку в один такт, что не позволяло следующей команде воспользоваться результатом команды загрузки без приостановки. В суперскалярном конвейере результат команды загрузки не может быть использован в том же самом и в следующем такте. Это означает, что следующие три команды не могут использовать результат команды загрузки без приостановки. Задержка перехода также становится длиною в три команды, поскольку команда перехода должна быть первой в паре команд. Чтобы эффективно использовать параллелизм, доступный на суперскалярной машине, нужны более сложные методы планирования потока команд, используемые компилятором или аппаратными средствами, а также более сложные схемы декодирования команд.

Рассмотрим, например, что дает разворачивание циклов и планирование потока команд для суперскалярного конвейера. Ниже представлен цикл, который мы уже разворачивали и планировали его выполнение на простом конвейере.

```
Loop: LD F0,0(R1) ;F0=элемент вектора
ADDD F4,F0,F2 ;добавление скалярной величины из F2
SD 0(R1),F4 ;запись результата
SUBI R1,R1,#8 ;декрементирование указателя
;8 байт на двойное слово
BNEZ R1,Loop ;переход R1!=нулю
```

Чтобы спланировать этот цикл для работы без задержек, необходимо его развернуть и сделать пять копий тела цикла. После такого разворачивания цикл будет содержать по пять команд LD, ADDD, и SD, а также одну команду SUBI и один условный переход BNEZ. Развернутая и оптимизированная программа этого цикла дана ниже в табл. 22.

Таблица 22

Оптимизированная работа цикла

Целочисленная команда	Команда ПТ	Номер такта
Loop: LD F0,0(R1)		1
LD F8,-8(R1)		2
LD F10,-16(R1)		3
LD F14,-24(R1)	ADDD F4,F0,F2	4
LD F18,-32(R1)	ADDD F8,F6,F2	5
SD 0(R1),F4	ADDD F12,F10,F2	6
SD -8(R1),F8	ADDD F16,F14,F2	7
SD -16(R1),F12	ADDD F20,F18,F2	8
SD -24(R1),F16		9
SUBI R1,R1,#40		10
BNEZ R1,Loop		11
SD -32(R1),F20		12

Этот развернутый суперскалярный цикл теперь работает со скоростью 12 тактов на итерацию, или 2.4 такта на один элемент (по сравнению с 3.5 тактами для оптимизированного развернутого цикла на обычном конвейере). В этом примере производительность суперскалярного конвейера ограничена существующим соотношением целочисленных операций и операций ПТ, но команд ПТ не достаточно для поддержания полной загрузки конвейера ПТ. Первоначальный оптимизированный неразвернутый цикл выполнялся со скоростью 6 тактов на итерацию, вычисляющую один элемент. Мы получили таким образом ускорение в 2.5 раза, больше половины которого произошло за счет разворачивания цикла. Чистое ускорение за счет суперскалярной обработки дало улучшение примерно в 1.5 раза.

В лучшем случае такой суперскалярный конвейер позволит выбирать две команды и выдавать их на выполнение, если первая из них является целочисленной, а вторая – с плавающей точкой. Если это условие не соблюдается, что легко проверить, то команды выдаются последовательно. Это показывает два главных преимущества суперскалярной машины по сравнению с WLIW-машиной. Во-первых, малое воздействие на плотность кода, поскольку машина сама определяет, может ли быть выдана следующая команда, и нам не надо следить за тем, чтобы команды соответствовали возможностям выдачи. Во-вторых, на таких машинах могут работать неоптимизированные программы, или программы, откомпилированные в расчете на более старую реализацию. Конечно, такие программы не могут

работать очень хорошо. Один из способов улучшить ситуацию заключается в использовании аппаратных средств динамической оптимизации.

В общем случае в суперскалярной системе команды могут выполняться параллельно и возможно не в порядке, предписанном программой. Если не предпринимать никаких мер, такое неупорядоченное выполнение команд и наличие множества функциональных устройств с разными временами выполнения операций могут приводить к дополнительным трудностям. Например, при выполнении некоторой длинной команды с плавающей точкой (команды деления или вычисления квадратного корня) может возникнуть исключительная ситуация уже после того, как завершилось выполнение более быстрой операции, выданной после этой длинной команды. Для того, чтобы поддерживать модель точных прерываний, аппаратура должна гарантировать корректное состояние процессора при прерывании для организации последующего возврата.

Обычно в машинах с неупорядоченным выполнением команд предусматриваются дополнительные буферные схемы, гарантирующие завершение выполнения команд в строгом порядке, предписанном программой. Такие схемы представляют собой некоторый буфер «истории», т. е. аппаратную очередь, в которую при выдаче попадают команды и текущие значения регистров результата этих команд в заданном программой порядке.

В момент выдачи команды на выполнение она помещается в конец этой очереди, организованной в виде буфера FIFO (первый вошел – первый вышел). Единственный способ для команды достичь головы этой очереди – завершение выполнения всех предшествующих ей операций. При неупорядоченном выполнении некоторая команда может завершить свое выполнение, но все еще будет находиться в середине очереди. Команда покидает очередь, когда она достигает головы очереди и ее выполнение завершается в соответствующем функциональном устройстве. Если команда находится в голове очереди, но ее выполнение в функциональном устройстве не закончено, она очередь не покидает. Такой механизм может поддерживать модель точных прерываний, поскольку вся необходимая информация хранится в буфере и позволяет скорректировать состояние процессора в любой момент времени.

Этот же буфер «истории» позволяет реализовать и условное (speculative) выполнение команд (выполнение по предположению), следующих за командами условного перехода. Это особенно важно для повышения производительности суперскалярных архитектур. Статистика показывает, что на каждые шесть обычных команд в программах приходится в среднем одна команда перехода. Если задерживать выполнение следующих за командой перехода команд, потери на конвейеризацию могут оказаться просто неприемлемыми. Например, при выдаче четырех команд в одном такте в среднем в каждом втором такте выполняется команда перехода. Механизм

условного выполнения команд, следующих за командой перехода, позволяет решить эту проблему. Это условное выполнение обычно связано с последовательным выполнением команд из заранее предсказанной ветви команды перехода. Устройство управления выдает команду условного перехода, прогнозирует направление перехода и продолжает выдавать команды из этой предсказанной ветви программы.

Если прогноз оказался верным, выдача команд так и будет продолжаться без приостановок. Однако если прогноз был ошибочным, устройство управления приостанавливает выполнение условно выданных команд и, если необходимо, использует информацию из буфера истории для ликвидации всех последствий выполнения условно выданных команд. Затем начинается выборка команд из правильной ветви программы. Таким образом, аппаратура, подобная буферу истории не только позволяет решить проблемы с реализацией точного прерывания, но и обеспечивает увеличение производительности суперскалярных архитектур.

10. Архитектура машин с длинным командным словом (VLIW). Средства поддержки большой степени распараллеливания

VLIW: старая архитектура нового поколения

Архитектура сверхдлинного командного слова (VLIW – Very Long Instruction Word) берет свое начало от параллельного микрокода, применявшегося еще на заре вычислительной техники, и от суперкомпьютеров Control Data CDC6600 и IBM 360/91. В 1970 г. многие вычислительные системы оснащались дополнительными векторными сигнальными процессорами, использующими VLIW-подобные длинные инструкции, прошитые в ПЗУ. Эти процессоры применялись для выполнения быстрого преобразования Фурье и других вычислительных алгоритмов. Первыми настоящими VLIW-компьютерами стали мини-суперкомпьютеры, выпущенные в начале 1980 г. компаниями MultiFlow, Culler и Cydrome, но они не имели коммерческого успеха. Планировщик вычислений и программная конвейризация были предложены Фишером и Рау (Cydrome). Сегодня это является основой технологии VLIW-компилятора.

Первый VLIW-компьютер компании MultiFlow 7/300 использовал два арифметико-логических устройства для целых чисел, два АЛУ для чисел с плавающей точкой и блок логического ветвления – все это было собрано на нескольких микросхемах. Его 256бит командное слово содержало восемь 32бит кодов операций. Модули для обработки целых чисел могли выполнять две операции за один такт длиной 130ns (т. е. всего четыре при двух АЛУ), что при обработке целых чисел обеспечивало быстродействие около 30 MIPS. Можно было также комбинировать аппаратные решения так, чтобы получать или 256бит, или 1024бит вычислительные машины.

Первый VLIW-компьютер Cydrome Cydra-5 использовал 256бит инструкцию и специальный режим, обеспечивающий выполнение команд как последовательности из шести 40бит операций, поэтому его компиляторы могли генерировать смесь параллельного кода и обычного последовательного. Существует мнение, что в то время, как эти VLIW-ВМ использовали несколько микросхем, процессор Intel i860 стал первым VLIW-процессором на одной микросхеме. Однако, i860 можно отнести к VLIW достаточно условно – по сути у него есть всего лишь программно-управляемое спаривание инструкций, в отличие от более позднего программно-неуправляемого, ставшего частью суперскалярных процессоров. В качестве исторической справки также хотелось бы упомянуть компьютеры фирмы FPS (AP-120B, AP-190L и все более поздние под маркой FPS), также основанные на VLIW-архитектуре, которые были в свое время достаточно распространенными и успешными на рынке. Кроме этого, существовали такие «канонические» машины, как М10 и М13 Карцева, а также «Эльбрус-3» – при всем «неуспехе» последнего проекта, он все же явился этапом VLIW. Вообще, быстродействие VLIW-процессора в большей степени зависит от компилятора, нежели от аппаратуры, поскольку здесь эффект от оптимизации последовательности операций превышает результат, возникающий от повышения частоты.

Архитектура машин с очень длинным командным словом позволяет сократить объем оборудования, требуемого для реализации параллельной выдачи нескольких команд, и потенциально чем большее количество команд выдается параллельно, тем больше эта экономия. Например, суперскалярная машина, обеспечивающая параллельную выдачу двух команд, требует параллельного анализа двух кодов операций, шести полей номеров регистров, а также того, чтобы динамически анализировалась возможность выдачи одной или двух команд и выполнялось распределение этих команд по функциональным устройствам. Хотя требования по объему аппаратуры для параллельной выдачи двух команд остаются достаточно умеренными, и можно даже увеличить степень распараллеливания до четырех (что применяется в современных микропроцессорах), дальнейшее увеличение количества выдаваемых параллельно для выполнения команд приводит к нарастанию сложности реализации из-за необходимости определения порядка следования команд и существующих между ними зависимостей.

Архитектура VLIW базируется на множестве независимых функциональных устройств. Вместо того, чтобы пытаться параллельно выдавать в эти устройства независимые команды, в таких машинах несколько операций упаковываются в одну очень длинную команду. При этом ответственность за выбор параллельно выдаваемых для выполнения операций полно-

стью ложится на компилятор, а аппаратные средства, необходимые для реализации суперскалярной обработки, просто отсутствуют.

VLIW-команда может включать, например, две целочисленные операции, две операции с плавающей точкой, две операции обращения к памяти и операцию перехода. Такая команда будет иметь набор полей для каждого функционального устройства, возможно от 16 до 24 бит на устройство, что приводит к команде длиною от 112 до 168 бит.

Рассмотрим работу цикла инкрементирования элементов вектора на подобного рода машине в предположении, что одновременно могут выдаваться две операции обращения к памяти, две операции с плавающей точкой и одна целочисленная операция либо одна команда перехода. В таблице 23 показан код для реализации этого цикла. Цикл был развернут семь раз, что позволило устранить все возможные приостановки конвейера. Один проход по циклу осуществляется за 9 тактов и вырабатывает 7 результатов. Таким образом, на вычисление каждого результата расходуется 1.28 такта (в нашем примере для суперскалярной машины на вычисление каждого результата расходовалось 2.4 такта).

Таблица 23

Оптимизированная работа цикла на VLIW-машине

Обращение к памяти 1	Обращение к памяти 2	Операция ПТ 1	Операция ПТ 2	Целочисленная операция/ переход

LD F0,0(R1)
LD F10,-16(R1)
LD F18,-32(R1)
LD F26,-48(R1)
SD 0(R1),F4
SD -16(R1),F12
SD -32(R1),F20
SD 0(R1),F28
	LD F6,-8(R1)
	LD F14,-24(R1)
	LD F22,-40(R1)
	SD -8(R1),F8
	SD -24(R1),F16
	SD -40(R1),F24
		ADDD F4,F0,F2
		ADDD F12,F10,F2
		ADDD F20,F18,F2
		ADDD F28,F26,F2
			ADDD F8,F6,F2
			ADDD F16,F14,F2
			ADDD F24,F22,F2
				SUBI R1,R1,#48
				BNEZ R1,Loop

Для машин с VLIW-архитектурой был разработан новый метод планирования выдачи команд, названный «трассировочным планированием». При использовании этого метода из последовательности исходной программы генерируются длинные команды путем просмотра программы за пределами базовых блоков. Как уже отмечалось, базовый блок – это линейный участок программы без ветвлений.

С точки зрения архитектурных идей машину с очень длинным командным словом можно рассматривать как расширение RISC-архитектуры. Как и в RISC-архитектуре аппаратные ресурсы VLIW-машины предоставлены компилятору, и ресурсы планируются статически. В машинах с очень длинным командным словом к этим ресурсам относятся конвейерные функциональные устройства, шины и банки памяти. Для поддержки высокой пропускной способности между функциональными устройствами и ре-

гистрами необходимо использовать несколько наборов регистров. Аппаратное разрешение конфликтов исключается и предпочтение отдается простой логике управления. В отличие от традиционных машин регистры и шины не резервируются, а их использование полностью определяется во время компиляции.

В машинах типа VLIW, кроме того, этот принцип замены управления во время выполнения программы планированием во время компиляции распространен на системы памяти. Для поддержания занятости конвейерных функциональных устройств должна быть обеспечена высокая пропускная способность памяти. Одним из современных подходов к увеличению пропускной способности памяти является использование расслоения памяти. Однако в системе с расслоенной памятью возникает конфликт банка, если банк занят предыдущим обращением. В обычных машинах состояние занятости банков памяти отслеживается аппаратно и проверяется, когда выдается команда, выполнение которой связано с обращением к памяти. В машине типа VLIW эта функция передана программным средствам. Возможные конфликты банков определяет специальный модуль компилятора – модуль предотвращения конфликтов.

Обнаружение конфликтов не является задачей оптимизации, это скорее функция контроля корректности выполнения операций. Компилятор должен быть способен определять, что конфликты невозможны или, в противном случае, допускать, что может возникнуть наихудшая ситуация. В определенных ситуациях, например, в том случае, когда производится обращение к массиву, а индекс вычисляется во время выполнения программы, простого решения здесь нет. Если компилятор не может определить, что конфликт не произойдет, операции не могут планироваться для параллельного выполнения, а это ведет к снижению производительности.

Компилятор с трассировочным планированием определяет участок программы без обратных дуг (переходов назад), которая становится кандидатом для составления расписания. Обратные дуги обычно имеются в программах с циклами. Для увеличения размера тела цикла широко используется методика раскрутки циклов, что приводит к образованию больших фрагментов программы, не содержащих обратных дуг. Если дана программа, содержащая только переходы вперед, компилятор делает эвристическое предсказание выбора условных ветвей. Путь, имеющий наибольшую вероятность выполнения (его называют трассой), используется для оптимизации, проводимой с учетом зависимостей по данным между командами и ограничений аппаратных ресурсов. Во время планирования генерируется длинное командное слово. Все операции длинного командного слова выдаются одновременно и выполняются параллельно.

После обработки первой трассы планируется следующий путь, имеющий наибольшую вероятность выполнения (предыдущая трасса больше не рассматривается). Процесс упаковки команд последовательной программы в длинные командные слова продолжается до тех пор, пока не будет оптимизирована вся программа.

Ключевым условием достижения эффективной работы VLIW-машины является корректное предсказание выбора условных ветвей. Отмечено, например, что прогноз условных ветвей для научных программ часто оказывается точным. Возвраты назад имеются во всех итерациях цикла, за исключением последней. Таким образом, «прогноз», который уже дается самими переходами назад, будет корректен в большинстве случаев. Другие условные ветви, например ветвь обработки переполнения и проверки граничных условий (выход за границы массива), также надежно предсказуемы.

Относительно недавно мы были свидетелями «противостояния» CISC против RISC, а теперь уже намечается новое «сражение» – VLIW против RISC. Строго говоря, VLIW и суперскалярный RISC – никак не антагонисты, ни в коей мере. Справедливости ради необходимо отметить, что последние – это вовсе не «внешнеархитектурное» свойство, а просто некий способ исполнения. Возможно, что в дальнейшем появятся суперскалярные VLIW-процессоры, которые тем самым приобретут, если так можно выразиться, «параллелизм в квадрате» – объединение явного статического параллелизма с неявным динамическим. Но на сегодняшнем этапе развития процессоров нет видимых способов совмещать статическое и динамическое переупорядочивание. Именно поэтому Itanium/Itanium2 сейчас следует рассматривать не столько в контексте сравнения VLIW «против» CISC (и тем более не VLIW «против» O3E), а скорее как «синхронный VLIW» vs «асинхронный (Out-Of-Order) RISC». Не стоит забывать, что альянс Intel-HP придумали для своей архитектуры отдельное название – EPIC, т. е. явный параллелизм.

Несмотря на то, что архитектура VLIW появилась еще на заре компьютерной индустрии (Тьюринг разработал VLIW-компьютер еще в 1946 г.), она до сих пор не имела коммерческого успеха. Теперь Intel воплотила некоторые идеи VLIW в линейке процессоров Itanium. Но значительного повышения производительности и скорости вычислений в системах на базе этих процессоров по отношению к существующим классическим «RISC-inside CISC-outside» архитектурам можно добиться лишь путем переноса интеллектуальных функций из аппаратного обеспечения в программное (компилятор). Таким образом, успех Itanium/Itanium2 определяется в основном программными средствами – именно в этом и состоит проблема. Причем довольно сдержанное отношение индустрии к сравнительно давно существующему Itanium только подтвердило факт ее наличия.

Аппаратно-программный комплекс VLIW

Архитектура VLIW представляет собой одну из реализаций концепции внутреннего параллелизма в микропроцессорах. Их быстродействие можно повысить двумя способами: увеличив либо тактовую частоту, либо количество операций, выполняемых за один такт. В первом случае требуется применение «быстрых» технологий (например, использование арсенида галлия вместо кремния) и таких архитектурных решений, как глубинная конвейеризация (конвейеризация в пределах одного такта, когда в каждый момент времени задействованы все логические блоки кристалла, а не отдельные его части). Для увеличения количества выполняемых за один цикл операций необходимо на одном чипе разместить множество функциональных модулей обработки и обеспечить надежное параллельное исполнение машинных инструкций, что дает возможность включить в работу все модули одновременно. Надежность в таком контексте означает, что результаты вычислений будут правильными. Для примера рассмотрим два выражения, которые связаны друг с другом следующим образом: A=B+C и B=D+E. Значение переменной A будет разным в зависимости от порядка, в котором вычисляются эти выражения (сначала A, а потом B, или наоборот), но ведь в программе подразумевается только одно определенное значение. И если теперь вычислить эти выражения параллельно, то на правильный результат можно рассчитывать лишь с определенной вероятностью, а не гарантировано.

Планирование порядка вычислений – довольно трудная задача, которую приходится решать при проектировании современного процессора. В суперскалярных архитектурах для распознавания зависимостей между машинными инструкциями применяется специальное довольно сложное аппаратное решение (например, в P6- и post-P6-архитектуре от Intel для этого используется буфер переупорядочивания инструкций – ReOrder Buffer, ROB). Однако размеры такого аппаратного планировщика при увеличении количества функциональных модулей обработки возрастают в геометрической прогрессии, что, в конце концов, может занять весь кристалл процессора. Поэтому суперскалярные проекты остановились на отметке 5-6 обрабатываемых за цикл инструкций. На самом же деле, текущие реализации VLIW тоже далеко не всегда могут похвастаться 100% заполнением пакетов – реальная загрузка около 6-7 команд в такте – примерно столько же, сколько и лидеры среди RISC-процессоров. При другом подходе можно передать все планирование программному обеспечению, как это делается в конструкциях с VLIW. «Умный» компилятор должен выискать в программе все инструкции, которые являются совершенно независимыми, собрать их вместе в очень длинные строки (длинные инструкции) и затем отпра-

вить на одновременное исполнение функциональными модулями, количество которых, как минимум, не меньше, чем количество операций в такой длинной команде. Очень длинные инструкции (VLIW) обычно имеют размер 256-1024 бит, однако, бывает и меньше. Сам же размер полей, кодирующих операции для каждого функционального модуля, в тажкой метаинструкции намного меньше.

Логический слой VLIW-процессора

Процессор VLIW, имеющий схему, представленную ниже, может выполнять в предельном случае восемь операций за один такт и работать при меньшей тактовой частоте намного более эффективнее существующих суперскалярных чипов. Добавочные функциональные блоки могут повысить производительность (за счет уменьшения конфликтов при распределении ресурсов), не слишком усложняя чип. Однако такое расширение ограничивается физическими возможностями: количеством портов чтения/записи, необходимых для обеспечения одновременного доступа функциональных блоков к файлу регистров, и взаимосвязей, число которых геометрически растет при увеличении количества функциональных блоков. К тому же компилятор должен распараллелить программу до необходимого уровня, чтобы обеспечить загрузку каждому блоку – это, думается, самый главный момент, ограничивающий применимость данной архитектуры.

Эта гипотетическая инструкция имеет восемь операционных полей, каждое из которых выполняет традиционную трехоперандную RISC-подобную инструкцию типа <регистр приемника> = <регистр источника 1> - <операция> - <регистр источника 2> (типа классической команды MOV AX BX) и может непосредственно управлять специфическим функциональным блоком при минимальном декодировании.

Для большей конкретности рассмотрим кратко IA-64 как один из примеров воплощения VLIW. Со временем эта архитектура способна вытеснить x86 (IA-32) не только на рынке, но вообще как класс, хотя это уже удел далекого будущего. Тем не менее, необходимость разработки для IA-64 весьма сложных компиляторов и трудности с созданием оптимизированных машинных кодов может вызвать дефицит специалистов, работающих на ассемблере IA-64, особенно на начальных этапах, как самых сложных.

Наиболее кардинальным нововведением IA-64 по сравнению с RISC является явный параллелизм команд (EPIC), вносящий некоторые элементы, напоминающие архитектуру сверхдлинного командного слова, которые назвали связками (bundle). Так, в обеих архитектурах явный параллелизм представлен уже на уровне команд, управляющих одновременной работой функциональных исполнительных устройств (или функциональных модулей, или просто функциональных устройств, ФУ).

В данном случае связка имеет длину 128бит и включает в себя 3 поля для команд длиной 41бит каждое, и 5-разрядный слот шаблона. Предполагается, что команды связки могут выполняться параллельно разными ФУ. Возможные взаимозависимости, препятствующие параллельному выполнению команд одной связки, отражаются в поле шаблона. Не утверждается, впрочем, что параллельно не могут выполняться и команды разных связок. Хотя, на основании заявленного уровня параллельности исполнения, достигающего шести команд за такт, логично предположить, что одновременно могут выполняться как минимум две связки.

Шаблон указывает, какого непосредственно типа команды находятся в слотах связки. В общем случае однотипные команды могут выполняться в более чем одном типе функциональных устройств. Шаблоном задаются так называемые остановки, определяющие слот, после начала выполнения команд которого инструкции последующих полей должны ждать завершения. Порядок слотов в связке (более важные справа) отвечает и порядку байт (Little Endian), однако данные в памяти могут располагаться и в режиме Big Endian (более важные слева), который устанавливается специальным битом в регистре маски пользователя.

Вращение регистров является в некотором роде частным случаем переименования регистров, применяемого во многих современных суперскалярных процессорах с внеочередным спекулятивным («умозрительным») выполнением команд. В отличие от них, вращение регистров в IA-64 управляется программно. Использование этого механизма в IA-64 позволяет избежать накладных расходов, связанных с сохранением/восстановлением большого числа регистров при вызовах подпрограмм и возвратах из них, однако статические регистры при необходимости все-таки приходится сохранять и восстанавливать, явно кодируя соответствующие команды.

К слову, система команд IA-64 довольно уникальна. Среди принципиальных особенностей следует отдельно отметить спекулятивное выполнение команд и применение предикатов – именно это подмножество и определяет исключительность IA-64. Все подобные команды можно подразделить на команды работы со стеком регистров, целочисленные команды, команды сравнения и работы с предикатами, команды доступа в память, команды перехода, мультимедийные команды, команды пересылок между регистрами, «разные» (операции над строками и подсчет числа единиц в слове) и команды работы с плавающей запятой.

Аппаратная реализация VLIW-процессора очень проста: несколько небольших функциональных модулей (сложения, умножения, ветвления и т. д.), подключенных к шине процессора, и несколько регистров и блоков кэш-памяти. VLIW-архитектура представляет интерес для полупроводниковой промышленности по двум причинам. Первая – теперь на кристалле

больше места может быть отведено для блоков обработки, а не, скажем, для блока предсказания переходов. Вторая причина – VLIW-процессор может быть высокоскоростным, так как предельная скорость обработки определяется только внутренними особенностями самих функциональных модулей. Привлекает и то, что VLIW при определенных условиях может реализовать старые CISC-инструкции эффективнее RISC. Это потому, что программирование VLIW-процессора очень напоминает написание микрокода (исключительно низкоуровневый язык, позволяющий всесторонне программировать физический слой, синхронизируя работу логических вентилей с шинами обмена данными и управляя передачей информации между функциональными модулями).

В те времена, когда память для ПК была дорогостоящей, программисты экономили ее, прибегая к сложным инструкциям процессора x86 типа STOS и LODS (косвенная запись/чтение в/из памяти). CISC реализует такие инструкции, как микропрограммы, зашитые в постоянную память (ROM) и выполняемые процессором. Архитектура RISC вообще исключает использование микрокода, реализуя инструкции чисто аппаратным путем – фактически, инструкции RISC-процессора почти аналогичны микрокоду, используемому в CISC. VLIW делает по-другому – изымает процедуру генерирования микрокода из процессора (да и вообще стадии исполнения) и переносит его в компилятор, на этап создания исполняемого кода. В результате эмуляция инструкций процессора x86, таких как STOS, осуществляется очень эффективно, поскольку процессор получает для исполнения уже готовые макросы. Но вместе с тем, это порождает и некоторые трудности, поскольку написание достаточно эффективного микрокода – невероятно трудоемкий процесс. Архитектуре VLIW может обеспечить жизнеспособность только «умный» компилятор, который возьмет эту работу на себя. Именно это обстоятельство ограничивает использование вычислительных машин с архитектурой VLIW: пока они нашли свое применение в основном в векторных (для научных расчетов) и сигнальных процессорах.

Принцип действия VLIW-компилятора

Вновь вспыхнувший в последнее время интерес к VLIW как к архитектуре, которую можно использовать для реализации вычислений общего назначения, дал существенный толчок развитию техники VLIW-компиляции. Такой компилятор упаковывает группы независимых операций в очень длинные слова инструкций таким способом, чтобы обеспечить быстрый запуск и более эффективное их исполнение функциональными модулями. Компилятор сначала обнаруживает все зависимости между данными, а затем определяет, как их развязать. Чаще всего это делается путем переупорядочивания всей программы – разные ее блоки перемещаются с одного места в другое. Данный подход отличается от применяемого в суперскалярном процессоре, который для определения зависимостей исполь-

зует специальное аппаратное решение прямо во время выполнения приложения (оптимизирующие компиляторы, безусловно, улучшают работу суперскалярного процессора, но не делают его «привязанным» к ним). Большинство суперскалярных процессоров может обнаружить зависимости и планировать параллельное исполнение только внутри базовых программных блоков (группа последовательных операторов программы, не содержащих внутри себя останова или логического ветвления, допустимых только в конце). Некоторые переупорядочивающие системы положили начало расширению области сканирования, не ограничивая ее базовыми блоками. Для обеспечения большего параллелизма VLIW-компьютеры должны наблюдать за операциями из разных базовых блоков, чтобы поместить эти операции в одну и ту же длинную инструкцию (их «область обзора» должна быть шире, чем у суперскалярных процессоров) – это обеспечивается путем прокладки «маршрута» по всей программе (трассировка). Трассировка – наиболее оптимальный для некоторого набора исходных данных маршрут по программе (для обеспечения правильного результата гарантируется не пересечение этих данных), т. е. маршрут, который «проходит» по участкам, пригодным для параллельного выполнения (эти участки формируются, кроме всего прочего, и путем переноса кода из других мест программы), после чего остается упаковать их в длинные инструкции и передать на выполнение. Планировщик вычислений осуществляет оптимизацию на уровне всей программы, а не ее отдельных базовых блоков. Для VLIW, так же как и для RISC, ветвления в программе являются «врагом», препятствующим эффективному ее выполнению. В то время как RISC для прогнозирования ветвлений использует аппаратное решение, VLIW оставляет это компилятору. Сам компилятор использует информацию, собранную им путем профилирования программы, хотя у будущих VLIW-процессоров предполагается небольшое аппаратное расширение, обеспечивающее сбор для компилятора статистических данных непосредственно во время выполнения программы, что принципиально важно при циклической работе с переменным набором. Компилятор прогнозирует наиболее подходящий маршрут и планирует прохождение, рассматривая его как один большой базовый блок, затем повторяет этот процесс для всех других возникших после этого программных веток, и так до самого конца программы. Он также умеет делать при анализе кода и другие «интеллектуальные шаги», такие как развертывание программного цикла и IF-преобразование, в процессе которого временно удаляются все логические переходы из секции, подвергающейся трассировке. Там, где RISC может только просмотреть код вперед на предмет ветвлений, VLIW-компилятор перемещает его с одного места в другое до обнаруженного ветвления (согласно трассировке), но предусматривает при необходимости возможность отката назад, к предыдущему программному состоянию. Формально ничего не мешает это

же сделать и RISC процессору, просто соотношение «цена/эффективность» оказывается слишком высоким. Соответствующее аппаратное обеспечение, добавленное к VLIW-процессору, может оказать определенную поддержку компилятору. Например, операции, имеющие по несколько ветвлений, могут входить в одну длинную инструкцию и, следовательно, выполняться за один машинный такт. Поэтому выполнение условных операций, которые зависят от предыдущих результатов, может быть реализовано программным способом, а не аппаратным. Цена, которую приходится платить за увеличение быстродействия VLIW-процессора, намного меньше стоимости компиляции – именно поэтому основные расходы приходятся на сами компиляторы.

VLIW: обратная сторона медали

Тем не менее, при реализации архитектуры VLIW возникают и другие серьезные проблемы. VLIW-компилятор должен в деталях «знать» внутренние особенности архитектуры процессора, опускаясь до устройства самих функциональных блоков. Как следствие, при выпуске новой версии VLIW-процессора с большим количеством обрабатывающих модулей (или даже с тем же количеством, но другим быстродействием) все старое программное обеспечение может потребовать полной перекомпиляции. Производители VLIW-процессоров обрекли себя на, как минимум, не уменьшение ширины пакета, хотя бы ради того, чтобы «старые» программы могли гарантированно исполнятся на новых устройствах. Например, на настоящее время существуют пакеты по 8 команд, а следующая версия не может иметь в реализации всего шесть функциональных устройств даже ради двух-трех- кратного прироста по частоте. Кроме того, что программа, скомпилированная для восьмиканального VLIW, не сможет без специальных дорогостоящих (и в плане сложности, и в плане производительности) аппаратных решений исполняться на шестиканальной архитектуре, придется радикально переписывать и компилятор. С этой точки зрения представляется разумным использование Intel трехкомандного слова в системе IA64 – такое, на первый взгляд, неудобное ограничение позволяет в будущем довольно свободно варьировать число исполнительных устройств в процессорах IA64. И если при переходе с 386 на процессор 486 производить перекомпиляцию имеющегося ПО было совершенно ненужно, то теперь придется. В качестве одного из возможных компромиссных решений предлагается разделить процесс компиляции на две стадии. Все программное обеспечение должно готовиться в аппаратно-независимом формате с использованием промежуточного кода, который окончательно транслируется в машинно-зависимый код только в процессе установки на оборудовании конечного пользователя. Пример такого подхода демонстрирует фонд OSF со своим архитектурно-независимым форматом ANDF (Architecture-Neutral Distribution Format). Но кроссплатформенное программное обеспе-

чение пока что не оправдывает когда-то возлагавшихся на него радужных надежд. Во-первых, оно все равно требует наличия портов, которые «объясняют» компилятору в каждом конкретном случае, что следует делать при компиляции данной программы именно на этой платформе (и даже именно на этой ОС). Во-вторых, кроссплатформенное ПО пока отнюдь не является «Speed Demon», а даже наоборот, как правило работает медленнее, чем написанное под конкретную платформу аналогичного класса приложения.

Другая трудность – это по своей сути статическая природа оптимизации, которую обеспечивает VLIW-компилятор. Трудно предугадать как, например, поведет себя программа, когда столкнется во время компиляции с непредусмотренными динамическими ситуациями, такими как ожидание ввода/вывода. Архитектура VLIW возникла в ответ на требования со стороны научно-технических организаций, где при вычислениях особенно необходимо большое быстродействие процессора, но для объектно-ориентированных и управляемых по событиям программ она менее подходит, а ведь именно такие приложения составляют сейчас большинство в сфере Информационных Технологий. Остается труднопредполагаемой проверка, что компилятор выполняет такие сложные преобразования надежно и правильно. Напротив, Out-Of-Order RISC-процессоры вполне способны «адаптироваться» под конкретную ситуацию самым выгодным образом.

Средства поддержки большой степени распараллеливания

В этом разделе мы обсудим методы компиляции, которые позволяют увеличить степень параллелизма, который можно использовать при выполнении программы. Мы начнем с изучения методов обнаружения зависимостей и устранения зависимостей по именам.

Обнаружение и устранение зависимостей

Нахождение зависимостей по данным в программе является важной частью трех задач: хорошее планирование программного кода, определение циклов, которые могут содержать параллелизм, и устранение зависимостей по именам. Сложность анализа зависимостей связана с наличием массивов (и указателей в языках, подобных языку Си). Поскольку обращения к скалярным переменным осуществляется явно по имени, они обычно могут анализироваться достаточно просто. При этом наличие указателей-алиасов и обращений к параметрам вызывает усложнения, поскольку они могут быть неизвестны в процессе анализа.

При анализе необходимо найти все зависимости и определить, имеется ли цикл в этих зависимостях, поскольку это то, что не позволяет нам выполнять цикл параллельно. Рассмотрим следующий пример:

```
for (i=1;i<=100;i=i+1) {
A[i] = B[i] + C[i];
D[i] = A[i] + E[i];
}
```

Поскольку в данном случае зависимость, связанная с А, не приводит к зависимости между итерациями цикла, можно развернуть цикл для выявления большей степени параллелизма. Мы не можем прямо поменять местами два обращения к А. Если цикл имеет зависимости между итерациями, которые не являются циклическими, можно сначала преобразовать цикл для устранения этих зависимостей, а затем развернуть цикл для выявления большей степени параллелизма. Во многих параллельных циклах степень параллелизма ограничена только количеством разворотов цикла, которое в свою очередь ограничивается только количеством итераций цикла. Конечно на практике, чтобы получить выигрыш от этой большей степени параллелизма, потребуется много функциональных устройств и огромное количество регистров. Отсутствие зависимости между итерациями цикла просто сообщает нам, что нам доступна большая степень параллелизма.

Фрагмент вышеприведенного кода иллюстрирует также другую возможность для улучшения машинного кода. Второе обращение к А не нужно транслировать в команду загрузки из памяти, поскольку мы знаем, что значение вычислено и записано предыдущим оператором. Поэтому второе обращение к А может выполняться с помощью обращения к регистру, в котором значение А было вычислено. Выполнение этой оптимизации требует знания того, что два обращения всегда относятся к одному и тому же адресу памяти, и что к той же самой ячейке между этими двумя обращениями другие обращения (по записи) отсутствуют. Обычно анализ зависимостей по данным дает информацию только о том, что одно обращение может зависеть от другого. Для определения того, что два обращения должны выполняться точно по одному и тому же адресу, требуется более сложный анализ. В вышеприведенном примере достаточно простейшей версии такого анализа, поскольку оба обращения находятся в одном и том же базовом блоке.

Часто зависимости между итерациями цикла появляются в форме рекуррентного отношения:

```
for (i=2;i<=100;i=i+1) {
Y[i] = Y[i-1] + Y[i];
}
```

Определение наличия рекуррентных отношений может оказаться важным по двум причинам. Некоторые архитектуры (особенно векторные машины) имеют специальную поддержку для выполнения рекуррентных отношений и некоторые рекуррентные отношения могут быть источником значительной степени параллелизма. Например, рассмотрим цикл:

```
for (i=6;i<=100;i=i+1) {
Y[i] = Y[i-5] + Y[i];
}
```

На итерации j цикл обращается к элементу j-5. Говорят, что цикл имеет зависимость с расстоянием 5. Предыдущий цикл имел зависимость с расстоянием 1. Чем больше расстояние, тем больше степень потенциального параллелизма, которую можно получить при помощи разворачивания цикла. Например, если мы разворачиваем первый цикл, имеющий зависимость с расстоянием 1, последовательные операторы зависят друг от друга; имеется некоторая степень параллелизма между отдельными командами, но не очень большая. Если мы разворачиваем цикл, который имеет зависимость с расстоянием 5, то имеется последовательность пяти команд, которые не имеют зависимостей, и тем самым обладают значительно большей степенью параллелизма уровня команд. Хотя многие циклы с зависимостями между итерациями имеют расстояние зависимостей 1, случаи с большими расстояниями в действительности возникают, и большее расстояние между зависимостями может обеспечивать достаточную степень параллелизма для поддержания машины занятой.

В общем случае во время компиляции мы не можем определить, имеет ли место зависимость. Например, значения a, b, c и d могут быть неизвестными (они могут быть значениями другого массива), а следовательно, невозможно сказать, что зависимость существует. В других случаях проверка зависимостей может оказаться очень дорогой, но в принципе возможной во время компиляции. Например, обращения могут зависеть от индексов итераций множества вложенных циклов. Однако многие программы содержат в основном простые индексы, где a, b, c и d все являются константами. Для этих случаев возможно придумать недорогие тесты для обнаружения зависимостей.

Кроме определения наличия зависимостей, компилятор старается также классифицировать тип зависимости. Это позволяет компилятору распознать зависимости по именам и устранить их путем переименования и копирования.

Например, следующий цикл имеет несколько типов зависимостей. Попробуем найти все истинные зависимости, зависимости по выходу и антизависимости и устранить зависимости по выходу и антизависимости с помощью переименования.

```
for (i=1;i<=100;i=i+1) {
Y[i] = X[i] / c; /*S1*/
X[i] = X[i] + c; /*S2*/
Z[i] = Y[i] + c; /*S3*/
Y[i] = c - Y[i]; /*S4*/
}
```

В этих четырех операторах имеются следующие зависимости.

Имеются истинные зависимости от S1 к S3 и от S1 к S4 из-за Y[i]. Отсутствует зависимость между итерациями цикла, что позволяет рас-

сматривать цикл как параллельный. Эти зависимости приведут к ожиданию операторами S3 и S4 завершения оператора S1.

Имеется антизависимость от S1 к S2.

Имеется зависимость по выходу от S1 к S4.

Следующая версия цикла устраняет эти ложные (или псевдо-) зависимости.

```
for (i=1;i<=100;i=i+1) {
/* Y переименовывается в T для устранения
зависимости по выходу */
T[i] = X[i] / c;
/* X переименовывается в X1 для устранения
антизависимости */
X1[i] = X[i] + c;
Z[i] = T[i] + c;
Y[i] = c - T[i];
}
```

После цикла переменная X оказалась переименованной в X1. В коде программы, следующем за циклом, компилятор просто может заменить имя X на имя X1. В данном случае переименование не требует действительной операции копирования, а может быть выполнено с помощью заменяющего имени или соответствующего распределения регистров. Однако в других случаях переименование может потребовать копирования.

Анализ зависимостей является важнейшей технологией для улучшения использования параллелизма. На уровне команд она дает информацию, необходимую для изменения в процессе планирования порядка обращений к памяти, а также для определения полезности разворачивания цикла. Для обнаружения параллелизма уровня цикла анализ зависимостей является базовым инструментом. Эффективная компиляция программ для векторных машин, а также для мультипроцессоров существенно зависит от этого анализа. Кроме того, при планировании потока команд полезно определить, являются ли потенциально зависимыми обращения к памяти. Главный недостаток анализа зависимостей заключается в том, что он применим при ограниченном наборе обстоятельств. Имеется огромное многообразие ситуаций, при которых анализ зависимостей не может сообщить нам то, что мы хотели бы знать, а именно:

1) когда обращения к объектам выполняются с помощью указателей, а не индексов массива;

2) когда индексация массива осуществляется косвенно через другой массив, что имеет место при работе с разреженными массивами;

3) зависимость может существовать для некоторого значения входов, но отсутствовать в действительности при выполнении программы, поскольку входы никогда не принимают определенных значений;

4) когда оптимизация зависит не просто от знания возможности наличия зависимости, но требует точного определения того, от какой операции записи зависит чтение переменной.

Программная конвейеризация: символическое разворачивание циклов

Мы уже видели, что один из методов компиляции – разворачивание циклов – полезен для увеличения степени параллелизма на уровне команд посредством создания более длинных последовательностей линейного кода. Имеются два других важных метода, которые разработаны для этих целей: программная конвейеризация и планирование трасс.

Программная конвейеризация – это метод реорганизации циклов таким образом, что каждая итерация в программно конвейеризованном коде составляется из команд, выбранных из разных итераций первоначального цикла. Планировщик по существу чередует команды из разных итераций цикла так, чтобы отдалить друг от друга зависимые команды, которые возникают в одной итерации цикла. Программно конвейеризованный цикл чередует команды из разных итераций без реального разворачивания цикла (рис. 50). Этот метод по существу программно выполняет то, что алгоритм Томасуло делает с помощью аппаратных средств. Программно конвейеризованный цикл будет содержать по одной команде, каждая из которых относится к разным итерациям. Для начального запуска цикла (пролог цикла) и для завершения цикла (эпилог цикла) требуются некоторые команды.

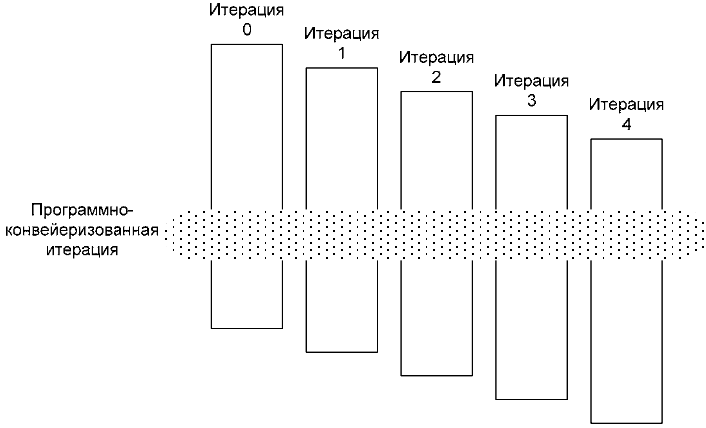


Рис. 50. Программная конвейеризация

Например, рассмотрим программно конвейеризованную версию нижеприведенного цикла, который складывает с содержимым регистра F2 все элементы некоторого массива с начальным адресом, хранящимся в регистре R1.

```
Loop: LD F0,0(R1)
ADDD F4,F0,F2
SD 0(R1),F4
SUBI R1,R1,#8
BNEZ R1, Loop
```

Игнорируя пролог и эпилог мы можем переписать цикл следующим образом:

```
Loop: SD 0(R1),F4 ; записывает в M[i]
ADDD F4,F0,F2 ; складывает с M[i-1]
LD F0,-16(R1); загружает M[i-2]
BNEZ R1, Loop
SUBI R1,R1,#8 ; вычитает в слоте задержки
```

Если не принимать во внимание пролог и эпилог, этот цикл может работать со скоростью 5 тактов на один проход. Поскольку команда загрузки осуществляет выборку на расстоянии двух элементов от счетчика цикла, цикл должен выполнять на две итерации меньше. При этом перед началом цикла из содержимого регистра R1 необходимо вычесть 16. Заметим, что повторное использование регистров (например, F4, F0 и R1) требует использования специальных аппаратных средств, чтобы обойти конфликты типа WAR и приостановки конвейера. В данном случае это не должно привести к каким-либо проблемам, поскольку никаких приостановок по причине зависимостей по данным произойти не должно.

Управление регистрами в программно конвейеризуемых циклах может быть достаточно сложным. Вышеприведенный пример не слишком тяжелый, поскольку в регистры выполняется запись в одной итерации, а их чтение происходит в следующей. В других случаях может потребоваться увеличить количество итераций между моментом выдачи команды и моментом, когда используется ее результат. Это происходит, когда в теле цикла имеется небольшое количество команд, а задержки их выполнения достаточно большие. В этих случаях требуется комбинация методов программной конвейеризации и разворачивания цикла.

Программную конвейеризацию можно рассматривать как символическое разворачивание цикла. Действительно, некоторые алгоритмы программной конвейеризации используют разворачивание цикла в качестве исходного материала для расчета (вычисления) выполнения программной конвейеризации. Главное преимущество программной конвейеризации по отношению к прямому разворачиванию циклов заключается в том, что первая генерирует в результате меньший по размеру программный код. Программная конвейеризация и разворачивание циклов в дополнение к тому, что они дают лучше спланированный внутренний цикл, сами по себе сокращают разные типы накладных расходов. Разворачивание циклов сокращает накладные расходы на организацию цикла, связанные с командами перехода и изменения значения счетчика циклов. Программная конвейеризация сокращает время, когда цикл не работает с полной скоростью, что происходит только однажды в начале и в конце цикла. Если мы разворачиваем цикл, который выполняет 100 итераций постоянное количество раз, скажем 4 раза, мы будем иметь накладные расходы 100/4=25 раз – ка-

ждый раз, когда будет инициироваться внутренний развернутый цикл. Поскольку эти методы направлены на два различных типа накладных расходов, наилучший результат может быть получен при использовании обоих методов.

Трассировочное планирование

Другим методом, используемым для выделения дополнительного параллелизма, является трассировочное планирование. Трассировочное планирование расширяет метод разворачивания циклов методикой для нахождения параллелизма в программах с условными переходами, не связанными с организацией циклов. Трассировочное планирование полезно для машин с очень большим количеством команд, выдаваемых для выполнения в одном такте, где одного разворачивания циклов может оказаться недостаточно для выявления необходимой степени параллелизма уровня команд для поддержания машины в занятом состоянии. Трассировочное планирование является комбинацией двух отдельных процессов. Первый процесс, который называется выбором трассы (trace selection), старается найти возможную последовательность базовых блоков, операции которых будут собираться вместе в меньшее количество команд; эта последовательность называется трассой. Разворачивание циклов используется для генерации длинных трасс, поскольку переходы циклов выполняются с высокой вероятностью. Дополнительно при использовании статического прогнозирования направления переходов другие условные переходы (не связанные с организацией цикла) также выбираются как выполняемые или как невыполняемые, так что результирующая трасса представляет собой линейную последовательность, полученную в результате конкатенации (объединения) многих базовых блоков. Когда трасса выбрана, другой процесс, называемый уплотнением трассы (trace compaction), старается сжать трассу в небольшое количество широких команд. Процесс уплотнения трасс пытается перенести операции как можно ближе к началу последовательности (трассы), упаковывая операции насколько это возможно в минимальное количество широких команд (или пакетов для выдачи).

Уплотнение трассы представляет собой процесс глобального планирования кода. Имеются два разных ограничения, которые возникают и должны обрабатываться любой схемой глобальной оптимизации кода: зависимости по данным, которые задают определенный порядок операций, и точки условного перехода, которые создают места, через которые команды не могут просто перемещаться. По существу код должен быть уплотненным в наиболее короткую последовательность, которая сохраняет зависимости по данным и по управлению. Зависимости по данным преодолеваются посредством разворачивания циклов и использования анализа зависимостей для определения того, относятся ли два обращения к одному и тому же адресу. Зависимости по управлению также сокращаются при раз-

ворачивании циклов. Главным преимуществом методики трассировочного планирования по сравнению с более простым методом планирования загрузки конвейера заключается в том, что она обеспечивает схему для снижения эффекта зависимостей по управлению посредством переноса команд через условные переходы, не связанные с циклами, используя прогнозируемое поведение переходов. На рисунке 51 показан фрагмент кода, который может рассматриваться как итерация развернутого цикла, и выбранная трасса.

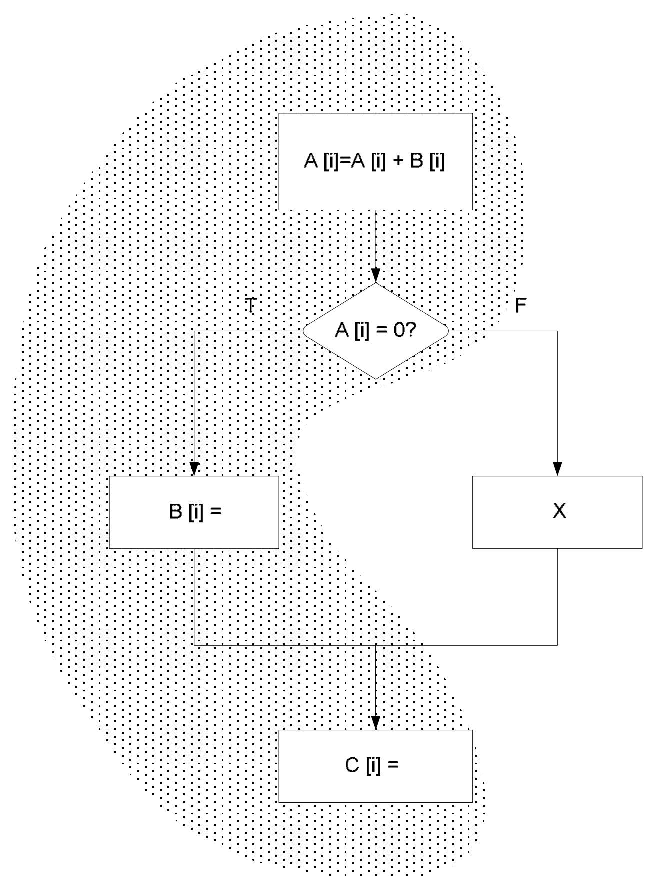


Рис. 51. Фрагмент кода с выбранной трассой

Когда трасса, как показано на рис. 51, выбрана, она должна быть уплотнена так, чтобы заполнить машинный ресурс. Уплотнение трассы приводит к перемещению операторов присваиваний переменным B и C вверх по блоку, чтобы разместить их перед точкой решения о направлении перехода. Любая схема глобального планирования, включая трассировочное планирование, выполняет такое перемещение команд при наличии набора ограничений. В трассировочном планировании условные переходы рассматриваются как безусловные переходы во внутрь или во вне выбранной трассы, которая предполагалась как наиболее вероятный путь. Когда команды перемещаются через такие точки входа и выхода трассы, во входной и выходной точке могут потребоваться дополнительные команды. Главное предположение состоит в том, что выбранная трасса является

наиболее вероятным событием, в противном случае стоимость дополнительной работы (дополнительных команд) может оказаться чрезмерной.

Что включает в себя процесс перемещения присваиваний B и C? Вычисление и присваивание B является зависимым по управлению от условного перехода, а вычисление C нет. Перемещение этих операторов может быть выполнено только, если ни один из них не меняет зависимость по управлению или по данным, или эффект от изменения зависимости не виден и тем самым не приводит к изменению выполнения программы. Рассмотрим типичную последовательность генерации кода для приведенного выше случая (рис. 52).

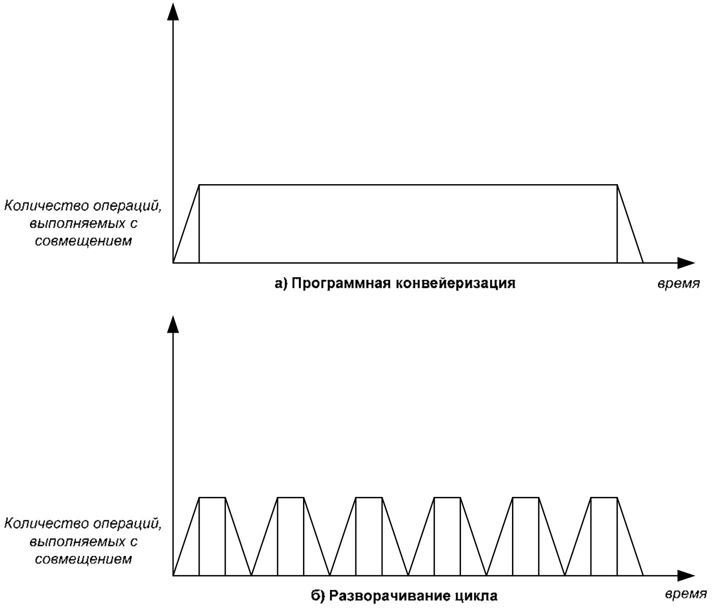


Рис. 52. Генерация кода для различных методик

Ниже представлена такая последовательность в предположении, что адреса для A, B и C находятся в регистрах R1, R2 и R3 соответственно:

```
LW R4,0(R1) ;загрузка A[i]
ADDI R4,R4,... ;добавление к A[i]
SW 0(R1),R4 ;запись в A[i]
...
BNEZ R4,elsepart ;проверка A[i]
... ;часть then
SW 0(R2),... ;запись в B[i]
J join ;прыжок через else
elsepart: ... ;часть else
X ;код для X
...
join: ... ;после if
SW 0(R3),... ;запись в C[i]
```

Сначала рассмотрим проблему перемещения операции присваивания B на место перед командой BNEZ. Поскольку B зависит по управлению от того перехода, перед которым мы ее хотим расположить, и не будет зависеть от него после перемещения, необходимо гарантировать, что выполнение оператора не может вызвать появление исключительной ситуации, поскольку такая исключительная ситуация не могла возникнуть в первоначальной программе, если бы была выбрана часть else условного оператора. Перемещение B не должно также воздействовать на поток данных. Чтобы более ясно определить требования, нам нужна концепция живучести переменной. Переменная Z живет в операторе, если имеется путь выполнения от этого оператора до использования переменной Z, на котором нового присваивания переменной Z не делается. На интуитивном уровне, переменная живет в операторе, если добавление операции присваивания этой переменной в операторе может изменить семантику программы.

Возвращаясь к нашему примеру, можно видеть, что имеются два возможных случая, когда перемещение B может изменить поток данных в этой программе:

1) обращение к B происходит в коде X (часть else) прежде, чем B будет присвоено новое значение;

2) B «живет» в конце оператора if и ей не делается присваивания в X.

В обоих случаях перемещение операции присваивания переменной B приведет к тому, что некоторая команда i (либо в X, либо далее в программе) станет зависимой по данным от этой перемещенной команды, а не от более ранней операции присваивания B, которая выполняется перед циклом и от которой i первоначально зависела. Поскольку это приведет к изменению результата программы, операция присваивания B не может быть перемещена в случае, если справедливо любое из приведенных выше условий. Можно представить себе более изощренные схемы: например, в первом случае перед оператором if можно сделать теневую копию B и использовать только эту теневую копию в X. Такие схемы в общем случае не используются, поскольку, во-первых, их сложно реализовать, и, во-вторых, поскольку они будут замедлять программу, если выбранная трасса не оптимальна и завершение операций требует выполнения дополнительных команд.

Для перемещения операции присваивания C на место сразу за первым условным переходом требуется, чтобы она переместилась выше точки объединения трассы (входа трассы) с направлением else. Это делает команды для C зависимыми по управлению от условного перехода и означает, что они не будут выполняться, если выбран путь else, который не находится на трассе. Поэтому будут затронуты команды, которые были зависимыми по данным от присваивания C и которые выполняются после этого кодового фрагмента. Для обеспечения вычисления корректного значения

для этих команд делается копия команд, которые вычисляют и присваивают значение C на переходе, на трассе, а именно в конце X на пути else. Мы можем переместить C из ветви then перехода через условие перехода, если это не воздействует ни на какой поток данных в условии перехода. Если C перемещается на место перед проверкой условия if, копия C в части else перехода может быть ликвидирована.

Все рассмотренные методы: разворачивание цикла, планирование трасс и программная конвейеризация стараются увеличить степень параллелизма уровня команд, который может использоваться машиной, выдающей для выполнения более одной команды в каждом такте. Эффективность каждого из этих методов и их удобство для различных архитектурных подходов являются наиболее горячими темами, которыми активно занимаются исследователи и разработчики высокоскоростных процессоров.

Аппаратные средства поддержки большой степени распараллеливания. Методы, подобные разворачиванию циклов и планированию трасс, могут использоваться для увеличения степени доступного параллелизма, когда поведение условных переходов достаточно предсказуемо во время компиляции. Если же поведение переходов не известно, одной тех.-ники компиляторов может оказаться не достаточно для выявления большей степени параллелизма уровня команд. В этом разделе представлены два метода, которые могут помочь преодолеть подобные ограничения. Первый метод заключается в расширении набора команд условными или предикатными командами. Такие команды могут использоваться для ликвидации условных переходов и помогают компилятору перемещать команды через точки условных переходов. Условные команды увеличивают степень параллелизма уровня команд, но имеют существенные ограничения. Для использования большей степени параллелизма раз работчики исследовали идею, которая называется «выполнением по предположению» (speculation), и позволяет выполнить команду еще до того, как процессор узнает, что она должна выполняться (т. е. этот метод позволяет избежать приостановок конвейера, связанных с зависимостями по управлению).

Условные команды

Концепция, лежащая в основе условных команд, достаточно проста: команда обращается к некоторому условию, оценка которого является частью выполнения команды. Если условие истинно, то команда выполняется нормально; если условие ложно, то выполнение команды осуществляется, как если бы это была пустая команда. Многие новейшие архитектуры включают в себя ту или иную форму условных команд. Наиболее общим примером такой команды является команда условной пересылки, которая выполняет пересылку значения одного регистра в другой, если условие ис-

тинно. Такая команда может использоваться для полного устранения условных переходов в простых последовательностях программного кода.

Например, рассмотрим следующий оператор:

if (A=0) {S=T;};

Предполагая, что регистры R1, R2 и R3 хранят значения A, S и T соответственно, представим код этого оператора с командой условного перехода и с командой условной пересылки.

Код с использованием команды условного перехода будет иметь следующий вид:

```
BEQZ R1,L
MOV R2,R3
L:
```

Используя команду условной пересылки, которая выполняет пересылку только если ее третий операнд равен нулю, мы можем реализовать этот оператор с помощью одной команды:

```
CMOVZ R2,R3,R1
```

Условная команда позволяет преобразовать зависимость по управлению, присутствующую в коде с командой условного перехода, в зависимость по данным. (Это преобразование используется также в векторных машинах, в которых оно называется if-преобразованием – if-convertion). Для конвейерной машины такое преобразование позволяет перенести точку, в которой должна разрешаться зависимость, от начала конвейера, где она разрешается для условных переходов, в конец конвейера, где происходит запись в регистр.

Одним из примеров использования команд условной пересылки является реализация функции вычисления абсолютного значения: A = abs (B), которая реализуется оператором

if (B<0) {A=-B} else {A=B}.

Этот оператор if может быть реализован парой команд условных пересылок или командой безусловной пересылки (A=B), за которой следует команда условной пересылки (A=-B).

Условные команды могут использоваться также для улучшения планирования в суперскалярных или VLIW-процессорах. Ниже приведен пример кодовой последовательности для суперскалярной машины с одновременной выдачей для выполнения не более двух команд. При этом в каждом такте может выдаваться комбинация одной команды обращения к памяти и одной команды АЛУ или только одна команда условного перехода:

```
LW R1,40(R2) ADD R3,R4,R5
ADD R6,R3,R7
BEQZ R10,L
LW R8,20(R10)
LW R9,0(R8)
```

Эта последовательность теряет слот операции обращения к памяти во втором такте и приостанавливается из-за зависимости по данным, если переход невыполняемый, поскольку вторая команда LW после перехода зависит от предыдущей команды загрузки. Если доступна условная версия команды LW, то команда LW, немедленно следующая за переходом (LW R8,20(R10)), может быть перенесена во второй слот выдачи. Это улучшает время выполнения на несколько тактов, поскольку устраняет один слот выдачи команды и сокращает приостановку конвейера для последней команды последовательности.

Для успешного использования условных команд в примерах, подобных этому, семантика команды должна определять команду таким образом, чтобы не было никакого побочного эффекта, если условие не выполняется. Это означает, что если условие не выполняется, команда не должна записывать результат по месту назначения, а также не должна вызывать исключительную ситуацию. Как показывает вышеприведенный пример, способность не вызывать исключительную ситуацию достаточно важна: если регистр R10 содержит нуль, команда LW R8,20(R10), выполненная безусловно, возможно вызовет исключительную ситуацию по защите памяти, а эта исключительная ситуация не должна возникать. Именно эта вероятность возникновения исключительной ситуации не дает возможность компилятору просто перенести команду загрузки R8 через команду условного перехода. Конечно, если условие удовлетворено, команда LW все еще может вызвать исключительную ситуацию (например, ошибку страницы), и аппаратура должна воспринять эту исключительную ситуацию, поскольку она знает, что управляющее условие истинно.

Условные команды определенно полезны для реализации коротких альтернативных потоков управления. Тем не менее, полезность условных команд существенно ограничивается несколькими факторами:

1. Аннулируемые условные команды (т. е. команды, условие которых является ложным) все же отнимают определенное время выполнения. Поэтому перенос команды через команду условного перехода и превращение ее в условную будет замедлять программу всякий раз, когда перенесенная команда не будет нормально выполняться. Важное исключение из этого правила возникает, когда такты, используемые перенесенной невыполняемой командой, были бы в любом случае холостыми (как в вышеприведенном примере с суперскалярной обработкой). Перенос команды через команду условного перехода существенно базируется на предположении о направлении перехода. Условные команды упрощают реализацию такого переноса, но не устраняют время выполнения, которое будет затрачено при неправильном предположении.

2. Условные команды наиболее полезны, когда условие может быть вычислено заранее. Если условие и условный переход не могут быть отде-

лены друг от друга (из-за зависимости по данным при определении условия), то условная команда не поможет, хотя все еще может оказаться полезной, поскольку она задерживает момент времени, когда условие должно стать известным, почти до конца конвейера.

3. Использование условных команд ограничено, когда в поток управления вовлечено больше одной простой альтернативной последовательности команд. Например, при переносе команды через пару команд условного перехода необходимо, чтобы она оставалась зависимой от обоих условий, что требует либо спецификации в команде сразу двух условий (маловероятная возможность), либо вставки дополнительных команд для вычисления конъюнкции условий.

4. Условные команды могут давать некоторые потери скорости по сравнению с безусловными командами. Это может проявиться либо в большем количестве тактов, необходимых для выполнения таких команд, либо в уменьшении общей частоты синхронизации машины. Если условные команды являются более дорогими с точки зрения скорости выполнения, то их следует использовать осмысленно.

По этим причинам во многих современных архитектурах используется небольшое число условных команд (наиболее популярными являются команды условных пересылок), хотя некоторые из них включают условные версии большинства команд (табл. 24).

Таблица 24

Условные команды в современных архитектурах

Alpha	HP-PA	MIPS	PowerPC	SPARC
Условная пересылка	Любая команда типа регистр-регистр может аннулировать следующую команду, делая ее условной	Условная пересылка	Условная пересылка	Условная пересылка

Выполнение по предположению (speculation)

Поддерживаемое аппаратурой выполнение по предположению позволяет выполнить команду до момента определения направления условного перехода, от которого данная команда зависит. Это снижает потери, которые возникают при наличии в программе зависимостей по управлению. Чтобы понять, почему выполнение по предположению оказывается полезным, рассмотрим следующий простой пример программного кода, который реализует проход по связанному списку и инкрементирование каждого элемента этого списка:

```
for (p=head; p <> nil; *p=*p.next) {
*p.value = *p.value+1;
}
```

Подобно циклам for, с которыми мы встречались в более ранних разделах, разворачивание этого цикла не увеличит степени доступного параллелизма уровня команд. Действительно, каждая развернутая итерация будет содержать оператор if и выход из цикла. Ниже приведена последовательность команд в предположении, что значение head находится в регистре R4, который используется для хранения p, и что каждый элемент списка состоит из поля значения и следующего за ним поля указателя. Проверка размещается внизу так, что на каждой итерации цикла выполняется только один переход.

```
J looptest
start: LW R5,0(R4)
ADDI R5,R5,#1
SW 0(R4),R5
LW R4,4(R4)
looptest: BNEZ R4,start
```

Развернув цикл однажды можно видеть, что разворачивание в данном случае не помогает:

```
J looptest
start: LW R5,0(R4)
ADDI R5,R5,#1
SW 0(R4),R5
LW R4,4(R4)
BNEZ R4,end
LW R5,0(R4)
ADDI R5,R5,#1
SW 0(R4),R5
LW R4,4(R4)
looptest: BNEZ R4,start
end:
```

Даже прогнозируя направление перехода, мы не можем выполнять с перекрытием команды из двух разных итераций цикла, и условные команды в любом случае здесь не помогут. Имеются несколько сложных моментов для выявления параллелизма из этого развернутого цикла:

1. Первая команда в итерации цикла (LW R5,0(R4)) зависит по управлению от обоих условных переходов. Таким образом, команда не может выполняться успешно (и безопасно) до тех пор, пока мы не узнаем исходы команд перехода.

2. Вторая и третья команды в итерации цикла зависят по данным от первой команды цикла.

3. Четвертая команда в каждой итерации цикла (LW R4,4(R4)) зависит по управлению от обоих переходов и антизависит от непосредственно предшествующей ей команды SW.

4. Последняя команда итерации цикла зависит от четвертой. Вместе эти условия означают, что мы не можем совмещать выполнение никаких команд между последовательными итерациями цикла! Имеется небольшая возможность совмещения посредством переименования регистров либо аппаратными, либо программными средствами, если цикл развернут, так что вторая загрузка более не антизависит от SW и может быть перенесена выше.

В альтернативном варианте, при выполнении по предположению, что переход не будет выполняться, мы можем попытаться совместить выполнение последовательных итераций цикла. Действительно, это в точности то, что делает компилятор с планированием трасс. Когда направление переходов может прогнозироваться во время компиляции, и компилятор может найти команды, которые он может безопасно перенести на место перед точкой перехода, решение, базирующееся на технологии компилятора, идеально. Эти два условия являются ключевыми ограничениями для выявления параллелизма уровня команд статически с помощью компилятора. Рассмотрим развернутый выше цикл. Переход просто трудно прогнозируем, поскольку частота, с которой он является выполняемым, зависит от длины списка, по которому осуществляется проход. Кроме того, мы не можем безопасно перенести команду загрузки через переход, поскольку, если содержимое R4 равно nil, то команда загрузки слова, которая использует R4 как базовый регистр, гарантированно приведет к ошибке и обычно сгенерирует исключительную ситуацию по защите. Во многих системах значение nil реализуется с помощью указателя на неиспользуемую страницу виртуальной памяти, что обеспечивает ловушку (trap) при обращении по нему. Такое решение хорошо для универсальной схемы обнаружения указателей на nil, но в данном случае это не очень помогает, поскольку мы можем регулярно генерировать эту исключительную ситуацию, и стоимость обработки исключительной ситуации плюс уничтожения результатов выполнения по предположению будет огромной.

Чтобы преодолеть эти сложности, машина может иметь в своем составе специальные аппаратные средства поддержки выполнения по предположению. Эта методика позволяет машине выполнять команду, которая может быть зависимой по управлению, и избежать любых последствий выполнения этой команды (включая исключительные ситуации), если окажется, что в действительности команда не должна выполняться. Таким образом, выполнение по предположению, подобно условным командам, позволяет преодолеть два сложных момента, которые могут возникнуть при более раннем выполнении команд: возможность появления исключительной ситуации и ненужное изменение состояния машины, вызванное выполнением команды. Кроме того, механизмы выполнения по предположению позволяют выполнять команду даже до момента оценки условия ко-

мандой условного перехода, что невозможно при условных командах. Конечно, аппаратная поддержка выполнения по предположению достаточно сложна и требует значительных аппаратных ресурсов.

Один из подходов, который был хорошо исследован во множестве исследовательских проектов и используется в той или иной степени в машинах, которые разработаны или находятся на стадии разработки в настоящее время, заключается в объединении аппаратных средств динамического планирования и выполнения по предположению. В определенной степени подобную работу делала и IBM 360/91, поскольку она могла использовать средства прогнозирования направления переходов для выборки команд и назначения этих команд на станции резервирования. Механизмы, допускающие выполнение по предположению, идут дальше и позволяют действительно выполнять эти команды, а также другие команды, зависящие от команд, выполняющихся по предположению. Как и для алгоритма Томасуло, поясним аппаратное выполнение по предположению на примере устройства плавающей точки, но все идеи естественно применимы и для целочисленного устройства.

Аппаратура, реализующая алгоритм Томасуло, может быть расширена для обеспечения поддержки выполнения по предположению. С этой целью необходимо отделить средства пересылки результатов команд, которые требуются для выполнения по предположению некоторой команды, от механизма действительного завершения команды. Имея такое разделение функций, мы можем допустить выполнение команды и пересылать ее результаты другим командам, не позволяя ей однако делать никакие обновления состояния машины, которые не могут быть ликвидированы, до тех пор, пока мы не узнаем, что команда должна безусловно выполниться. Использование цепей ускоренной пересылки также подобно выполнению по предположению чтения регистра, поскольку мы не знаем, обеспечивает ли команда, формирующая значение регистра-источника, корректный результат до тех пор, пока ее выполнение не станет безусловным. Если команда, выполняемая по предположению, становится безусловной, ей разрешается обновить регистровый файл или память. Этот дополнительный этап выполнения команд обычно называется стадией фиксации результатов команды (instruction commit).

Главная идея, лежащая в основе реализации выполнения по предположению, заключается в разрешении неупорядоченного выполнения команд, но в строгом соблюдении порядка фиксации результатов и предотвращением любого безвозвратного действия (например, обновления состояния или приема исключительной ситуации) до тех пор, пока результат команды не фиксируется. В простом конвейере с выдачей одиночных ко-

манд мы могли бы гарантировать, что команда фиксируется в порядке, предписанном программой, и только после проверки отсутствия исключительной ситуации, вырабатываемой этой командой, просто посредством переноса этапа записи результата в конец конвейера. Когда мы добавляем механизм выполнения по предположению, мы должны отделить процесс фиксации команды, поскольку он может произойти намного позже, чем в простом конвейере. Добавление к последовательности выполнения команды этой фазы фиксации требует некоторых изменений в последовательности действий, а также в дополнительного набора аппаратных буферов, которые хранят результаты команд, которые завершили выполнение, но результаты которых еще не зафиксированы. Этот аппаратный буфер, который можно назвать буфером переупорядочивания, используется также для передачи результатов между командами, которые могут выполняться по предположению.

Буфер переупорядочивания предоставляет дополнительные виртуальные регистры точно так же, как станции резервирования в алгоритме Томасуло расширяют набор регистров. Буфер переупорядочивания хранит результат некоторой операции в промежутке времени от момента завершения операции, связанной с этой командой, до момента фиксации результатов команды. Поэтому буфер переупорядочивания является источником операндов для команд, точно также как станции резервирования обеспечивают промежуточное хранение и передачу операндов в алгоритме Томасуло. Основная разница заключается в том, что когда в алгоритме Томасуло команда записывает свой результат, любая последующая выдаваемая команда будет выбирать этот результат из регистрового файла. При выполнении по предположению регистровый файл не обновляется до тех пор, пока команда не фиксируется (и мы знаем определенно, что команда должна выполняться); таким образом, буфер переупорядочивания поставляет операнды в интервале между завершением выполнения и фиксацией результатов команды. Буфер переупорядочивания не похож на буфер записи в алгоритме Томасуло, и в нашем примере функции буфера записи интегрированы с буфером переупорядочивания только с целью упрощения. Поскольку буфер переупорядочивания отвечает за хранение результатов до момента их записи в регистры, он также выполняет функции буфера загрузки.

Каждая строка в буфере переупорядочивания содержит три поля: поле типа команды, поле места назначения (результата) и поле значения. Поле типа команды определяет, является ли команда условным переходом (для которого отсутствует место назначения результата), командой записи (которая в качестве места назначения результата использует адрес памяти) или регистровой операцией (команда АЛУ или команда загрузки, в кото-

рых местом назначения результата является регистр). Поле назначения обеспечивает хранение номера регистра (для команд загрузки и АЛУ) или адрес памяти (для команд записи), в который должен быть записан результат команды. Поле значения используется для хранения результата операции до момента фиксации результата команды. На рисунке 53 показана аппаратная структура машины с буфером переупорядочивания.

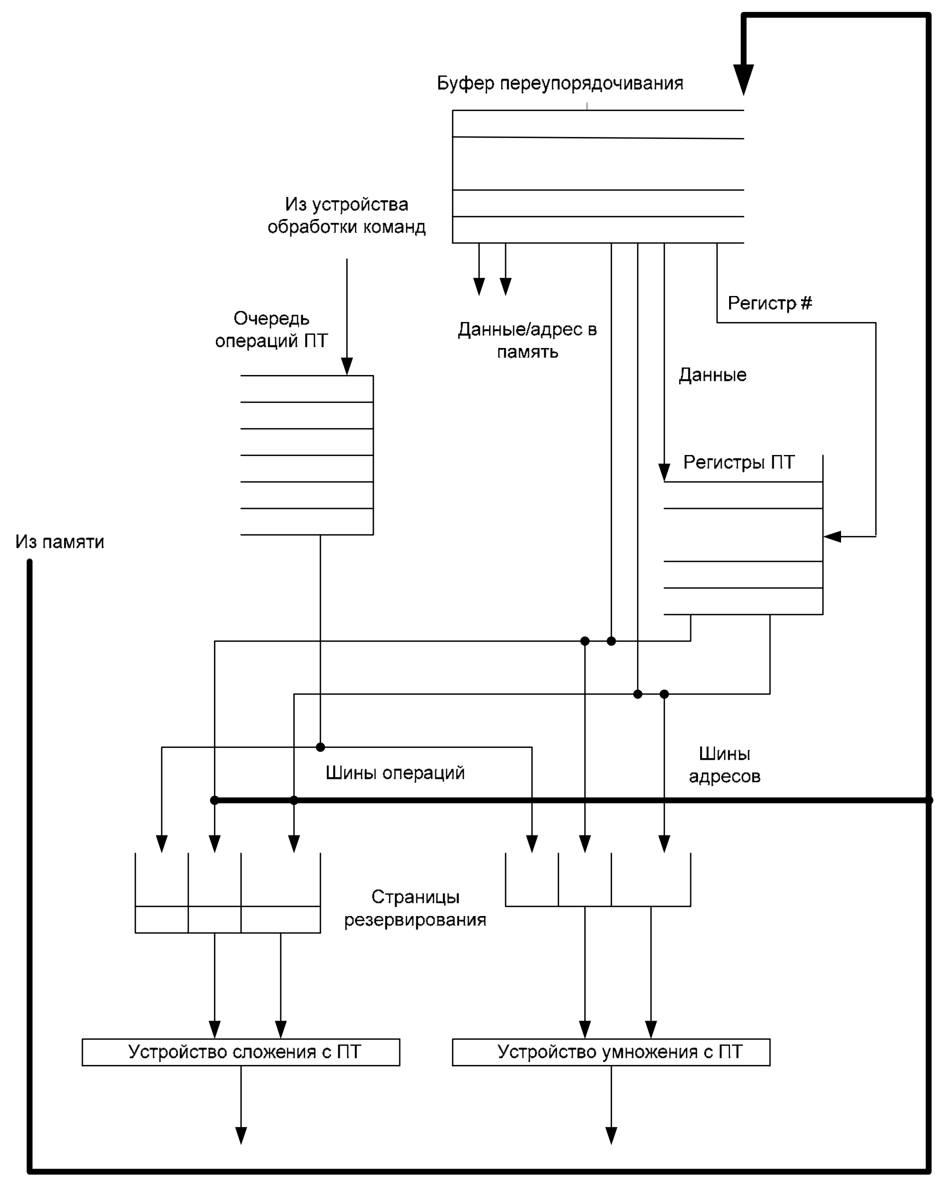


Рис. 53. Расширение устройства ПТ средствами выполнения по предположению

Буфер переупорядочивания полностью заменяет буфера загрузки и записи. Хотя функция переименования станций резервирования заменена буфером переупорядочивания, нам все еще необходимо некоторое место для буферизации операций (и операндов) между моментом их выдачи и

началом выполнения. Эту функцию выполняют регистровые станции резервирования. Поскольку каждая команда имеет позицию в буфере переупорядочивания до тех пор, пока она не будет зафиксирована (и результаты не будут отправлены в регистровый файл), результат тегируется посредством номера строки буфера переупорядочивания, а не номером станции резервирования. Это требует, чтобы номер строки буфера переупорядочивания, присвоенный команде, отслеживался станцией резервирования.

Ниже перечислены четыре этапа выполнение команды:

1. Выдача. Получает команду из очереди команд плавающей точки. Выдает команду для выполнения, если имеется свободная станция резервирования и свободная строка в буфере переупорядочивания; передает на станцию резервирования операнды, если они находятся в регистрах или в буфере переупорядочивания; и обновляет поля управления для индикации того, что буфера используются. Номер отведенной под результат строки буфера переупорядочивания также записывается в станцию резервирования, так что этот номер может использоваться для тегирования (пометки) результата, когда он помещается на CDB. Если все станции резервирования заполнены, или полон буфер переупорядочивания, выдача команды приостанавливается до тех пор, пока в обоих буферах не появится доступной строки.

2. Выполнение. Если один или несколько операндов еще не готовы (отсутствуют), осуществляется просмотр CDB (Common Data Bus) и происходит ожидание вычисления значения требуемого регистра. На этом шаге выполняется проверка наличия конфликтов типа RAW. Когда оба операнда оказываются на станции резервирования, происходит вычисление результата операции.

3. Запись результата. Когда результат вычислен и становится доступным, выполняется его запись на CDB (с посылкой тега буфера переупорядочивания, который был присвоен команде на этапе выдачи для выполнения) и из CDB в буфер переупорядочивания, а также в каждую станцию резервирования, ожидающую этот результат. (Можно было бы также читать результат из буфера переупорядочивания, а не из CDB, точно так же, как централизованная схема управления (scoreboard) читает результаты из регистров, а не с шины завершения). Станция резервирования помечается как свободная.

4. Фиксация. Когда команда достигает головы буфера переупорядочивания и ее результат присутствует в буфере, соответствующий регистр обновляется значением результата (или выполняется запись в память, если операция – запись в память), и команда изымается из буфера переупорядочивания.

Когда команда фиксируется, соответствующая строка буфера переупорядочивания очищается, а место назначения результата (регистр или ячейка памяти) обновляется. Чтобы не менять номера строк буфера пере-

упорядочивания после фиксации результата команды, буфер переупорядочивания реализуется в виде циклической очереди, так что позиции в буфере переупорядочивания меняются, только когда команда фиксируется. Если буфер переупорядочивания полностью заполнен, выдача команд останавливается до тех пор, пока не освободится очередная строка буфера.

Поскольку никакая запись в регистры или ячейки памяти не происходит до тех пор, пока команда не фиксируется, машина может просто ликвидировать все свои выполненные по предположению действия, если обнаруживается, что направление условного перехода было спрогнозировано не верно.

Рассмотрим следующий пример:

```
LD F6, 34(R2)
LD F2, 45(R3)
MULTD F0, F2, F4
SUBD F8, F6, F2
DIVD F10, F0, F6
ADDD F6, F8, F2
```

Представим, что в приведенном выше примере команда условного перехода BNEZ в первый раз не выполняется. Тогда команды, предшествующие команде условного перехода, будут просто фиксироваться по мере достижения каждой из них головы буфера переупорядочивания. Когда головы этого буфера достигает команда условного перехода, содержимое буфера просто гасится, и машина начинает выборку команд из другой ветви программы.

Исключительные ситуации в подобной машине не воспринимаются до тех пор, пока соответствующая команда не готова к фиксации. Если выполняемая по предположению команда вызывает исключительную ситуацию, эта исключительная ситуация записывается в буфер упорядочивания. Если обнаруживается неправильный прогноз направления условного перехода и выясняется, что команда не должна была выполняться, исключительная ситуация гасится вместе с командой, когда обнуляется буфер переупорядочивания. Если же команда достигает вершины буфера переупорядочивания, то мы знаем, что она более не является выполняемой по предположению (она уже стала безусловной), и исключительная ситуация должна действительно восприниматься.

Эту методику выполнения по предположению легко распространить и на целочисленные регистры и функциональные устройства. Действительно, выполнение по предположению может быть более полезно в целочисленных программах, поскольку именно такие программы имеют менее предсказуемое поведение переходов. Кроме того, эти методы могут быть расширены так, чтобы обеспечить работу в машинах с выдачей на выполнение и фиксацией результатов нескольких команд в каждом такте. Выполнение по предположению, возможно, является наиболее интересным

методом именно для таких машин, поскольку менее амбициозные машины могут довольствоваться параллелизмом уровня команд внутри базовых блоков при соответствующей поддержке со стороны компилятора, использующего технологию разворачивания циклов.

Очевидно, все рассмотренные ранее методы не могут достичь большей степени распараллеливания, чем заложено в конкретной прикладной программе. Вопрос увеличения степени параллелизма прикладных систем в настоящее время является предметом интенсивных исследований, проводимых во всем мире.

11. Архитектура EPIC

EPIC: явный параллелизм команд. Концепция реализации параллелизма на уровне команд (Explicitly Parallel Instruction Computing) определяет новый тип архитектуры, способной конкурировать по масштабам влияния с RISC. Эта идеология направлена на то, чтобы упростить аппаратное обеспечение и, в то же время, извлечь как можно больше «скрытого параллелизма» на уровне команд, используя большую ширину «выдачи» команд (WIW -Wide Issue-Width) и длинные (глубокие) конвейеры с большой задержкой (DPL – Deep Pipeline-Latency), чем это можно сделать при реализации VLIW или суперскалярных стратегий. EPIC упрощает два ключевых момента, реализуемых во время выполнения. Во-первых, его принципы позволяют во время исполнения отказаться от проверки зависимостей между операциями, которые компилятор уже объявил как независимые. Во-вторых, данная архитектура позволяет отказаться от сложной логики внеочередного исполнения операций, полагаясь на порядок выдачи команд, определенный компилятором. Более того, EPIC совершенствует возможность компилятора статически генерировать планы выполнения за счет поддержки разного рода перемещений кода во время компиляции, которые были бы некорректными в последовательной архитектуре. Более ранние решения достигали этой цели главным образом за счет серьезного увеличения сложности аппаратного обеспечения, которое стало настолько значительным, что превратилась в препятствие, не позволяющее отрасли добиваться еще более высокой производительности. EPIC разработан именно для того, чтобы обеспечить более высокую степень параллелизма на уровне команд, поддерживая при этом приемлемую сложность аппаратного обеспечения.

Более высокая производительность достигается как за счет увеличения скорости передачи сигналов, так и благодаря увеличению плотности расположения функциональных устройств на кристалле. Зафиксировав рост этих двух составляющих, дальнейшего увеличения скорости выпол-

нения программ можно добиться в первую очередь благодаря реализации определенного вида параллелизма. Так, параллелизм на уровне команд (ILP – Instruction-Level Parallelism) стал возможен благодаря созданию процессоров и методик компиляции, которые ускоряют работу за счет параллельного выполнения отдельных RISC-операций. Системы на базе ILP используют программы, написанные на традиционных языках высокого уровня, для последовательных процессоров, а обнаружение «скрытого параллелизма» автоматически выполняется благодаря применению соответствующей компиляторной технологии и аппаратного обеспечения.

Тот факт, что эти методики не требуют от прикладных программистов дополнительных усилий, имеет крайне важное значение, поскольку данное решение резко отличается от традиционного микропроцессорного параллелизма, который предполагает, что программисты должны переписывать свои приложения. Параллельная обработка на уровне команд является единственным надежным подходом, позволяющим добиться увеличения производительности без фундаментальной переработки приложения.

Суперскалярные процессоры – это реализации ILP-процессора для последовательных архитектур, программа для которых не должна передавать и, фактически, не может передавать точную информацию о параллелизме. Поскольку программа не содержит точной информации о наличии ILP, задача обнаружения параллелизма должна решаться аппаратурой, которая, в свою очередь, должна создавать план действий для обнаружения «скрытого параллелизма». Процессоры VLIW представляют собой пример архитектуры, для которой программа предоставляет точную информацию о параллелизме – компилятор выявляет параллелизм в программе и сообщает аппаратному обеспечению какие операции не зависят друг от друга. Эта информация имеет важное значение для физического слоя, поскольку в этом случае он «знает» без дальнейших проверок какие операции можно начинать выполнять в одном и том же такте. Архитектура EPIC – это эволюция архитектуры VLIW, которая абсорбировала в себе многие концепции суперскалярной архитектуры, хотя и в форме, адаптированной к EPIC. По сути – это «идеология», определяющая, как создавать ILP-процессоры, а также набор характеристик архитектуры, которые поддерживают данную основу. В таком смысле EPIC похож на RISC: определяющий класс архитектур, подчиняющихся общим основным принципам. Точно так же, как существует множество различных архитектур наборов команд (ISA) для RISC, может существовать и больше одной ISA для EPIC. В зависимости от того, какие из характеристик EPIC использует архитектура EPIC ISA, она может быть оптимизирована для различных приложений – например, для систем общего назначение или встроенных устройств. Первым примером коммерческой EPIC ISA стала архитектура IA-64.

Код для суперскалярных процессоров содержит последовательность команд, которая порождает корректный результат, если выполняется в установленном порядке. Код указывает последовательный алгоритм и, за исключением того, что он использует конкретный набор команд, не представляет себе точно природу аппаратного обеспечения, на котором он будет работать или точный временной порядок, в котором будут выполняться команды. В отличие от программ для суперскалярных процессоров, код VLIW предлагает точный план (POE – Plan Of Execution, схема исполнения создается статически во время компиляции) того, как процессор будет выполнять программу. Код точно указывает, когда будет выполнена каждая операция, какие функциональные устройства будут работать, и какие регистры будут содержать операнды. Компилятор VLIW создает такой план выполнения, имея полное представление о самом процессоре, чтобы добиться требуемой записи исполнения (ROE – Record Of Execution) – последовательности событий, которые действительно происходят во время работы программы. Компилятор передает POE (через архитектуру набора команд, которая точно описывает параллелизм) аппаратному обеспечению, которое, в свою очередь, выполняет указанный план. Этот план позволяет VLIW использовать относительно простое аппаратное обеспечение, способное добиться высокого уровня ILP. В отличие от VLIW, суперскалярная аппаратура динамически строит POE на основе последовательного кода. Хотя такой подход и увеличивает сложность физической реализации, суперскалярный процессор создает план, используя преимущества тех факторов, которые могут быть определены только во время выполнения.

Одна из целей, которые ставили перед собой при создании EPIC, состояла в том, чтобы сохранить реализованный во VLIW принцип статического создания POE, но и в то же время обогатить его возможностями, аналогичными возможностям суперскалярного процессора, позволяющими новой архитектуре лучше учитывать динамические факторы, традиционно ограничивающие параллелизм, свойственный VLIW. Чтобы добиться этих целей, «идеология» EPIC была построена на некоторых основных принципах. Первый — это создание плана выполнения во время компиляции. EPIC возлагает нагрузку по созданию POE на компилятор. Хотя, в общем, архитектура и физическая реализация могут препятствовать компилятору в выполнении этой задачи, процессоры EPIC предоставляют функции, которые помогают компилятору создавать план выполнения. Во время исполнения поведение процессора EPIC с точки зрения компилятора должно быть предсказуемым и управляемым. Динамическое внеочередное исполнение команд может «запутать» компилятор так, что он не будет «понимать», как его решения повлияют на реальную запись выполнения, созданную процессором, поэтому ему необходимо уметь предсказывать действия

процессора, что еще больше усложняет задачу. В данной ситуации предпочтителен процессор, четко исполняющий то, что ему указывает программа. Суть же создания плана во время компиляции состоит в переупорядочивании исходного последовательного кода так, чтобы использовать все преимущества параллелизма приложения и максимально эффективно тратить аппаратные ресурсы, минимизируя время выполнения. Без соответствующей поддержки архитектуры такое переупорядочивание может нарушить корректность программы. Таким образом, поскольку EPIC возлагает создание POE на компилятор, она должна обеспечивать еще и архитектурные возможности, поддерживающие интенсивное переупорядочивание кода во время компиляции.

Следующим принципом является использование компилятором вероятностных оценок. Компилятор EPIC сталкивается с серьезной проблемой при создании плана выполнения: информация определенного типа, которая существенно влияет на запись исполнения, становится известна только лишь в момент выполнения программы. Например, компилятор не может точно знать, какая из ветвей после оператора перехода будет выполняться, когда запланированный код пройдет базовые блоки, и какой из путей графа будет выбран. Кроме того, обычно невозможно создать статический план, который одновременно оптимизирует все пути в программе. Неоднозначность также возникает и в тех случаях, когда компилятор не может решить, будут ли ссылки указывать на одно и то же место в памяти. Если да, то обращение к ним должно осуществляться последовательно; если нет, то их можно запланировать в произвольном порядке. При такой неоднозначности часто наиболее вероятен некий конкретный результат. Одним из важнейших принципов EPIC в данной ситуации является возможность разрешения компилятору оперировать вероятностными оценками – он создает и оптимизирует POE для наиболее вероятных случаев. Однако EPIC обеспечивает архитектурную поддержку, такую как спекулятивное выполнение по управлению и по данным (Control and Data Speculation), с тем, чтобы гарантировать корректность программы, даже если исходные предположения были не верны. Когда предположение оказывается неверным, совершенно очевидно падение производительности при выполнении программы. Такой эффект производительности иногда виден на плане программы, к примеру, в тех случаях, когда существует высоко оптимизированная программная область, а код исполняется в менее оптимизированной. Также падение производительности может возникнуть в моменты «остановки» (Stall), которые на плане программы не видны – определенные операции, подпадающие под наиболее вероятный и, следовательно, оптимизированный случай, выполняются при максимальной производительности, но приостанавливают процессор для того, чтобы гарантировать корректность, если возникнет менее вероятный, не оптимизированный случай.

После того, как создан план, компилятор передает его аппаратному обеспечению. Для этого ISA должен обладать возможностями достаточно богатыми, чтобы сообщить решения компилятора о том, когда инициировать каждую операцию и какие ресурсы использовать (в частности, должен существовать способ указать, какие операции инициируются одновременно). В качестве альтернативного решения компилятор мог бы создавать последовательную программу, которую процессор динамически реорганизует с тем, чтобы получить требуемую запись. Но в таком случае цель, сводимая к освобождению аппаратного обеспечения от динамического планирования, не достигается. При передаче POE аппаратному обеспечению крайне важно своевременно предоставить необходимую информацию. Примером этому может служить операция перехода, которая, в случае ее использования, требует, чтобы по адресу перехода команды выбирались с упреждением, заведомо до того, как будет инициирован сам переход. Вместо того, чтобы решение о том, когда это нужно сделать и какой адрес перехода отдавать на откуп аппаратному обеспечению, такая информация в соответствии с основными принципами EPIC передается аппаратному обеспечению точно и своевременно через код. Микроархитектура принимает и другие решения, не связанные напрямую с выполнением кода, но которые влияют на время выполнения. Один из таких примеров – управление иерархией кэш-памяти и соответствующие решения о том, какие данные нужны для поддержки иерархии, а какие следует заменить. Такие правила обычно предусматриваются алгоритмом функционирования контроллера кэша. EPIC расширяет принцип, утверждающий, что план выполнения компилятор создает так, чтобы тоже иметь возможность управлять этими механизмами микроархитектуры. Для этого обеспечиваются архитектурные возможности, позволяющие осуществлять программный контроль механизмами, которыми обычно управляет микроархитектура.

Архитектура EPIC представляет собой пример научно-исследовательской разработки, представляющей собой скорее принцип обработки информации, нежели архитектуру конкретного процессора. В архитектуре сочетаются многие технологические решения, довольно разнотипные, которые в результате сочетания дают значительное повышение скорости обработки и решение некоторых проблем трансляции программ.

Архитектура EPIC (базовые принципы) были разработаны в университете Иллинойса, проект имел название Impact. В начале 1990-х годов были заложены теоретические основы самой архитектуры, затем были начаты работы в рамках создания инструментальных средств для процессора EPIC. Проект по созданию инструментальных средств известен под названием Trimaran. Архитектура EPIC имеет следующие основные особенно-

сти: (мы не будем обсуждать конкретные параметры конвейеров процессора, так как EPIC является в большей степени концептуальной моделью, чем типовым образцом для тиражирования процессоров):

1) поддержка явно выделенного компилятором параллелизма. Формат команд имеет много общего с архитектурой с длинным командным словом – параллелизм так же явно выделен. Однако, если длинное командное слово имеет обычно четко заданную ширину (хотя в процессорах с нерегулярным длинным командным словом слова могут быть разными), в процессоре EPIC существует некоторое количество образцов длинных команд, в которых функциональным устройствам процессора явно сопоставлена операция. Кроме того, ряд образцов может сопоставлять инструкции из различных тактов выполнения, т. е. инструкция явно выполняется за 1-2 такта, в этом случае в образце инструкции явно специфированы между какими командами имеется слот задержки (фактически «граница» между командами, выполняемыми в разных тактах). К каждой длинной команде (128 бит) прилагается небольшой (3-5 битный) ярлык, который специфицирует формат команды;

2) наличие большого регистрового файла;

3) наличие предикатных регистров, подобно архитектуре PowerPC. Предикатные регистры позволяют избавиться от известной проблемы с неразделяемым ресурсом регистра флагов большинства процессоров, поддерживающих скалярный параллелизм – программная конвееризация циклов, содержащих условные переходы, практически невозможна. Множественный предикатный флаговый регистр позволяет избавится от паразитных связей по управлению из-за регистра флагов;

4) спекулятивная загрузка данных, позволяющая избежать простоев конвейера при загрузке данных из оперативной памяти;

5) поддерка предикатно-выполняемых команд, которая позволяет: а) избежать излишних инструкций ветвления, если количество команд в ветвях условного оператора невелико; б) уменьшить нагрузку на устройство предсказания переходов;

6) аппаратная поддержка програмной конвейеризации с поомшью механизма переименования регистров;

7) используется стек регистров (и регистровые окна);

8) для предсказания инструкций используется поддержка компилятора;

9) имеется специальная поддержка програмной конвейеризации (метод составления расписания команд «по модулю») циклов, когда на каждой итерации имена регистров сдвигаются и каждая новая итерация оперирует уже с новым набором регистров;

10) введена поддержка инструкций циклического выполнения команд без потерь времени на инструкции циклического выполнения.

Наиболее интересной в архитектуре EPIC является поддержка спекулятивного выполнения команд и загрузки данных.

Спекулятивное исполнение команд. Большинство команд загрузки данных из памяти выполняется длительное время. Выполнение команды за 1-2 такта возможно только в том случае, если значение содержится в кеш-памяти второго уровня. Время несколько увеличивается, если значение находится в кеш-памяти второго уровня. В случае чтения же данных из микросхем динамической памяти даже при попадании на активную страницу происходит значительная задержка при загрузке, а при смене страницы задержка имеет астрономическую величину, причем конвейер быстро блокируется, так команд, которые можно выполнить без нарушения зависимости по данным, обычно оказывается крайне мало.

При спекулятивном выполнении используется вынесение команд загрузки далеко вперед инструкций, использующих эти данные, в основном вверх за инструкции условного перехода. При достижении потоком команд места, где необходимо использовать загружаемые данные, вставляется инструкция, проверяющая не произошло ли исключения в процессе загрузки (например, какая-то инструкция произвела запись по этому адресу), если исключение произошло, то вызывается специально написанный восстановительный код, который попросту перезагружает значение в регистр.

При спекулятивных операциях по данным оптимизируется часто встречающийся участок во многих программах. Рассмотрим фрагмент программы:

```
int *p,*t;
int a,b;
*p = b;
a = *t;
```

В случае, если компилятор не имеет достаточно информации для проведения анализа потока данных, он не может определить, перекрываются ли области памяти, адресуемые указателями p и t. В этом случае каждый раз значение надо перезагружать, но, мало того, инструкции загрузки нельзя переупорядочить. Для получения возможности переупорядочивания этих инструкций используется спекуляция по данным. Инструкция загрузки переносится наверх, при этом, если произведена запись, затрагивающая загружаемое содержимое, то производится вызов кода восстановления, перезагружающего значение.

МОДУЛЬ III. МИКРОКОНТРОЛЛЕРЫ И СПЕЦИАЛИЗИРОВАННЫЕ МИКРОПРОЦЕССОРЫ

1. Микроконтроллеры. 8-разрядные микроконтроллеры. 16- и 32-разрядные микроконтроллеры. Структурная организация. Построение микропроцессорной системы на базе микроконтроллеров. Особенности программирования

8-разрядные микроконтроллеры. Фирма Motorola предлагает самую широкую в мире номенклатуру микроконтроллеров, охватывающую практически все области применения и включающую в себя около 300 моделей: от простейших дешевых до высокопроизводительных 32-разрядных микроконтроллеров с RISC-ядром и мощной периферией. Как следствие, пользователь имеет возможность выбрать для своего приложения оптимальную модель микроконтроллера как по набору встроенных функций, минимизируя число компонентов в системе, так и по экономическим параметрам, соответствующим объему и особенностям производства.

Второй важной особенностью микроконтроллеров (и остальной продукции) фирмы Motorola является их высокое качество и надежность. Являясь традиционным поставщиком военно-промышленного и аэрокосмического комплексов, а также автомобильной промышленности США, предъявляющих повышенные требования к надежности компонентов, фирма Motorola выработала и продолжает развивать специальную программу повышения качества продукции. Заслуги фирмы в этой области отмечены национальной наградой США «за высшее качество продукции», а также многочисленными наградами фирме как лучшему поставщику от таких крупных компаний, как General Motors, Ford, Chrysler, Bosch и др.

Семейство HC05. Семейство HC05 содержит наибольшее количество модификаций микроконтроллеров (около 180), поскольку это семейство в немалой степени формировалось крупными потребителями фирмы Motorola, заказывающими разработку микроконтроллеров нужной конфигурации под свою конкретную продукцию. Поэтому семейство HC05 иногда называют семейством «заказных» микроконтроллеров.

Областями применения семейства HC05 являются самые разнообразные устройства связи, автомобильной и бытовой электроники, промышленного управления, компьютерной периферии.

Все микроконтроллеры этого семейства имеют одинаковое 8-разрядное процессорное ядро, основанное на популярной процессорной архитектуре 6800, и отличаются набором периферийных функций. Это означает, что применение любого микроконтроллера этого семейства открывает

пользователю возможность использовать приобретенный опыт при создании новых устройств как с применением других микроконтроллеров из обширного семейства HC05, так и на основе более производительного, но программно совместимого семейства HC08.

В состав семейства HC05 входят: ПЗУ всех типов, ОЗУ, таймеры, АЦП, ШИМ, контроллеры ЖКИ и других дисплеев, последовательные интерфейсы и многие другие устройства. Все представители семейства HC05 имеют версии с пониженным питанием и расширенным температурным диапазоном и выпускаются в самых разнообразных корпусах.

Семейство HC08. Семейство HC08 является следующим шагом в развитии заказных микроконтроллеров фирмы Motorola для массовых приложений и характеризуется повышенной в 5-10 раз производительностью процессорного ядра, совместимого по системе команд с ЦПУ HC05. Семейство HC08 поддерживает дополнительные эффективные команды и методы адресации, а также такие новые функции, как прямой доступ к памяти, технология «нечеткой логики» и элементы цифровой обработки сигналов.

При этом полностью статическое процессорное ядро оптимизировано для работы с пониженным напряжением питания и позволяет гибко управлять потреблением с помощью встроенного синтезатора тактовой частоты. Семейство HC08 является первым 8-разрядным семейством с определяемой пользователем архитектурой на базе набора стандартных модулей, что значительно ускоряет цикл разработки нового заказного микроконтроллера.

Набор модулей в настоящее время включает в себя различные типы ПЗУ и ОЗУ, таймеры, последовательные интерфейсы, АЦП, контроллер ЖКИ, контроллер ПДП, силовые и высоковольтные ключи и т. д.

Первые представители этого семейства появились в 1994 г., в настоящий момент в состав семейства входят около 20 моделей. Новая программа «Заказной микроконтроллер за 7 дней», введенная фирмой Motorola в 1996 г., позволила радикально сократить цикл разработки новых микроконтроллеров семейства HC08, что безусловно приведет к его динамичному развитию.

Семейство HC11. В отличие от относительно специализированных микроконтроллеров «заказных» семейств семейство MC68HC11 содержит набор примерно из 40 более универсальных и высокопроизводительных микроконтроллеров, ориентированных как на массовые рынки, так и на среднее и мелкое производство. Процессорное ядро микроконтроллеров этого семейства имеет повышенную производительность, отличается от HC05 более эффективной архитектурой, системой команд, наличием дополнительных методов адресации и возможностью адресовать больший объем внешней памяти. Микроконтроллеры семейства HC11 содержат встроенную память различных типов и конфигураций.

Периферийные функции представлены многофункциональными таймерами, АЦП (до 12 каналов и 10 разрядов), встроенным сопроцессором, ускоряющим выполнение умножения и деления на порядок, ШИМ и ЦАП; последовательными интерфейсами, контроллером ПДП, синтезатором тактовой частоты и другими функциями. Как и в других семействах, имеется большое разнообразие корпусов, а также версии с пониженным напряжением питания и расширенным температурным диапазоном.

Процессор и система команд семейства HC05. Процессор семейства HC05, отличающееся простотой и удобством программирования, имеет стандартную внутреннюю тактовую частоту 2 Мгц, для некоторых МК существуют версии с тактовой частотой 4 Мгц (цикл команды 250 нс). Программная модель процессора содержит 5 регистров (рис. 54), которые не являются частью карты памяти.

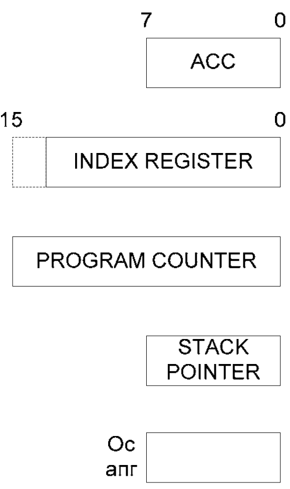


Рис. 54. Программная модель микроконтроллера семейства HC05

Аккумулятор (АСС) – это 8-битный регистр общего назначения, в котором хранятся операнды, результаты арифметических операций, а также данные, с которыми производятся какие-либо операции. Аккумулятор используется также и для логических операций.

Индексный регистр (X) используется либо при индексном режиме адресации, либо как вспомогательный аккумулятор. Этот регистр может быть загружен как непосредственно, так и из памяти, может быть сохранен в ячейке или сравнен с ее содержимым.

Программный счетчик (PC) содержит адрес команды, следующей за выполняемой, либо адрес операнда, входящего в код программы. Разрядность PC зависит от объема встроенного ПЗУ.

Указатель стека (SP) содержит адрес следующей (свободной) ячейки стека. Глубина стека МК семейства HC05 составляет 64 байта. Вызов подпрограммы использует 2 ячейки стека, прерывание – 5 ячеек.

Регистр признаков (CC) содержит 5 флагов, устанавливающихся в зависимости от результата выполнения арифметических и других операций. Этими флагами являются: флаг полупереноса (H), флаг отрицательного результата (N), флаг нулевого результата (Z), бит прерываний (I) и флаг переноса (C).

Система команд микроконтроллера включает в себя 65 команд, делящихся на следующие группы:

1) команды перемещения данных (LDA, STA, CLR, LDX, STX ...);
2) команды передачи управления (JMP, JSR, RTI, переходы по условиям и битам, …);
3) арифметические команды (ADD, SUB, MUL, ...);
4) логические команды (AND, OR, COM, NEG, ...);
5) команды работы с битами (BSET, BCLR, сдвиги, ...);
6) специальные команды (WAIT, STOP, SWI, ...).

Команды MC68HC05 выполняются, как правило, за 2...5 циклов внутренней тактовой частоты, что составляет 1...2,5 мкс при стандартной внутренней тактовой частоте 2 Мгц.

Микроконтроллеры семейства MC68HC05 используют восемь режимов адресации: неявная, непосредственная, прямая, расширенная, индексная без смещения, индексная с 8-разрядным смещением, индексная с 16-разрядным смещением, относительная.

Встроенная память МК семейства HC05. Встроенная память микроконтроллера семейства HC05 может включать в себя ПЗУ (масочное, однократно программируемое, программируемое с УФ-стиранием, программируемое с электрическим стиранием) и ОЗУ объемом до 768 байт.

В карту памяти включены регистры портов параллельного ввода/вывода (причем, как правило, эти адреса одинаковы для всех моделей семейства), а также адреса управляющих регистров и регистров данных периферийных устройств (таймера, последовательного интерфейса и т. д.). Область загрузочного ПЗУ является масочной и позволяет осуществлять «самозагрузку» данных во встроенную память (включая программирование ППЗУ) через параллельные или последовательные порты. Наконец, векторы прерываний, располагающиеся в ППЗУ, определяют адреса переходов по прерываниям от периферийных подсистем (таймера, последовательных интерфейсов, вывода внешнего прерывания) и в других случаях (RESET, программное прерывание).

Встроенные подсистемы МК семейства HC05. Ниже приведено описание некоторых встроенных подсистем, встречающихся в наибольшем количестве микроконтроллеров семейства HC05.

Основу блока таймера составляет 16-битный счетчик с предделителем, имеющий возможность формировать прерывание по переполнению и синхронизируемый внутренней тактовой частотой, деленной на 2.

Большинство микроконтроллеров семейства содержат также связанные со счетчиком подсистемы входной фиксации (IC) и выходного сравнения (OC). Система IC служит для обработки временных параметров внешних сигналов и позволяет записывать в регистр IC содержимое счетчика при перепаде уровня внешнего сигнала, с выдачей соответствующего прерывания или установкой флага. Система OC предназначена для генерации импульсного сигнала с программируемыми временными параметрами и позволяет выдавать в линию порта “0” или “1” в момент равенства содержимого счетчика и содержимого регистра OC. В наиболее простых моделях семейства функции IC и OC могут отсутствовать и заменяться прерываниями реального времени (RTI) с программируемым интервалом между прерываниями. Еще одной важной системой, связанной с таймером, является система слежения за выполнением программы (COP WatchDog). Эта система формирует RESET, если периодически с определенным промежутком времени не будет произведена запись в специальный регистр.

Последовательный интерфейс связи SCI представляет собой полнодуплексный асинхронный приемопередатчик и может быть использован для связи с терминалом, ПЭВМ (например, RS-232) или другими микроконтроллерами. Встроенный генератор частоты обмена позволяет делить внутреннюю тактовую частоту с получением 32 стандартных частот обмена – от 37,56 бод до 125 кбод. SCI также поддерживает такие функции, как программируемая длина посылки, выходы из режима ожидания приемника по свободной линии или адресному маркеру, отдельное разрешение приемника и передатчика, обнаружение ошибки кадра и шума в линии (с временным разрешением 1/16 бита). SCI может формировать пять видов прерываний (или устанавливать пять флагов) по следующим условиям: «регистр данных передатчика пуст», «передача завершена», «регистр данных приемника заполнен», «приемник переполнен» и «линия приема свободна».

Последовательный периферийный интерфейс SPI используется для синхронной передачи информации в последовательном коде на меньшие расстояния, но со значительно большей скоростью. SPI позволяет микроконтроллеру взаимодействовать с различными периферийными устройствами, от сдвигового регистра до подсистемы ЖКИ дисплея или внешнего АЦП. SPI поддерживает следующие функции: полный дуплекс; режим ведущего и ведомого; четыре программируемые тактовые частоты до 1,05 Мгц с программируемой полярностью и фазой; флаг прерывания по окончании передачи; защита от конфликтов на магистрали.

Аналого-цифровой преобразователь производит преобразование внешнего напряжения в диапазоне от Vss (нижнее опорное напряжение, подключаемое к общей шине) до Vrh (верхнее опорное напряжение) в 8-раз-

рядный код от $00 до $FF, соответственно. АЦП использует метод последовательных приближений, процесс преобразования занимает 32 цикла внутренней тактовой частоты (16 мкс при 2 Мгц). При тактовой частоте меньшей, чем 1 Мгц, используется встроенный RC-генератор частоты для АЦП (1,5 Мгц). Встроенный мультиплексор позволяет проводить преобразование по одному из внешних аналоговых входов (до 8), а также измерять Vss, Vrh, (Vrh+Vss)/2 для проведения контроля и юстировки. Подсистема АЦП содержит регистр управления (задает режим работы и запуск преобразования), регистр статуса (содержит флаг окончания преобразования) и регистр данных (результат преобразования).

Семейство HC08 является следующим шагом в развитии концепции заказных микроконтроллеров фирмы Motorola для массовых приложений, получившей заслуженное признание у пользователей во всем мире. Основными движущими силами возникновения нового заказного семейства явились необходимость повышения производительности ЦПУ, а также резкого сокращения цикла разработки нового заказного микроконтроллера. Разработанное для удовлетворения этих потребностей рынка семейство HC08 характеризуется повышенной в 5-10 раз производительностью процессорного ядра, совместимого по системе команд с ЦПУ HC05. Семейство HC08 поддерживает дополнительные эффективные команды и методы адресации, а также такие новые функции, как прямой доступ к памяти, технология «нечеткой логики» и элементы цифровой обработки сигналов. Одной из важных особенностей семейства HC08 является то, что все модели адресуют внешнюю память, что существенно упрощает отладку программ. Полностью статическое процессорное ядро оптимизировано для работы с пониженным напряжением питания и позволяет гибко управлять потреблением с помощью встроенного синтезатора тактовой частоты. Семейство HC08 является первым 8-разрядным семейством с определяемой пользователем архитектурой на базе набора стандартных модулей, что значительно ускоряет цикл разработки нового заказного МК.

Ниже кратко рассмотрены особенности некоторых модулей, входящих в состав микроконтроллеров семейства HC08.

Особенности центрального процессора (CPU08):

1) тактовая частота 8 Мгц (цикл 125 нс);

2) 16-разрядные: индексный регистр, программный счетчик и указатель стека;

3) аппаратная поддержка ПДП, точек останова;

4) быстрые операции умножения и деления;

5) 64К байт адресуемой памяти с возможностью расширения;

6) полностью статическая архитектура, низкое потребление, пониженное питание.

Модуль прямого доступа к памяти (DMA08) обеспечивает скоростной обмен между памятью и внешними устройствами без участия процессора. DMA08 может обслуживать последовательный интерфейс (прием и передача), таймер либо обеспечивать передачу блоков данных до 256 байт. Каждому из трех независимых каналов назначается адрес источника данных, адрес приемника данных и схема изменения адреса после каждой передачи (инкремент, декремент или прежнее значение). Специальный режим позволяет проводить автоматическую реинициализацию параметров ПДП после окончания пересылки. DMA08 дает возможность задавать предельную часть времени, занимаемую ПДП: 25, 50, 67 или 100%, а также распределять приоритеты прерываний, обрабатываемых ПДП и ЦПУ, что позволяет сбалансировать их взаимодействие. Практические исследования показали, что использование ПДП дает выигрыш по производительности до 5 раз при пересылке массива из памяти в память и до 7 раз при пересылке из последовательного порта в память. Варианты использования DMA08 практически безграничны: например он может быть задействован для автоматической инициализации регистра таймера при генерации ШИМ или для обновления информации на ЖКИ, подключенном через синхронный последовательный порт.

Модуль **таймера (TIM08)** представляет собой гибкое устройство для решения разнообразных задач, связанных с обработкой временных интервалов. Таймер выпускается в различных модификациях и может иметь до 2, 4 или 6 независимых каналов, каждый из которых содержит 16-битный счетчик с программируемым предделителем, регистры входной фиксации, выходного сравнения и ШИМ.

Счетчики TIM08 отличаются от счетчиков HC05 возможностью подстройки временной базы с помощью функций останова и сброса, а также возможностью внешнего тактирования. Пары каналов таймера могут быть объединены для организации буферированного ШИМ.

Модули **последовательного обмена** представлены универсальным асинхронным интерфейсом (SCI08), скоростным синхронным интерфейсом (SPI08), а также специализированными последовательными интерфейсами MSCAN08 и BDLC08, применяемыми в автомобильных системах и системах промышленного управления.

Основными функциями **модуля системной интеграции (SIM08)** являются:

1) формирование внутренней тактовой частоты для ЦПУ и встроенных подсистем (таймера, последовательных интерфейсов и т. д.);

2) обеспечение совместно с модулем формирователя тактовой частоты (CGM) режимов пониженного энергопотребления STOP и WAIT, а также программное управление тактовой частотой с помощью ФАПЧ;

3) управление прерываниями и RESET: формирование сигнала сброса при обнаружении неправильных кодов команды и адресов, а также поступлении сигналов от модулей контроля напряжения питания и сторожевого таймера; обработка и арбитраж программных и аппаратных прерываний.

Встроенная память может состоять из масочного или программируемого ПЗУ, ЭСПЗУ (EEPROM и Flash EEPROM), ОЗУ.

Модуль **управления ЖКИ-дисплеем (LCD08)** позволяет подключать до 1280 сегментов ЖКИ (32 группы по 40 сегментов) и содержит внутреннее буферное ОЗУ объемом 160 байт с побитовой адресацией. Встроенный генератор накачки заряда позволяет формировать необходимые уровни напряжения на выходе драйверов, а регулировка контрастности с 8-разрядным разрешением и обратной связью поддерживает заданную контрастность во всем диапазоне рабочих напряжений.

Другие периферийные модули семейства HC08:

1) АЦП (ADC08);

2) 12-разрядный 6-канальный контроллер ШИМ (PWM08);

3) таймер периодических прерываний (PIT08);

4) модуль расширения адресации внешней памяти до 16 Мбайт (ADX08).

Семейство HC11 является одним из наиболее распространенных и популярных в мире семейств микроконтроллеров (на настоящий момент продано более 100 млн микроконтроллеров этого семейства).

В отличие от специализированных микроконтроллеров «заказных» семейств семейство HC11 содержит набор приблизительно из 40 универсальных и высокопроизводительных микроконтроллеров, ориентированных как на массовые рынки, так и на среднее и мелкое производство.

Все микроконтроллеры этого семейства содержат одинаковое 8-разрядное ЦПУ, основанное на микропроцессорной 8-разрядной архитектуре «второго поколения» (MC6809) и отличающееся повышенной производительностью, эффективной системой команд и методов адресации. Ставшая фактически промышленным стандартом архитектура HC11 поддерживается наибольшим количеством доступных отладочных средств, бесплатным программным обеспечением, многочисленными примерами применения.

Микроконтроллеры семейства HC11 содержат встроенную память различных типов и конфигураций. Периферийные функции представлены подсистемами, наиболее часто требующимися в системах встроенного управления: многофункциональными таймерами, АЦП, ШИМ, ЦАП, последовательными интерфейсами, а также контроллером ПДП, синтезатором тактовой частоты, встроенным сопроцессором, ускоряющим выполнение умножения и деления на порядок, и другими функциями. Как и в других семействах, имеются большое разнообразие корпусов, а также версии с пониженным напряжением питания и расширенным температурным диапазоном.

Сочетание производительного процессорного ядра, большого разнообразия встроенной памяти и эффективных интерфейсов внешних устройств с пониженным потреблением, широким температурным диапазоном и высокой надежностью позволяют микроконтроллерам семейства HC11 лидировать в таких применениях, как бытовая техника, средства беспроводной связи, устройства сбора информации и промышленного управления, телефония, автомобильная электроника и т. д.

Процессор и система команд семейства HC11. Процессор семейства HC11 чрезвычайно удобен для программирования и оптимизирован по энергопотреблению и быстродействию. Наиболее характерные его особенности (рис. 55):

1) два 8-битных или один 16-битный аккумулятор;

2) два 16-битных индексных регистра;

3) два программно управляемых режима пониженного энергопотребления;

4) операции умножения 8×8 и деления 16/16;

5) внутренняя тактовая частота до 4 МГц.

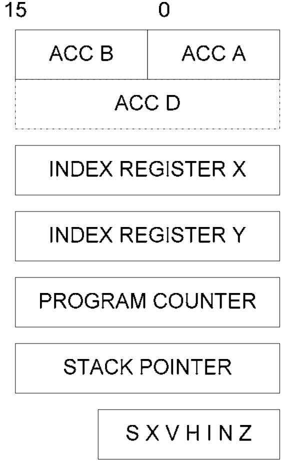


Рис. 55. Программная модель микроконтроллера семейства HC11

Процессор некоторых моделей семейства содержит встроенный математический сопроцессор, выполняющий 16-битные операции умножения и деления в 10 раз быстрее, чем процессор. Существуют версии микроконтроллеров с программно управляемым значением тактовой частоты на основе ФАПЧ, что позволяет гибко управлять энергопотреблением в зависимости от сложности вычислительных задач. Процессор семейства HC11 поддерживает следующие режимы адресации: неявная, непосредственная, прямая, расширенная, индексная и относительная.

Система команд представлена следующими группами:

1) команды пересылки данных, связанные с аккумуляторами (LDAB (load ACC B), LDD, STAB, TAB (transfer A to B), CLRA (Clear A), PSHA (push A to stack), PULA, ...);

2) команды пересылки для стека и индексных регистров (PSHX, TSX (transfer SP to X), ...);

3) команды переходов (JMP, JSR (jump to subroutine), RTS (return from subroutine), переходы по условиями состояниям битов, ...);

4) арифметические команды (ADD, SUB, INC, DEC, MUL, DIV, CMP, …);

5) логические команды (AND, OR, EOR, COM, ...);

6) команды работы с битами (установка, сброс, проверка, сравнение, сдвиги, ...);

7) специальные команды (STOP, WAI (wait for interrupt), SWI (software interrupt), ...).

Встроенная память МК семейства HC11. Микроконтроллеры семейства HC11 имеют в своем составе все типы внутренней памяти, упоминавшиеся при рассмотрении семейства MC68HC05: ПЗУ (программируемое или масочное), EEPROM, ОЗУ объемом до 2К байт. Характерно, что все микроконтроллеры семейства HC11 адресуют внешнюю память, причем есть версии с немультиплексированными магистралями данных и адреса (HC11F1), а также версии с расширенным до 256К...1М адресным пространством с помощью программируемых выборок внешней памяти (HC11Kx).

Микроконтроллеры семейства HC11 функционируют в одном из трех режимов, которые определяются состоянием специальных входов в момент RESET. В однокристальном режиме программа находится во встроенном ППЗУ (или EEPROM), при этом линии всех портов доступны для ввода/вывода. В расширенном режиме возможно подключение внешней памяти программ и данных. В режиме загрузки управление после RESET передается программе, записанной в масочном ПЗУ и производящей загрузку кода, например из персонального компьютера по последовательному порту в любую область памяти (включая программирование ППЗУ и EEPROM). Примечательно, что микроконтроллеры семейства HC11 позволяют программно переназначить начало областей ОЗУ, EEPROM и регистров на границу любой области размером 4К.

Встроенные подсистемы МК семейства HC11. Многофункциональный таймер микроконтроллеров семейства HC11 имеет структуру и выполняет функции, аналогичные таймеру семейства HC05 (16-разрядный счетчик с предделителем, функции входной фиксации и выходного сравнения). Счетчик внешних импульсов, также входящий в подсистему таймера, осуществляет либо подсчет перепадов уровня на входе, либо подсчет импульсов тактовой частоты при наличии активного уровня на входе. К подсистеме таймера относится также COP Watchdog таймер, формирующий аппаратный RESET при отсутствии обращения к нему больше установленного времени и защищающий тем самым систему от «зависаний».

Подсистема последовательного интерфейса представлена универсальным асинхронным последовательным интерфейсом SCI (от 1 до 3) и скоростным синхронным интерфейсом SPI (1 или 2), полностью аналогичным соответствующим подсистемам HC05.

Аналого-цифровой преобразователь производит преобразование внешнего напряжения в диапазоне от Vrl (нижнее опорное напряжение) в 8-разрядный код от $00 до $FF, соответственно. АЦП использует метод последовательных приближений и содержит встроенную схему выборки-хранения. Процесс преобразования занимает 16 мкс при стандартной тактовой частоте 4 Мгц. Встроенный мультиплексор позволяет проводить преобразование по одному из защищенных от перенапряжения внешних аналоговых входов (до 12), а также измерять Vrl, Vrh и (Vrh+Vrl)/2 для проведения контроля и юстировки.

Среди прочих подсистем, имеющихся в некоторых микроконтроллерах семейства HC11, можно выделить ШИМ, ЦАП, наборы силовых ключей, контроллер ЖКИ, арифметический сопроцессор и контроллер ПДП.

16- и 32-разрядные микроконтроллеры. По мере усложнения функций, выполняемых встраиваемым контроллером, и соответствующего роста требований, предъявляемых к производительности МК, а также в связи с увеличением разрядности АЦП и объемов передаваемых данных, перед многими разработчиками остро встал вопрос о переходе на МК с более высокой разрядностью, поскольку дальнейшее развитие архитектуры и повышение тактовой частоты 8-разрядных МК имеет очевидный предел и часто не дает требуемого эффекта.

Первой и основной причиной, привлекающей пользователей к 16- и 32-разрядным МК, является их *высокая производительность*. Помимо увеличенной разрядности, автоматически повышающей объем вычислений и передаваемых данных в единицу времени, производительность определяется тактовой частотой и дополнительной производительностью, обеспечиваемой встроенными сопроцессорами.

Дополнительное повышение производительности осуществляется за счет введения в структуру МК сопроцессоров разной функциональной ориентации: обмен данными; вычисления математических операций, ввода/вывода, цифровой обработки сигналов и др. Эти интеллектуальные, содержащие собственное микроядро, встроенные подсистемы позволяют разгрузить центральный процессор, берут на себя выполнение специфических функций и определяют ориентацию МК на конкретную область использования. Так, например, наличие коммуникационного сопроцессора обеспечивает эффективное использование МК в системах передачи данных, а наличие таймерного сопроцессора позволяет переложить на него обработку быстротекущих процессов в промышленных контроллерах.

Простота построения системы достигается за счет наличия специального модуля системной интеграции, позволяющего напрямую подключать к МК все типы внешней памяти и разнообразные внешние устройства, снижая количество дополнительных компонентов, площадь платы и стоимость системы. Возможность динамического изменения разрядности шины данных позволяет использовать дешевые 8-разрядные внешние устройства и память там, где это несущественно влияет на производительность.

Простота отладки системы обеспечивается наличием встроенного в МК отладчика, который без дополнительных аппаратных средств позволяет связаться с компьютером, просматривать и изменять содержимое регистров и памяти, осуществлять запуск и остановка программы и т. п.

Повышенная надежность функционирования системы на базе МК достигается защитой информации от несанкционированного доступа путем организации работы МК в одном из двух режимов: пользователя и супервизора. В режиме пользователя программе доступны только регистры программной модели пользователя (регистры общего назначения, программный счетчик и т. п.), а в режиме супервизора разрешается обращение к регистрам управления МК и могут выполняться системные команды, влияющие на безопасность функционирования. Кроме того, реализована аппаратная защита от ошибок на магистрали, а надежность программного обеспечения дополнительно обеспечивается сторожевым таймером. Возможность отключения процессорного ядра в структуре МК позволяет создавать многопроцессорные системы, в которых проявляется возможность обеспечивать заданные уровни надежности за счет перестройки системы при возникновении отказов.

Интерес к МК, а в дальнейшем и потребность в них определяется, прежде всего, тем, что МК как однокристальное устройство объединяет процессор, память и периферийные устройства, которые реализуют максимальный для данной задачи набор функций. В развитие этой идеологии Motorola строит свои МК на базе стандартных модулей, из набора которых быстро и с минимальными затратами создается новый МК.

Семейства 16- и 32-разрядных МК Motorola используют стандартную внутримодульную шину (IMB), основное преимущество которой заключается в том, что в МК с различным процессорным ядром могут использовать одни и те же периферийные модули. Это не только обеспечивает лучшее использование модулей, которые известны производителям и протестированы в тысячах приложений, но и позволяет, например, программы, написанные на C для 16-разрядного МК и затрагивающие периферийные устройства, просто перекомпилировать и выполнять без изменений на 32-разрядном МК.

Упрощенная модель МК, показывающая набор основных модулей, приведена на рис. 56.

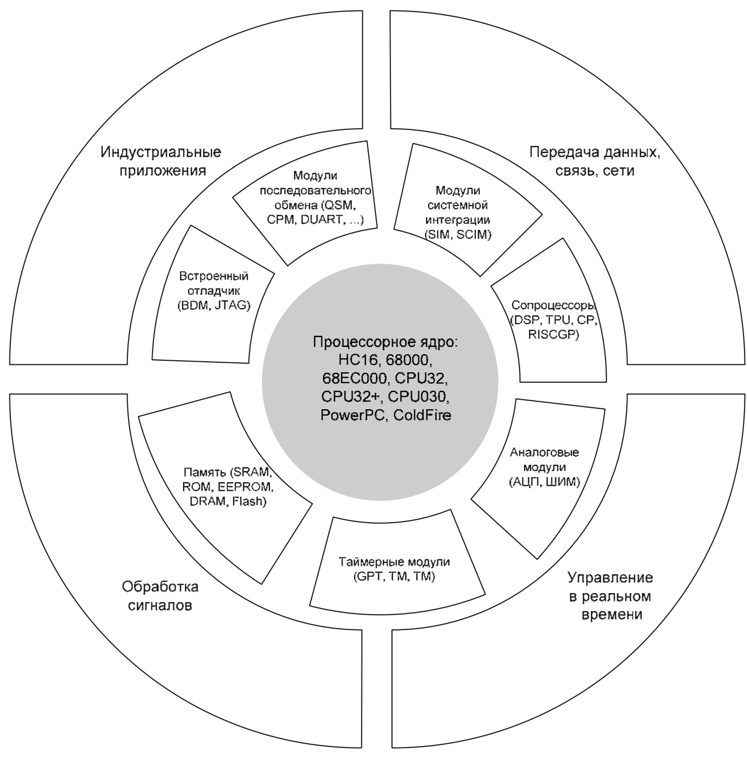


Рис. 56. Упрощенная модель микроконтроллера

Базовым модулем МК является процессорное ядро, которое и отличает одно семейство МК от другого. Для семейств 16-разрядных МК HC12 и HC16 используются ядра CPU12 и CPU16, для семейства МК M68300 используются ядра 68000, 68EC000, CPU32, CPU32+, CPU030. Данные два семейства основаны на CISC-архитектуре. Новые семейства встроенных и интегрированных процессоров и МК MPC500 и MPC800 основаны на 32-разрядном ядре PowerPC с RISC-архитектурой. Новое семейство интегрированных 32-разрядных RISC-процессоров ColdFire, имеющих переменную длину инструкций и многие черты семейства M68K, призваны обеспечить новый уровень соотношения производительность/цена для массовых рынков.

Семейство 16-разрядных МК HC12. Целевыми рынками МК данного семейства являются разнообразные портативные устройства, особенно средства беспроводной связи, автомобильная электроника, устройства промышленного управления. Процессорное ядро семейства HC12 (CPU12), основанное на популярной архитектуре HC11, обеспечивает полную поддержку операций нечеткой логики, что дает возможность экономии кода программы, упрощения алгоритмизации задачи и другие преимущества. Система команд CPU12 позволяет работать с нечетным байтом и имеет расширенную систему адресации внешней памяти (до 5 Мбайт). Ассемблер CPU12 воспринимает исходные тексты для HC11 без изменений, что

дает пользователям HC11 возможность перейти на новый уровень производительности с минимальными затратами. Наличие встроенного отладчика и поддержка отладочными средствами облегчают разработчику переход на новую архитектуру.

MC68HC812A4 содержит CPU12 с внутренней тактовой частотой 8 МГц; 4К ПЗУ EEPROM с побайтовым стиранием; 1К ОЗУ; 8-разрядный 8-канальный АЦП; 8-канальный 16-разрядный универсальный таймер; два асинхронных и один синхронный последовательный интерфейс; прерывания реального времени и сторожевой таймер; 7 программируемых выборок с поддержкой расширенной адресации (до 4М памяти программ и 1М памяти данных).

MC68HC912B32, разработанный для автомобильных и индустриальных приложений, 32К ПЗУ Flash EEPROM; 768 байт ОЗУ 8-разрядный 8-канальный АЦП; 8-канальный 16-разрядный универсальный таймер; 8-разрядный 4-канальный ШИМ, оптимизированный для управления двигателями; асинхронный и синхронный последовательный интерфейс, а также автомобильный контроллер обмена BDLC (J1850); прерывания реального времени и сторожевой таймер.

Семейство 16-разрядных МК HC16. МК семейства HC16 используются в различных системах управления автомобилями, телекоммуникационном оборудовании (сотовых телефонах, телефонных коммутаторах), бытовой электронике (видеокамерах, телевизорах, цифровых аудио системах), офисной технике (факсах, модемах, копировальной технике), медицинском оборудовании, робототехнике. Эффективному использованию МК HC16 в этих приложениях способствуют поддержка функций DSP, высокая производительность истинно 16-битного CPU16 с частотой до 25 МГц и мощная периферия, простота построения системы на базе МК HC16, обеспечиваемая модулем системной интеграции, а также простота отладки программы благодаря встроенному отладчику.

CPU16 является процессорным ядром для микроконтроллеров семейства M68HC16. Это полностью 16-разрядное процессорное устройство, совместимое с процессорным ядром МК HC11 и обладающее дополнительными возможностями. CPU16 обеспечивает адресацию памяти до 1 Мбайта для данных и 1 Мбайта для программ, а также развитую приоритетную систему обработки прерываний. Кроме того, CPU16 имеет функциональные возможности цифрового сигнального процессора DSP с полной архитектурой умножителя-сумматора с накоплением (MAC), позволяющие умножать 16-разрядные числа и сохранять результат в 36-разрядном накапливающем аккумуляторе за одну команду. Все это в совокупности с эффективными методами адресации и удобным набором команд дает пользователю мощный инструмент для реализации сложных и интенсивных вычислений.

Общие характеристики CPU 16 (рис. 57):

1) 16-битное АЛУ и внутренняя шина данных;

2) 20-битная внешняя шина адреса;

3) два 16-разрядных аккумулятора;

4) три 16-разрядных индексных регистра;

5) раздельные пространства программ и данных по 1 Мбайт;

6) многоуровневая обработка прерываний;

7) программная совместимость снизу вверх с процессорным ядром МК 68НС11;

8) быстрые команды умножения, деления и сдвига;

9) усовершенствованная обработка исключительных ситуаций;

10) расширенные режимы адресации (9 режимов);

11) полный набор 16-разрядных команд;

12) поддержка функций DSP с помощью 16-разрядного умножителя с 36-разрядным сумматором;

13) инструкция LPSTOP – остановка с малым энергопотреблением;

14) аппаратный BREAKPOINT, режим фоновой отладки;

15) тактовая частота до 16 МГц;

16) полностью статическая технология.

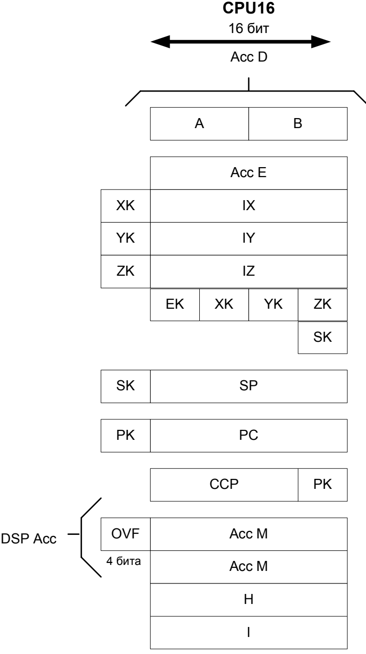


Рис. 57. Программная модель микроконтроллера семейства НС16

Аналогично ядру M68HC11, CPU16 имеет классическую аккумуляторную архитектуру. Все арифметико-логические операции выполняются над аккумуляторами и операндами памяти, для доступа к которым существует набор индексных регистров, обеспечивающих эффективный механизм адресации к ним. В наборе команд CPU16 имеются также 32-разрядные команды, предназначенные для использования с 32-разрядными регистрами, заложенными в некоторых модулях.

Совершенно новой функцией ядра являются возможности умножителя с накоплением. В CPU16 введен ряд новых регистров, предназначенных для умножения двух 16-разрядных чисел, содержащихся в регистрах H и I, и результат добавляется к содержимому 36-разрядного регистра, называемого М-регистром, за одну MAC-инструкцию (multiply and accumulate). Эта инструкция может выполняться в режиме многократного умножения с накоплением (RMAC). 16-разрядные числа для умножения последовательно извлекаются по адресам, определяемым индексными регистрами X и Y, и загружаются в регистры H и I, с авто инкрементом/декрементом обеих индексных регистров X и Y на величину до 15. Эта функция, необходимая для цифровой фильтрации, выполняется за 720нс (12 циклов тактовой частоты).

Семейство 32-разрядных МК 68300. Основными особенностями МК семейства 68300 являются: высокопроизводительное 32-разрядное процессорное ядро, основанное на промышленном стандарте 68000; широкая номенклатура периферийных модулей, в том числе специализированных для определенных приложений; исключительная гибкость и простота построения и отладки системы; распространенность и доступность.

В семействе МК 68300 можно выделить три основные группы МК, принципиально отличающиеся по функциональному назначению:

1) коммуникационные МК. В эту группу можно отнести все МК, содержащие коммуникационный сопроцессор;

2) МК для промышленного управления. МК этой группы содержат таймерный сопроцессор и применяются в индустриальных системах управления, автомобильных контроллерах и т. д.;

3) МК общего назначения. Эти МК, иногда называемые интегрированными процессорами (ИП), содержат, помимо центрального процессора, только наиболее распространенную универсальную периферию: модуль системной интеграции, контроллер ПДП, последовательный интерфейс, часы реального времени, и т. д.

Отличительной особенностью коммуникационных МК (КМК) является наличие в их составе специализированного скоростного коммуникационного сопроцессора с RISC-ядром, управляющего обменом данными по нескольким независимым каналам, поддерживающего практически все распространенные протоколы обмена и позволяющего гибко и эффективно распределять и обрабатывать последовательные потоки данных с временным разделением каналов (например, ИКМ и ISDN PR1). Среди многочисленных применений КМК можно выделить цифровые телефонные станции, абонентское и групповое оборудование ISDN, базовые станции сотовой связи, модемы, терминалы, мосты, маршрутизаторы, а также распределенные промышленные контроллеры и многие другие устройства.

Все КМК имеют похожую структуру, включающую центральный процессор (CPU), осуществляющий общее управление; коммуникационный процессор (CPM), обрабатывающий последовательные данные; и модуль системной интеграции (SIM), упрощающий подключение памяти и внешних устройств. Обмен данными требует минимального участия CPU, функции которого сводятся, как правило, к обработке флагов окончания передачи и переустановке указателей – все остальные задачи по обработке протокола и управлению обменом автоматически выполняет интеллектуальный коммуникационный сопроцессор.

Семейства 32-разрядных МК с RISC-архитектурой. Семейство ColdFire. Основанный на концепции переменной длины команд, Co/dFire сочетает архитектурную простоту стандартного 32-разрядного RISC-процессора с экономией памяти. Использование архитектуры с инструкциями переменной длины дает значительные преимущества по сравнению со стандартной RISC-архитектурой с инструкциями фиксированной длины. Уплотненный двоичный код процессора ColdFire занимает меньшие объемы памяти, чем код RISC-процессора с инструкциями фиксированной длины. Это позволяет более эффективно использовать память системы для прикладной программы, а также выбирать более медленную и менее дорогостоящую память для достижения заданного уровня производительности. Интегрированные периферийные функции обеспечивают высокую эффективность и гибкость. Стандартная конфигурация предполагает наличие последовательного интерфейса, двух многофункциональных таймеров, программируемого сторожевого таймера и системной интеграции: управления внешними шинами, системной защиты, прерываниями, встроенного отладчика.

Процессорное ядро MCF5204 имеет производительность 13,5 MIPS при тактовой частоте 33 МГц, при этом большинство команд выполняются за 1 цикл. Команды управления потреблением позволяют снижать интегральный показатель потребляемой мощности. Системный интерфейс обеспечивает прямое подключение ОЗУ, ПЗУ и внешних устройств с разрядностью 8 или 16, а также обработку прерываний и системную защиту (имеется 16-битный сторожевой таймер с предделителем, отслеживаются двойные ошибки шины, превышение времени отклика устройства на шине и т. д.). MCF5204 содержит встроенный кэш команд объемом 512 байт и быстрое статическое ОЗУ того же объема, что обеспечивает 1-цикловый доступ к наиболее критичным командам и данным. Асинхронный последовательный интерфейс обеспечивает дуплексную работу с поддержкой модемных сигналов управления (CTS, RTS). Двухканальный 16-битный универсальный таймер с 8-битным предделителем имеет разрешение 30 нс при тактовой частоте 33 МГц.

MCF5206 имеет производительность 17 MIPS при частоте 33 МГц, содержит встроенный контроллер DRAM, а также синхронный последова-

тельный интерфейс M-bus для подключения ЖК-дисплеев, последовательного EEPROM и других устройств с последовательным доступом.

Семейства МК на базе PowerPC. Архитектура микропроцессора PowerPC основывается на архитектуре POWER (Performance Optimization With Enhanced RISC – оптимизация производительности с расширенной RISC-архитектурой), первоначально предложенной фирмой IBM. Архитектура была переопределена фирмой Motorola для того, чтобы обеспечить более эффективные однокристальные решения. Процессоры PowerPC позволяют реализовать на их основе различные системы – от недорогих портативных устройств до мультипроцессорных суперкомпьютеров. Они стали основой RISC-ядра мощных высокопроизводительных микроконтроллеров и интегрированных процессоров семейств MPC 500 и MPC 800. Основной особенностью МК с ядрами PowerPC является суперскалярная RISC-архитектура.

В состав ядра PowerPC входят исполнительные блоки процессора, кэш-память и блоки управления памятью. Исполнительные блоки, работающие параллельно, представлены целочисленным блоком, выполняющим все команды целочисленной арифметики, блоком загрузки/хранения, выполняющим операции перемещения данных регистровым блоком, содержащим тридцать два 32-разрядных регистров общего назначения, а также блоки данных с историей последних операций, блоком переходов, состоящим из формирователя следующего адреса, очереди предварительной выборки команд и системы обработки прерываний, а также фонового отладчика, позволяющего подробно прослеживать работу процессора, включая работу конвейера и кэш-памяти. Двунаправленный ассоциативный кэш команд объемом 4К организован в виде 128 наборов по две линии из четырех слов в каждом. Физически адресуемый кэш данных объемом 4К, также организованный в виде двунаправленных ассоциативных наборов, имеет 1-цикловый доступ при попадании и 1-цикловую задержку при непопадании. Блок управления памятью, осуществляющий трансляцию логических адресов в физические, содержит раздельные 32-входовые буферы трансляции для памяти команд и данных.

2. Специализированные микропроцессоры. Цифровые процессоры обработки сигналов

DSP-процессоры предназначены для осуществления цифровой обработки сигнала – математических манипуляций над оцифрованными сигналами. Они широко применяются в беспроводных системах, аудио- и видеообработке, системах управления. С ростом числа приложений, исполь-

зующих DSP, и сложности алгоритмов обработки увеличивается и требования к ним в плане повышения быстродействия и оснащенности интерфейсными и другими специализированными узлами. К настоящему времени появилось множество типов DSP, как универсальных, так и ориентированных на достаточно узкий круг задач. Естественно, ни один из процессоров не может подойти для всех приложений. Например, для таких портативных устройств, как мобильные телефоны, портативные цифровые плееры, стоимость, степень интеграции и потребляемая мощность являются первостепенными, а максимальная производительность зачастую не нужна (т. к. обычно влечет за собой значительное повышение потребляемой мощности, не давая преимуществ при обработке относительно низкоскоростных аудиоданных). В то же время для гидроакустических или радиолокационных систем определяющими параметрами являются скорость работы, наличие высокоскоростных интерфейсов и удобная система разработки, а стоимость является второстепенным критерием. Кроме того, во многих случаях имеет смысл учитывать и место на рынке, занимаемое поставщиком процессора, т. к. далеко не все производители могут предоставить в ваше распоряжение спектр процессоров, покрывающих все ваши потребности. Сложившееся к настоящему времени распределение рынка между ведущими поставщиками (см. табл. 25) показывает, что 4 компании, стоящие в начале списка, поставляют более 80% всех используемых в мире DSP. Именно эти компании наиболее известны и на российском рынке, и их продукция часто упоминается.

Таблица 25

Основные производители DSP и принадлежащие им доли рынка

Company Name	Доля рынка DSP
Texas Instruments	54,3%
Freescale Semiconductor	14,1%
Analog Devices	8,0%
Philips Semiconductors	7,5%
Agere Systems	7,3%
Toshiba	4,9%
DSP Group	2,2%
NEC Electronics	0,6%
Fujitsu	0,4%
Intersil	0,3%
Other Companies	0,5%
Total	100,0%

Следует помнить, что производители DSP, проектируя новые микросхемы, достаточно четко позиционируют их для использования в тех или иных приложениях. Это оказывает влияние и на их архитектуру, и на быстродействие, и на оснащение процессора тем или иным набором периферийных модулей. В таблице 26 показано позиционирование DSP с точки зрения их создателей.

Таблица 26

Области применения семейств сигнальных процессоров разных производителей

Область применения	Семейства процессоров
Обработка видео, видеонаблюдение, цифровые камеры, 3D графика	TMS320DM64x/DaVinci, TMS320C64xx, TMS320C62xx (TI), PNX1300, PNX1500, PNX1700 (Philips) , MPC52xx (Freescale)
Обработка аудио, распознавание речи, синтез звука	TMS320C62xx, TMS320C67xx (TI), SHARC (Analog Devices)
Портативные медиа устройства	TMS320C54xx, TMS320C55xx (TI), Blackfin (Analog Devices)
Беспроводная связь, телекоммуникации, модемы, сетевые устройства	TMS320C64xx, TMS320C54xx, TMS320C55xx (TI), MPC7xxx, MPC86xx, MPC8xx PowerQUICC I, MPC82xx PowerQUICC II, MPC83xx PowerQUICC II Pro, MPC85xx PowerQUICC III (Freescale), Blackfin, TigerSHARC (Analog Devices), PNX1300 (Philips)
Управление приводами, преобразование мощности, автомобильная электроника, предметы домашнего обихода, офисное оборудование	TMS320C28xx, TMS320C24xx (TI), ADSP-21xx (Analog Devices), MPC55xx, MPC55xx (Freescale)
Медицина, биометрия, измерительные системы	TMS320C62xx, TMS320C67xx, TMS320C55xx, TMS320C28xx (TI), TigerSHARC, SHARC (Analog Devices)

Формат данных и разрядность. Одна из основных характеристик цифровых сигнальных процессоров – формат обрабатываемых данных. Все DSP работают либо с целыми числами, либо с числами в формате с плавающей точкой, причем для целых чисел разрядность составляет 16 или 32, а для чисел с плавающей точкой она равна 32. Выбирая формат данных, необходимо иметь в виду следующее: DSP с целочисленными данными (или данными с фиксированной точкой) обычно дешевле и обеспечивают большую абсолютную точность при равной разрядности (т. к. на мантиссу в 32-битном процессоре с фиксированной точкой отводятся все 32 бита, а в процессоре с плавающей точкой – только 24).

В то же время динамический диапазон сигналов, с которыми могут без искажений работать процессоры, у процессоров с фиксированной точкой значительно уже (на несколько десятичных порядков). При относительно простых алгоритмах обработки это может быть неважно, т. к. динамический диапазон реальных входных сигналов чаще всего меньше, чем допускает DSP, однако в некоторых случаях возможно возникновение ошибок переполнения при выполнении программы. Это приводит к принципиально неустранимым нелинейным искажениям выходного сигнала, аналогичным искажениям из-за ограничения в аналоговых схемах.

Следовательно, при выборе DSP необходимо тщательно анализировать алгоритм обработки и входные сигналы для правильного выбора разрядности и типа арифметики. Иногда при невозможности подобрать подходящий процессор с плавающей точкой (из-за большей его стоимости или энергопотребления) используют DSP с фиксированной точкой и сжатие динамического диапазона обрабатываемых сигналов (компрессию), однако это приводит к увеличению сложности алгоритма обработки сигнала и повышает требования к быстродействию.

Конечно, можно эмулировать операции с плавающей точкой и на процессоре с целочисленной арифметикой или перейти к обработке чисел удвоенной разрядности, однако это также значительно усложняет программу и значительно снижает быстродействие.

Несмотря на все ограничения, большинство встроенных приложений используют процессоры с фиксированной точкой из-за меньшей цены и энергопотребления. Увеличение количества разрядов повышает стоимость, размер кристалла и число необходимых выводов процессора, а также необходимый объем внешней памяти. Поэтому разработчики стремятся использовать кристалл с минимально возможной разрядностью.

Стоит заметить, что разрядность данных и разрядность команд процессоров не всегда эквивалентны.

Скорость. Ключевой параметр при выборе процессора – это скорость. Она влияет на время выполнения обработки входного сигнала и, следовательно, определяет его максимальную частоту. Одна из самых частых ошибок разработчика – отождествление тактовой частоты и быстродействия, что в большинстве случаев неправильно. Очень часто скорость работы DSP указывают в MIPS (миллионах инструкций в секунду). Это наиболее просто измеряемый параметр.

Однако проблема сравнения скорости различных DSP состоит в том, что процессоры имеют различные системы команд, и для выполнения од-

ного и того же алгоритма разными процессорами требуется разное число этих команд. Кроме того, иногда для выполнения различных команд одним процессором требуется различное количество тактов синхронизации. В результате процессор со скоростью 1000 MIPS вполне может оказаться в разы медленнее процессора со скоростью 300 MIPS, особенно при различной их разрядности.

Одно из решений этой проблемы – сравнивать процессоры по скорости выполнения определенных операций, например, операции умножения с накоплением (MAC). Скорость выполнения таких операций критична для алгоритмов, использующих цифровую фильтрацию, корреляцию и преобразования Фурье. К сожалению, такая оценка также не дает полной информации о реальном быстродействии процессора.

Наиболее точной является оценка скорости исполнения определенных алгоритмов – например, КИХ- и БИХ-фильтрации, однако это требует разработки соответствующих программ и тщательного анализа результатов тестирования.

Существуют компании, занимающиеся анализом и сравнением процессоров по основным характеристикам, в том числе и по скорости. Лидером среди таких компаний является BDTI – Berkeley Design Technology. Наибольшее преимущество по скорости выполнения принадлежит DSP процессора компании Texas instruments.

Организация памяти. Организация системы памяти процессора влияет на производительность. Это связано с тем, что ключевые команды DSP являются многооперандными и ускорение их работы требует одновременного чтения нескольких ячеек памяти. Например, команда MAC требует одновременного чтения 2 операндов и самой команды для того, чтобы ее можно было выполнить за 1 такт. Это достигается различными методами, среди которых применение многопортовой памяти, разделение на память программ и память данных (Гарвардская архитектура), использование кэша команд и т. д.

Необходимый объем памяти определяется приложением. Необходимо учитывать, что встроенная в процессор память обычно имеет значительно большую скорость работы, чем внешняя, однако увеличение ее объема увеличивает стоимость и энергопотребление DSP, а ограниченный объем памяти программ не позволяет хранить сложные алгоритмы. В то же время при достаточности этого объема для ваших целей наличие встроенной памяти позволяет значительно упростить конструкцию в целом и понизить ее размеры, энергопотребление и стоимость.

Большинство DSP с фиксированной точкой, применяющиеся во встраиваемых приложениях, предполагают малый объем внутренней памяти, обычно от 4 до 256 Кбайт и невысокую разрядность внешних шин данных.

В то же время DSP с плавающей точкой обычно предполагают работу с большими массивами данных и сложными алгоритмами и имеют либо встроенную память большого объема, либо большую разрядность адресных шин для подключение внешней памяти (а иногда и то, и другое). Еще раз подчеркнем – выбор типа и объема памяти должен быть результатом тщательного анализа приложения, в котором используется DSP.

Удобство разработки приложений. Степень сложности разработки определятся приложением. При этом необходимо иметь в виду, что большее удобство для разработчика (обычно связываемое с использованием при программировании DSP языков высокого уровня) в большинстве случаев оборачивается получением менее компактного и быстрого кода, что оборачивается необходимостью использования более мощных и дорогих DSP. С другой стороны, в современных условиях скорость разработки (и, следовательно, выхода нового изделия на рынок) может принести больше выгод, чем затраты времени на оптимизацию кода при написании программы на ассемблере.

Кроме того, следует помнить, что безошибочных программ не бывает, поэтому средства отладки и возможность коррекции программ в готовом устройстве очень часто имеют первостепенное значение. В то же время при выборе DSP и средств разработки необходимо учитывать некоторые особенности архитектуры процессоров.

Те, кто использует компиляторы с языков высокого уровня (ЯВУ), иногда замечают, что они генерируют лучший код для процессоров с плавающей точкой. Это происходит по нескольким причинам: во-первых, большинство языков высокого уровня изначально не поддерживают арифметику с фиксированной точкой, во-вторых, система команд DSP с фиксированной точкой более ограничена, и, в-третьих, процессоры с плавающей точкой обычно накладывают меньшие ограничения на объем используемой памяти.

Наилучшие результаты получаются при компиляции программ на ЯВУ для VLIW-процессоров (процессорах со сверхдлинным словом команды) с простой ортогональной RISC-системой команд и большими регистровыми файлами. Однако даже для этих процессоров генерируемый компилятором код получается более медленным по сравнению с оптимизированным вручную ассемблерным. С другой стороны, возможность сначала смоделировать процесс обработки сигнала в программе типа MathLab

с дальнейшей автоматической трансляцией его в программу для DSP позволяет избавиться от множества серьезных ошибок еще на начальном этапе разработки.

Отладку готовых программ можно производить либо на аппаратном эмуляторе готовой системы, либо на программном симуляторе. Обычно отладка на симуляторе несколько проще с точки зрения используемой аппаратуры, однако она не позволяет выявить все возможные ошибки. Почти все производители обеспечивают разработчиков и симуляторами, и эмуляторами своих DSP. Почти все современные DSP поддерживают внутрисхемную эмуляцию в соответствии со стандартом IEEE 1149.1 JTAG. При использовании технологии JTAG мы переходим от эмуляции процессора внешним устройством к непосредственному контролю над процессором при выполнении программы, что позволяет значительно увеличить степень соответствия макета реальному устройству и, следовательно, повысить надежность процесса отладки.

Помимо эмуляторов, производители предлагают широкий набор так называемых «стартер-китов» и «оценочных модулей», с помощью которых можно сразу приступить к разработке приложения, не дожидаясь изготовления макета разрабатываемого устройства. Кроме этого, в некоторых приложениях эти средства разработки можно использовать как конечные устройства.

Энергопотребление. DSP-процессоры широко используются в мобильных устройствах, где потребление мощности является основной характеристикой. Для снижения энергопотребления используется множество методов, в том числе уменьшение напряжения питания и введение функций управления потреблением, например, динамического изменения тактовой частоты, переключения в спящий или дежурный режим или отключения неиспользуемой в данный момент периферии. Следует отметить, что эти меры оказывают значительное воздействие на скорость работы процессора и при некорректном использовании могут привести к неработоспособности проектируемого устройства (в качестве примера можно привести некоторые сотовые телефоны, которые в результате ошибок в программах управления энергопотреблением иногда переставали включаться) или к ухудшению его эксплуатационных характеристик (например, значительному времени восстановления работоспособности при выходе из спящего режима).

Оценка потребления мощности является не простой задачей, так как эта величина варьируется в зависимости от выполняемых процессором задач. К сожалению, большинство производителей публикуют только «типичное» и «максимальное» потребление, а что понимается под этими опреде-

лениями, не всегда ясно. Исключением является компания Texas Instruments, которая указывает потребление мощности в зависимости от типа команды и конфигурации процессора.

Стоимость. Стоимость процессора, несомненно, является определяющей величиной при выборе DSP, особенно при больших объемах производства. Обычно разработчики стремятся выбрать наиболее дешевый процессор, однако следует учитывать, что это может привести к значительным затратам на переделку устройства, если выбранный процессор по каким-либо причинам не позволит добиться нужных характеристик. Кроме того, при выборе процессора по критерию стоимости необходимо принимать во внимание стоимость внешних компонентов (например, DSP со встроенной памятью достаточного объема стоит дороже аналогичного без встроенной памяти, но цена устройства в целом на его основе может быть значительно ниже из-за отсутствия других компонентов и меньшего размера печатной платы). Очень значимым фактором, влияющим на стоимость DSP, является тип его корпуса: ИС в керамических корпусах, рассчитанные на промышленные или специальные условия эксплуатации, стоят значительно дороже таких же ИС, работающих в коммерческом диапазоне температур. И, наконец, цена процессора очень сильно зависит от объема и регулярности поставок.

Методология выбора процессора. Как показано ранее, правильный выбор DSP сильно зависит от приложения: процессор может хорошо подходить для одних приложений, но абсолютно не подходить для других. При выборе процессора нужно определить самые важные в конкретном случае характеристики и расставить их по степени важности. Затем в соответствии с этими критериями отобрать возможных кандидатов и, наконец, выбрать из подходящих лучший, обращая внимание на дополнительные, не критичные характеристики. При этом целесообразно воспользоваться оценкой характеристик процессоров, производимой какой-либо авторитетной компанией (например, BTDI). Следует помнить, что BTDI производит оценку DSP не только по быстродействию, но и по другим критериям: эффективности памяти, энергопотреблению и т. д.

Например, для реализации приложения в первую очередь важны скорость, цена, эффективность работы памяти и энергопотребление. Мы определили основных претендентов, среди которых DSP с ядром C64x и C64x+ от Texas Instruments и TigerSHARC от Analog Devices. На рисунке 2 показан граф сравнительных характеристик этих процессоров по критериям скорости, стоимости, энергопотребления и удобству средств разработки.

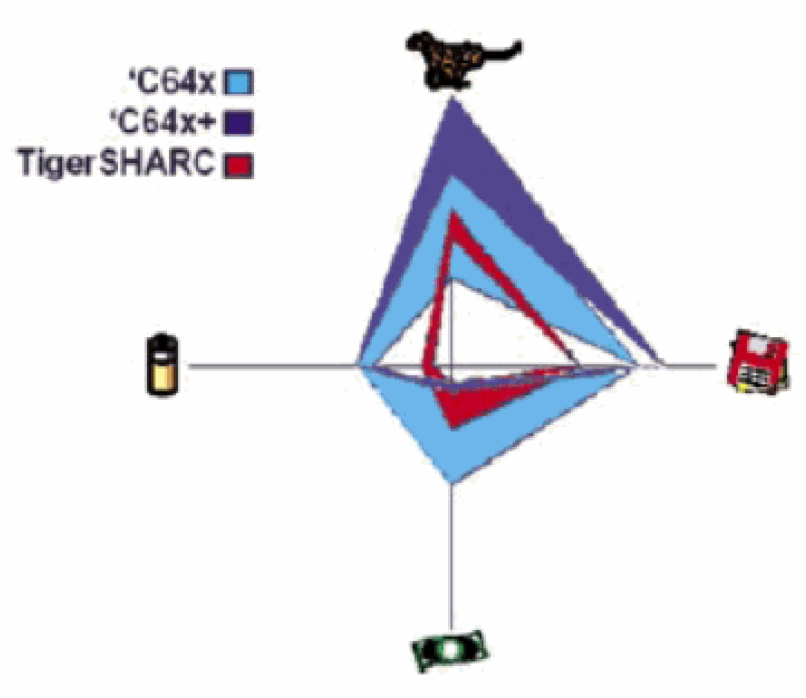


Рис. 59. Диаграмма для выбора DSP

Приоритеты. Если в первую очередь необходима высокая скорость и низкая цена, выбирается Texas Instruments. Если конструируется мобильное устройство и нам нужно низкое энергопотребление, причем мы готовы пожертвовать скоростью, берем Analog Devices. Не исключена вероятность того, что выбранные процессоры окажутся очень близки по ключевым параметрам. В этом случае выбор будет определяться некритичными характеристиками: доступностью средств отладки, предыдущим опытом разработчика, доступностью компонентов и т. д.

МОДУЛЬ IV. ПЕРСПЕКТИВЫ РАЗВИТИЯ МИКРОПРОЦЕССОРНОЙ ТЕХНИКИ

Ассоциативные процессоры. Существующие в настоящее время алгоритмы прикладных задач, системное программное обеспечение и аппаратные средства преимущественно ориентированы на традиционную адресную обработку данных. Данные должны быть представлены в виде ограниченного количества форматов (например, массивы, списки, записи), должна быть явно создана структура связей между элементами данных посредством указателей на адреса элементов памяти, при обработке этих данных должна быть выполнена совокупность операций, обеспечивающих доступ к данным по указателям. Такой подход обуславливает громоздкость операционных систем и систем программирования, а также служит препятствием к созданию вычислительных средств с архитектурой, ориентированной на более эффективное использование параллелизма обработки данных.

Ассоциативный способ обработки данных позволяет преодолеть многие ограничения, присущие адресному доступу к памяти, за счет задания некоторого критерия отбора и проведение требуемых преобразований, только над теми данными, которые удовлетворяют этому критерию. Критерием отбора может быть совпадение с любым элементом данных, достаточным для выделения искомых данных из всех данных. Поиск данных может происходить по фрагменту, имеющему большую или меньшую корреляцию с заданным элементом данных.

Исследованы и в разной степени используются несколько подходов, различающихся полнотой реализации модели ассоциативной обработки. Если реализуется только ассоциативная выборка данных с последующим поочередным использованием найденных данных, то говорят об ассоциативной памяти или памяти, адресуемой по содержимому. При достаточно полной реализации всех свойств ассоциативной обработки, используется термин «ассоциативный процессор».

Ассоциативные системы относятся к классу: один поток команд – множество потоков данных (SIMD = Single Instruction Multiple Data). Эти системы включают большое число операционных устройств, способных одновременно по командам управляющего устройства вести обработку нескольких потоков данных. В ассоциативных вычислительных системах информация на обработку поступает от ассоциативных запоминающих устройств (АЗУ), характеризующиеся тем, что информация в них выбирается не по определенному адресу, а по ее содержанию.

ДНК-процессоры. В настоящее время в поисках реальной альтернативы полупроводниковым технологиям создания новых вычислительных

систем ученые обращают все большее внимание на биотехнологии, или биокомпьютинг, который представляет собой гибрид информационных, молекулярных технологий, также биохимии. Биокомпьютинг позволяет решать сложные вычислительные задачи, пользуясь методами, принятыми в биохимии и молекулярной биологии, организуя вычисления при помощи живых тканей, клеток, вирусов и биомолекул.

Наибольшее распространение получил подход, где в качестве основного элемента (процессора) используются молекулы дезоксирибонуклеиновой кислоты. Центральное место в этом подходе занимает так называемый ДНК-процессор. Кроме ДНК в качестве биопроцессора могут быть использованы также белковые молекулы и биологические мембраны.

Так же, как и любой другой процессор, ДНК процессор характеризуется структурой и набором команд. В нашем случае структура процессора – это структура молекулы ДНК. А набор команд – это перечень биохимических операций с молекулой. Принцип устройства компьютерной ДНК-памяти основан на последовательном соединении четырех нуклеотидов (основных кирпичиков ДНК-цепи). Три нуклеотида, соединяясь в любой последовательности, образуют элементарную ячейку памяти – кодон, которые затем формируют цепь ДНК. Основная трудность в разработке ДНК-компьютеров связана с проведением избирательных однокодонных реакций (взаимодействий) внутри цепи ДНК. Однако прогресс есть уже и в этом направлении. Уже есть экспериментальное оборудование, позволяющее работать с одним из 1020 кодонов или молекул ДНК. Другой проблемой является самосборка ДНК, приводящая к потере информации. Ее преодолевают введением в клетку специальных ингибиторов – веществ, предотвращающих химическую реакцию самосшивки.

Использование молекул DNA для организации вычислений – это не слишком новая идея. Теоретическое обоснование подобной возможности было сделано еще в 50-х годах прошлого века (Р. П. Фейманом). В деталях эта теория была проработана в 70-х годах Ч. Бенеттом и в 80-х М. Конрадом. Первый компьютер на базе ДНК был создан еще в 1994 г. американским ученым Леонардом Адлеманом. Он смешал в пробирке молекулу ДНК, в которой были закодированы исходные данные, и специальным образом подобранные ферменты. В результате химической реакции структура ДНК изменилась таким образом, что в ней в закодированном виде был представлен ответ задачи. Поскольку вычисления проводились в ходе химической реакции с участием ферментов, на них было затрачено очень мало времени. Ричард Липтон из Принстона первым показал, как, используя ДНК, кодировать двоичные числа и решать проблему удовлетворения логического выражения. Суть ее в том, что, имея некоторое логическое выра-

жение, включающее n логических переменных, нужно найти все комбинации значений переменных, делающих выражение истинным. Задачу можно решить только перебором 2^n комбинаций. Все эти комбинации легко закодировать с помощью ДНК, а дальше действовать по методике Адлемана.

Первую модель биокомпьютера, правда, в виде механизма из пластмассы, в 1999 г. создал Ихуд Шапиро из Вейцмановского института естественных наук. Она имитировала работу «молекулярной машины» в живой клетке, собирающей белковые молекулы по информации с ДНК, используя РНК в качестве посредника между ДНК и белком. В 2001 г. Шапиро удалось реализовать вычислительное устройство на основе ДНК, которое может работать почти без вмешательства человека. Система имитирует машину Тьюринга – одну из фундаментальных концепций вычислительной техники. Машина Тьюринга шаг за шагом считывает данные и в зависимости от их значений принимает решения о дальнейших действиях. Теоретически она может решить любую вычислительную задачу. По своей природе молекулы ДНК работают аналогичным образом, распадаясь и рекомбинируя в соответствии с информацией, закодированной в цепочках химических соединений.

Разработанная в Вейцмановском институте установка кодирует входные данные и программы в состоящих из двух цепей молекулах ДНК и смешивает их с двумя ферментами. Молекулы фермента выполняли роль аппаратного, а молекулы ДНК – программного обеспечения. Один фермент расщепляет молекулу ДНК с входными данными на отрезки разной длины в зависимости от содержащегося в ней кода. А другой рекомбинирует эти отрезки в соответствии с их кодом и кодом молекулы ДНК с программой. Процесс продолжается вдоль входной цепи, и, когда доходит до конца, получается выходная молекула, соответствующая конечному состоянию системы. Этот механизм может использоваться для решения самых разных задач. Хотя на уровне отдельных молекул обработка ДНК происходит медленно – с типичной скоростью от 500 до 1000 бит/с, что во много миллионов раз медленнее современных кремниевых процессоров, – по своей природе она допускает массовый параллелизм. По оценкам Шапиро и его коллег, в одной пробирке может одновременно происходить триллион процессов, так что при потребляемой мощности в единицы нановатт может выполняться миллиард операций в секунду.

В 2002 г. фирма Olympus Optical разработала версию ДНК-компьютера, предназначенного для генетического анализа. Он имеет молекулярную и электронную составляющие. Первая осуществляет химические реакции между молекулами ДНК, обеспечивает поиск и выделение результата вычислений. Вторая – обрабатывает информацию и анализирует полученные

результаты. Возможностями биокомпьютеров заинтересовались и военные. Американское агентство по исследованиям в области обороны DARPA выполняет проект, получивший название Bio-Comp (Biological Computations, биологические вычисления). Его цель – создание мощных вычислительных систем на основе ДНК.

Пока до практического применения компьютеров на базе ДНК еще очень далеко. Однако в будущем их смогут использовать не только для вычислений, но и как своеобразные нанофабрики лекарств. Поместив подобное «устройство» в клетку, врачи смогут влиять на ее состояние, исцеляя человека от самых опасных недугов.

Клеточные процессоры. Клеточные процессоры представляют собой самоорганизующиеся колонии различных «умных» микроорганизмов, в геном которых удалось включить некую логическую схему, которая могла бы активизироваться в присутствии определенного вещества. Такие компьютеры очень дешевы в производстве. Им не нужна столь стерильная атмосфера, как при производстве полупроводников.

Главным свойством процессора такого рода является то, что каждая их клетка представляет собой миниатюрную химическую лабораторию. Если биоорганизм запрограммирован, то он просто производит нужные вещества. Достаточно вырастить одну клетку, обладающую заданными качествами, и можно легко и быстро вырастить тысячи клеток с такой же программой. Основная проблема, с которой сталкиваются создатели клеточных биокомпьютеров – организация всех клеток в единую работающую систему. На сегодняшний день практические достижения в области клеточных компьютеров напоминают достижения 20-х годов в области ламповых и полупроводниковых компьютеров. В Лаборатории искусственного интеллекта Массачусетского технологического университета создана клетка, способная хранить на генетическом уровне 1 бит информации. Также разрабатываются технологии, позволяющие единичной бактерии отыскивать своих соседей, образовывать с ними упорядоченную структуру и осуществлять массив параллельных операций.

В 2001 г. американские ученые создали трансгенные микроорганизмы (т. е. микроорганизмы с искусственно измененными генами), клетки которых могут выполнять логические операции И и ИЛИ. Специалисты лаборатории Оук-Ридж, штат Теннесси, использовали способность генов синтезировать тот или иной белок под воздействием определенной группы химических раздражителей. Ученые изменили генетический код бактерий Pseudomonas putida таким образом, что их клетки обрели способность выполнять простые логические операции. Например, при выполнении операции И в клетку подаются два вещества (по сути - входные операнды), под

влиянием которых ген вырабатывает определенный белок. Теперь учеными ведутся работы по созданию на базе этих клеток более сложных логических элементов, а также работы по созданию клетки, выполняющей параллельно несколько логических операций. Потенциал биокомпьютеров очень велик. К достоинствам, выгодно отличающим их от компьютеров, основанных на кремниевых технологиях, относятся:

1) более простая технология изготовления, не требующая для своей реализации столь жестких условий, как при производстве полупроводников;

2) использование не бинарного, а тернарного кода (информация кодируется тройками нуклеотидов), что позволит при меньшем количестве шагов перебрать большее число вариантов при анализе сложных систем;

3) потенциально исключительно высокая производительность, которая может составлять до 1014 операций в секунду за счет одновременного вступления в реакцию триллионов молекул ДНК;

4) возможность хранить данные с плотностью, в триллионы раз превышающей показатели оптических дисков;

5) исключительно низкое энергопотребление.

Однако, наряду с очевидными достоинствами, биокомпьютеры имеют и существенные недостатки, такие как:

1) сложность со считыванием результатов – современные способы определения кодирующей последовательности не совершенны, сложны, трудоемки и дороги;

2) низкая точность вычислений, связанная с возникновением мутаций, прилипанием молекул к стенкам сосудов и т. д.;

3) невозможность длительного хранения результатов вычислений в связи с распадом ДНК в течение времени.

Хотя до практического использования биокомпьютеров еще очень далеко, но предполагается, что, они найдут достойное применение в медицине и фармакологии, а также с их помощью станет возможным объединение информационных и биотехнологий.

Коммуникационные процессоры. *Коммуникационные процессоры* – это микрочипы, являющие собой нечто среднее между жесткими специализированными интегральными микросхемами и гибкими процессорами общего назначения. Коммуникационные процессоры программируются, как и обычные процессоры, но построены с учетом сетевых задач, оптимизированы для сетевой работы, и на их основе производители – как процессоров, так и оборудования – пишут программное обеспечение для специфических приложений.

Коммуникационный процессор имеет собственную память и оснащен высокоскоростными внешними каналами для соединения с другими процессорными узлами. Его присутствие позволяет в значительной мере освободить вычислительный процессор от нагрузки, связанной с передачей

сообщений между процессорными узлами. Скоростной коммуникационный процессор с RISC-ядром позволяет управлять обменом данными по нескольким независимым каналам, поддерживать практически все распространенные протоколы обмена, гибко и эффективно распределять и обрабатывать последовательные потоки данных с временным разделением каналов. Сама идея создания процессоров, предназначенных для оптимизации сетевой работы, и при этом достаточно универсальных для программной модификации – родилась в связи с необходимостью устранить различия в подходах к созданию локальных сетей (различные подходы к архитектуре сети, классификации потоков, и т. д.).

Новая серия коммуникационных процессоров Intel IXP4xx построена на базе распределенной архитектуры XScale и включает мощные мультимедийные возможности, а также развитые сетевые интерфейсы Ethernet. Сочетание высокой производительности и низкого энергопотребления позволяет эффективно применять коммуникационные процессоры Intel не только в классических сетевых приложениях, но и для построения интернет-ориентированных встраиваемых систем промышленного назначения. Эффективность работы промышленных предприятий сегодня напрямую зависит от гибкости применяемых систем автоматизированного управления. Крупные производственные установки требуют использования нескольких децентрализованных систем управления, связанных друг с другом мощной информационной сетью, способной работать в сложных промышленных условиях. Зачастую эти средства промышленной коммуникации призваны обеспечить возможность гибкого управления, программирования и контроля работы распределенных систем управления из удаленных диспетчерских пунктов. Осуществление этих целей возможно с помощью коммуникационных процессоров, предназначенных для подключения персональных компьютеров к промышленным информационным сетям. Дополнительные возможности, обеспечиваемые коммуникационными процессорами должны быть интересны, прежде всего, тем пользователям, которым необходимо осуществлять сложные транзакции или наладить прямую голосовую и видео передачи в рамках сетевой инфраструктуры.

Процессоры с многозначной (нечеткой) логикой. Идея построения процессоров с нечеткой логикой (fuzzy logic) основывается на нечеткой математике. Математическая теория нечетких множеств, предложенная проф. Л. А. Заде, являясь предметом интенсивных исследований, открывает все большие возможности перед системными аналитиками. Основанные на этой теории различные компьютерные системы, в свою очередь, существенно расширяют область применения нечеткой логики.

Подходы нечеткой математики дают возможность оперировать входными данными, непрерывно меняющимися во времени и значениями,

которые невозможно задать однозначно, такими, например, как результаты статистических опросов. В отличие от традиционной формальной логики, известной со времен Аристотеля и оперирующей точными и четкими понятиями типа истина и ложь, да и нет, ноль и единица, нечеткая логика имеет дело со значениями, лежащими в некотором (непрерывном или дискретном) диапазоне.

Функция принадлежности элементов к заданному множеству также представляет собой нежесткий порог «принадлежит – не принадлежит», а плавную сигмоиду, проходящую все значения от нуля до единицы. Теория нечеткой логики позволяет выполнять над такими величинами весь спектр логических операций – объединение, пересечение, отрицание и др. Согласно знаменитой теореме FAT (Fuzzy Approximation Theorem), доказанной Коско, любая математическая система может быть аппроксимирована системой, основанной на нечеткой логике. Свое второе рождение теория нечеткой логики пережила в начале восьмидесятых годов, когда сразу несколько групп исследователей (в основном в США и Японии) всерьез занялись созданием электронных систем различного применения, использующих нечеткие управляющие алгоритмы. Используя преимущества нечеткой логики, заключающиеся в простоте содержательного представления, можно упростить проблему, представить ее в более доступном виде и повысить производительность системы.

Задачи с помощью нечеткой логики решаются по следующему принципу:

1) численные данные (показания измерительных приборов, результаты анкетирования) фаззируются (переводятся в нечеткий формат);

2) обрабатываются по определенным правилам;

3) дефаззируются и в виде привычной информации подаются на выход.

Оказалось возможным создание нечеткого процессора, позволяющего выполнять различные нечеткие операции и приближенные рассуждения (нечеткий вывод) в соответствии с правилами логического вывода. В 1986 г. в AT&T Bell Labs создавались процессоры с «прошитой» нечеткой логикой обработки информации.

В начале 90-х компания Adaptive Logic из США выпустила кристалл, сделанный по аналогово-цифровой технологии (рис. 60). Он позволит сократить сроки конструирования многих встроенных систем управления реального времени, заменив собой традиционные схемы нечетких микроконтроллеров. Аппаратный процессор нечеткой логики второго поколения принимает аналоговые сигналы, переводит их в нечеткий формат, затем, применяя соответствующие правила, преобразует результаты в формат обычной логики и далее – в аналоговый сигнал.

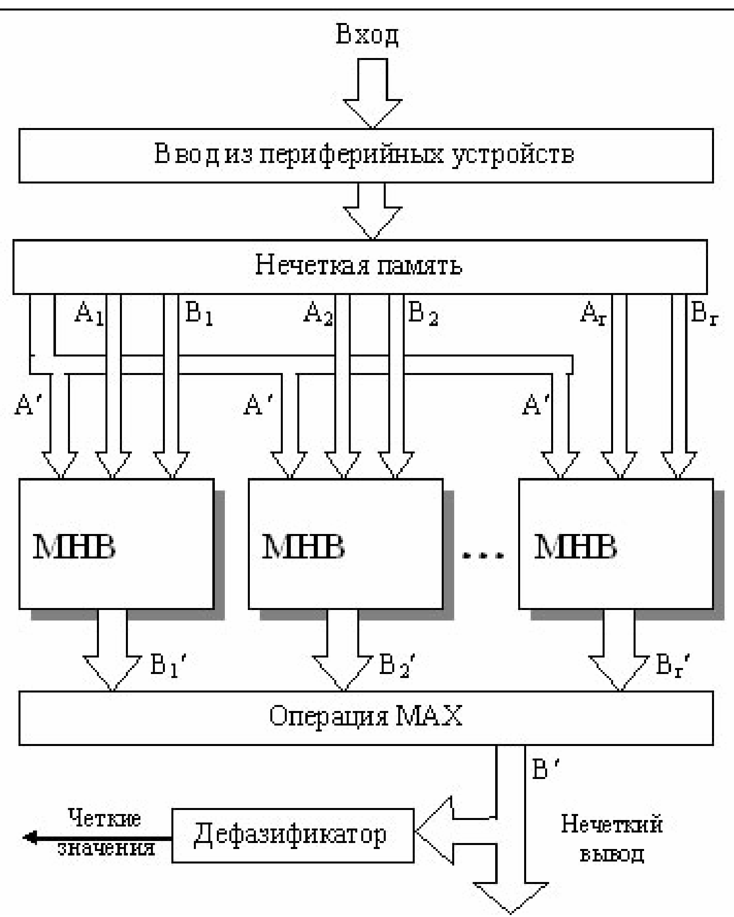


Рис. 60. Программная модель аналогово-цифрового процессора

Все это осуществляется без внешних запоминающих устройств, преобразователей и какого бы ни было программного обеспечения нечеткой логики. Этот микропроцессор относительно прост по сравнению с громоздкими программными обеспечениями. Но так как его основу составляет комбинированный цифровой/аналоговый кристалл, он функционирует на очень высоких скоростях (частота отсчетов входного сигнала – 10 кГц, а скорость расчета – 500 тыс. правил/с), что во многих случаях приводит к лучшим результатам в системах управления по сравнению с более сложными, но медлительными программами.

В Европе и США ведутся интенсивные работы по интеграции fuzzy команд в ассемблеры промышленных контроллеров встроенных устройств (чипы Motorola 68HC11. 12. 21). Такие аппаратные средства позволяют в несколько раз увеличить скорость выполнения приложений и компактность кода по сравнению с реализацией на обычном ядре. Кроме того, разрабатываются различные варианты fuzzy-сопроцессоров, которые контактируют с центральным процессором через общую шину данных, концен-

трируют свои усилия на размывании/уплотнении информации и оптимизации использования правил (продукты Siemens Nixdorf).

Идеи нечеткой логики не являются панацеей и не смогут совершить переворот в компьютерном мире. Нечеткая логика не решит тех задач, которые не решаются на основе логики двоичной, но во многих случаях она удобнее, производительнее и дешевле. Разработанные на ее основе специализированные аппаратные решения (fuzzy-вычислители) позволят получить реальные преимущества в быстродействии. Если каскадировать fuzzy-вычислители, мы получим один из вариантов нейропроцессора или нейронной сети. Во многих случаях эти понятия просто объединяют, называя общим термином «neuro-fuzzy logic».

В настоящее время перспективой использовать процессоры, основанные на нечеткой логике, всерьез заинтересовались военные. Известно, что NASA рассматривает возможность применения (если еще не применяет) нечеткие системы для управления процессами стыковки космических аппаратов.

ЛАБОРАТОРНЫЙ КУРС
7 СЕМЕСТР

Лабораторная работа 1

СУПЕРСКАЛЯРНЫЕ ВЫЧИСЛЕНИЯ. ПАРАЛЛЕЛЬНОЕ ВЫПОЛНЕНИЕ ВЫЧИСЛЕНИЙ

Цель работы: Реализовать программу, демонстрирующую преимущества суперскалярной архитектуры.

Краткие теоретические сведения

Для роста производительности вычислительных машин необходима параллельная выдача нескольких команд в каждом такте. Этот принцип реализуют суперскалярные машины. Суперскалярные машины могут выдавать на выполнение в каждом такте переменное число команд, и работа их конвейеров может планироваться как статически с помощью компилятора, так и с помощью аппаратных средств динамической оптимизации. Суперскалярные машины используют параллелизм на уровне команд путем посылки нескольких команд из обычного потока команд в несколько функциональных устройств. Таким образом, основная идея суперскалярной архитектуры заключается в параллельном выполнении вычислений. В данной лабораторной работе необходимо оценить время последовательного выполнения большого объема вычислений и сравнить это время со временем выполнения этих же вычислений, но уже параллельно. Для реализации параллельных вычислений создадим два потока, каждый из которых будет выполнять половину вычислений. В качестве времени окончания работы параллельных вычислений необходимо выбрать из двух времен максимальное время. Замерять время выполнения операций можно с помощью функции ftime(). Выглядит это следующим образом. С помощью функции ftime() получаем некоторое начальное время выполнения. Затем в цикле выполняем необходимое количество операций и снова получаем значение времени. Далее находим разницу времен после и до цикла. Полученное значение времени и есть время, затраченное процессором на выполнение цикла операций. Оценивать время выполнения цикла операций необходимо в миллисекундах. Вместо функции ftime() можно использовать функцию GetTickCount().

С запуском нового потока в существующем процессе (а первый поток запускается системой при создании процесса) уже связана одна тонкость. В Platform SDK, в разделе, посвященном процессам и потокам, говорится, что для запуска нового потока предназначена API-функция CreateThread. Этой функции надо передать адрес, с которого начнется ис-

полнение потока, иначе говоря, указатель на функцию потока и кучу дополнительных параметров. Можно также использовать функцию стандартной библиотеки _beginthreadex. Ее параметры и возвращаемое значение практически полностью соответствуют CreateThread, хотя есть и небольшие различия. В случае неудачи одна из них возвращает NULL, а другая – 1. Хэндл потока у API функции описан как HANDLE, а у С-функции, как unsigned long, хотя на самом деле это одна и та же величина. Существует еще функция _beginthread (без «ex»). В однопоточной версии стандартной библиотеки функции _beginthread и _beginthreadex вообще отсутствуют. Итак, чтобы стандартная библиотека С могла правильно работать, для запуска нового потока следует использовать _beginthreadex.

Теперь перейдем к вопросу о том, как поток правильно завершить. Прежде всего, хочу обратить особое внимание на то, что для этого никогда нельзя использовать функцию TerminateThread!

Почему же в обычном приложении нельзя использовать TerminateThread, и для чего же эта функция все-таки существует? Все дело в том, что TerminateThread предназначена не для завершения, а для аварийной остановки потока. Она существует лишь на «пожарный случай», когда завершить поток корректно нет возможности, например, если он банально зациклился. Поэтому место этой функции (вместе с TerminateProcess) – лишь в специальных системных приложениях вроде менеджера задач (Task Manager) или отладчика. И даже в таких приложениях к этим функциям стоит прибегать лишь в крайнем случае. Таким образом, если вы используете TerminateThread для уничтожения собственных потоков, это автоматически означает, что ваше приложение работает в аварийном режиме! Соответственно ожидать стабильной и надежной работы от него не приходится.

Чем же грозит насильственное «убийство» потоков? В первую очередь утечкой ресурсов. Почти в каждой программе нам приходится получать от системы различные ресурсы, которые по окончании работы необходимо освободить: память, файлы, объекты графического интерфейса, COM-интерфейсы и т. п. Поэтому если поток прекратил работу раньше времени, то функции, освобождающие эти объекты, не будут вызваны, и ресурсы будут висеть мертвым грузом до самого окончания работы, причем в лучшем случае заказавшего их процесса, а в худшем случае всей операционной системы. Еще больше неприятностей могут доставить эксклюзивно используемые ресурсы, работать с которыми одновременно может только один процесс или поток. Если такой ресурс вовремя не освободить, им не сможет пользоваться вообще никто!

Следует еще раз обратить внимание на стандартную библиотеку С. Она не только сама заказывает некоторые ресурсы при запуске потока/про-

цесса, но еще и устроена так, что должна отслеживать запуск/завершение каждого потока в процессе. Поэтому поток должен завершаться так, чтобы стандартная библиотека правильно отработала это событие.

Наконец, вряд ли кто сможет предсказать, какие возникнут побочные эффекты, если поток будет уничтожен во время вызова API-функции. Впрочем даже если ваш рабочий поток не обращается к системе, добиться, чтобы его аварийная остановка не вызывала побочных эффектов обойдется вам значительно дороже, чем просто использовать штатный механизм завершения потока.

Как же все-таки правильно завершить поток? В Platform SDK сказано, что для этого он сам должен вызвать API-функцию ExitThread. Правда здесь, как и в случае с CreateThread, оказывается, что на самом деле вместо нее следует использовать ее эквивалент из стандартной библиотеки _endthreadex (которая внутри себя вызовет ExitThread). Но, пожалуй, все-таки самый надежный способ завершить поток – это просто дать функции потока завершиться и вернуть управление системе. Исполнение потока начинается с некоего системного кода, которой уже в свою очередь вызывает функцию, указатель на которую вы передали _beginthreadex (или непосредственно CreateThread, если вы не пользуетесь стандартной библиотекой). Как только ваша функция завершит работу, системный код вызовет _endthreadex (или соответственно сразу ExitThread) автоматически. Почему же не советуют явно вызывать _endthreadex? Не забывайте, что перед завершением поток должен освободить все занятые ресурсы. Что касается ресурсов, используемых стандартной библиотекой, их освободит _endthreadex. Но практически любая программа сама использует ресурсы, и за их освобождение ответственность несете вы сами. В большинстве случаев во избежание путаницы освобождать ресурсы удобно в той же самой функции, в которой они были выделены. Поэтому если _endthreadex будет вызвана в одной из вложенных функций, то ресурсы, выделенные в вызывающих функциях, останутся не освобожденными. При программировании на C++ освобождение ресурсов и прочая «чистка» чаще всего происходит в деструкторах. Это удобно, поскольку деструкторы локальных переменных вызываются автоматически при выходе из процедуры, в которой они были созданы. Но если будет вызвана _endthreadex, этого не произойдет! Поэтому с этой функцией стоит быть аккуратным даже в процедуре самого верхнего уровня.

В заключение следует упомянуть функцию CloseHandle. Иногда начинающие ошибочно полагают, что эта функция может остановить поток или процесс. Нет, эта функция предназначена всего лишь для того, чтобы закрывать описатели или «хэндлы», возвращаемые функциями CreateThread,

CreateProcess и некоторыми другими. Хэндлы как потока, так и процесса, остаются в силе даже после того, как поток или процесс был завершен. Как мы видим, это дает возможность легко проверить из другого потока, работает ли еще поток или уже завершился, и даже точно отследить момент его завершения. Поэтому для того, чтобы освободить системные ресурсы, хэндл потока должен быть закрыт явным образом. Для этого и нужна функция CloseHandle. Впрочем, если вы вызовете ее еще до завершения потока, вы всего лишь лишитесь описателя, но никак не повлияете на сам поток.

Задание.

Написать программу на языке программирования C/C++, выполняющую следующие действия:

1. Произвести в цикле порядка одного миллиона операций сложения и замерить время выполнения всего цикла.

2. Разделить все вычисления на две равные части, и запустить их на выполнение параллельно, для чего создать два потока и замерить время работы каждого из них.

3. Выбрать максимальное время из двух поточных времен и сравнить его со временем последовательного выполнения вычислений, сделать выводы о преимуществе использования параллельных вычислений.

Допустимая среда разработки – Borland C++ Builder5.0, Borland C++ Builder6.0, Microsoft Visual C++6.0, Microsoft Visual C++7.0 и т. д.

Порядок выполнения работы

1. Написать программу, реализующую поставленную задачу.
2. Откомпилировать программу.
3. Запустить программу. Убедиться в ее корректной работе.
4. Уметь объяснить работу программы.
5. Оформить отчет.

Содержание отчета

1. Ф.И.О., группа, номер, название лабораторной работы и название дисциплины.
2. Цель работы.
3. Краткое описание алгоритма работы программы.
4. Результат работы программы.
5. Листинг программы.
6. Выводы.

Рекомендуемая литература

1. Дж. Рихтер «Windows для профессионалов».
2. MSDN.

Лабораторная работа 2

ОЦЕНКА ВРЕМЕНИ ВЫПОЛНЕНИЯ АРИФМЕТИЧЕСКИХ ОПЕРАЦИЙ

Цель работы: на конкретном примере определить время выполнения различных арифметических операций для различных операндов, сделать соответствующие выводы.

Краткие теоретические сведения

В данной лабораторной работе требуется оценить скорость выполнения процессором операций с операндами различных типов, например, с целыми числами и с числами с плавающей точкой. Так как оценить время выполнения одной операции для современного процессора проблематично, необходимо подсчитать время выполнения нескольких операций (для современной вычислительной машины порядка одного миллиона операций).

Замерять время выполнения операций можно с помощью функции ftime(). Выглядит это следующим образом. С помощью функции ftime() получаем некоторое начальное время выполнения. Затем в цикле выполняем необходимое количество операций и снова получаем значение времени. Далее находим разницу времен после и до цикла. Полученное значение времени и есть время, затраченное процессором на выполнение цикла операций. Оценивать время выполнения цикла операций необходимо в миллисекундах. Вместо функции ftime() можно использовать функцию GetTickCount().

Программа должна поочередно запускать на выполнение 8 циклов, каждый из которых будет выполнять различные арифметические операции (сложение, вычитание, умножение и деление). Первые четыре цикла будут выполнять операции для операндов первого типа, а вторые четыре для операндов второго типа.

В начале и в конце каждого цикла программа будет запоминать время (в миллисекундах) запуска и остановки этого цикла. Разница между этими величинами будет являться временем выполнения цикла конкретной арифметической операции.

Учитывая высокое быстродействие ЭВМ, используемых при программировании, количество итераций должно быть выбрано достаточно большим для того, чтобы заметить очевидную разницу во времени выполнения различных операций. Использование цикла для оценки времени выполнения требует дополнительных затрат процессорного времени на организацию цикла. Это означает, что сравниваться будут не реальные времена выполнения арифметических операций, а времена выполнения с добавленными к ним временными задержками на организацию циклов. Таким образом, число итераций для всех циклов должно быть выбрано одинаковым

для того, чтобы оценить реальную зависимость во времени выполнения операций в одинаковых условиях. Т. е. это означает, что если для всех операций используется одинаковое количество итераций, то и задержки на организацию циклов для всех операций одинаковы. Это в свою очередь означает что мы можем сопоставлять полученные результаты и анализировать закономерности затрат процессорного времени на выполнение арифметических операций без необходимости вычисления времени задержки на организацию циклов.

Программа должна выводить на экран результаты всех измерений. Приблизительный возможный внешний вид результатов работы программы представлен ниже на рисунке 1. Как видно из рисунка операции сложения, вычитания и умножения значительно отличаются по времени выполнения от операции деления. Кроме того, есть различия во времени выполнения между целочисленными операндами и операндами с плавающей точкой.

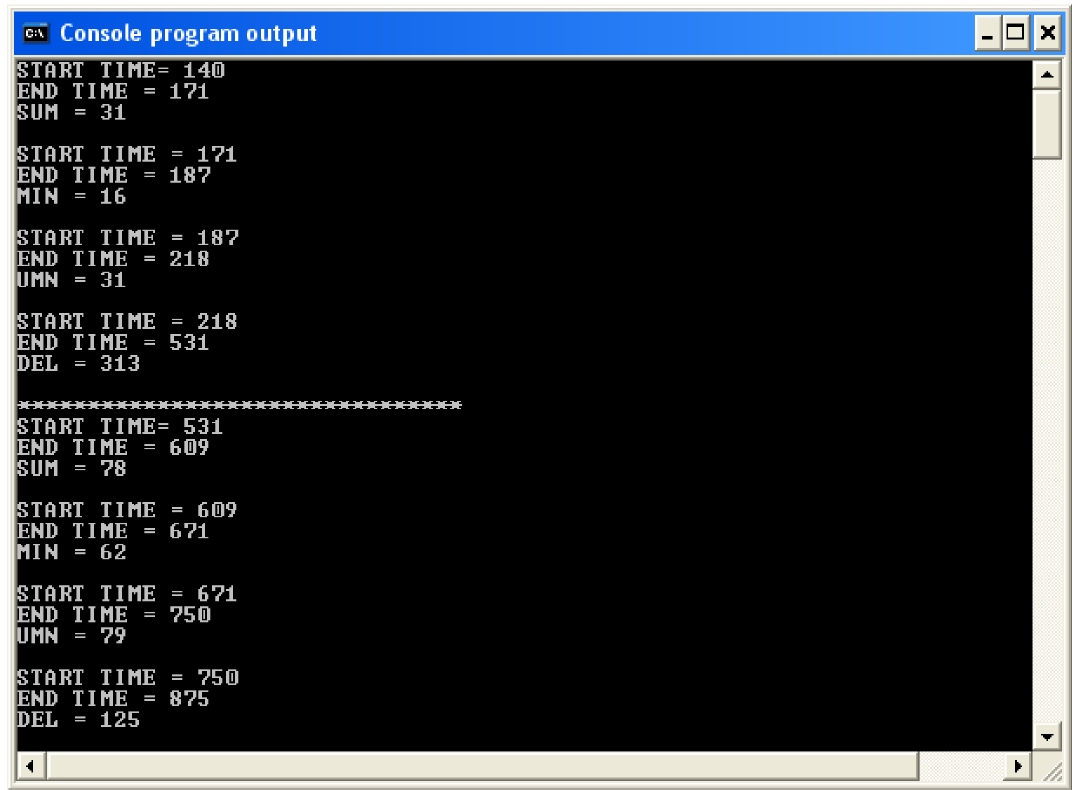


Рис. 1. Возможный вид программы

Задание.

Написать программу для оценки времени выполнения арифметических операций (сложение, вычитание, умножение, деление) для целочисленных операндов и для операндов с плавающей точкой на языке программирования C/C++. Допустимая среда разработки – Borland C++ Builder5.0, Borland C++ Builder6.0, Microsoft Visual C++6.0, Microsoft Visual C++7.0 и т. д.

Порядок выполнения работы

1. Написать программу, реализующую поставленную задачу.
2. Откомпилировать программу.
3. Запустить программу. Убедиться в ее корректной работе.
4. Уметь объяснить работу программы.
5. Оформить отчет.

Содержание отчета

1. Ф.И.О., группа, номер, название лабораторной работы и название дисциплины.
2. Цель работы.
3. Краткое описание алгоритма работы программы.
4. Результат работы программы.
5. Листинг программы.
6. Выводы.

Лабораторная работа 3

ВЫЧИСЛЕНИЯ С ИСПОЛЬЗОВАНИЕМ MMX РЕГИСТРОВ

Цель работы: Написать программу, для исследования возможности ускорения вычислительного процесса при использовании ОКМД команд (MMX).

Краткие теоретические сведения

Типы данных

Основными типами данных для набора инструкций IA MMX являются: упакованный целочисленный с фиксированной запятой, где несколько чисел сгруппированы в одно 64-х разрядное значение. Эти 64-х разрядные значения загружаются в регистры MMX. Позиция десятичной точки является подразумеваемой, что обеспечивает максимальную гибкость. Поддерживаются знаковые и беззнаковые числа: байт, слово, двойное и учетверенное слово (byte, word, doubleword and quadword).

Есть четыре MMX типа:

1) упакованный байт – 8 байт, упакованных в одно 64-битное значение;

2) упакованное слово – 4 слова, упакованных в одно 64-битное значение;

3) упакованное двойное слово – два 32-х битных двойных слова, упакованных в одно 64-битное значение;

4) учетверенное слово – одно 64-битное значение.

Например, цвет пикселя обычно представляется 8-битным целым (байтом). Используя технологию MMX можно упаковать 8 пикселей в од-

но 64-ти битное значение и загрузить в регистр MMX. Затем команды MMX будут сразу обрабатывать все 8 пикселей изображения.

Packed byte (eight 8-bit elements)
63 56 55 48 47 40 39 32 31 24 23 16 15 8 7

Packed word (four 16-bit elements)
63 48 47 32 31 16 15

Packed doubleword (two 32-bit elements)
63 32 31

Quadword (64-bit element)
63

Совместимость

Технология MMX создавалась как полностью совместимая со всем предыдущим программным обеспечением, в том числе операционными системами. Поэтому при ее реализации не было добавлено новых регистров или состояний процессора. Для расположения 8-и регистров MMX использованы регистры сопроцессора, что позволило оставить без изменения стандартные механизмы переключения контекстов задач в существующих операционных системах.

Инструкции сопроцессора, отвечающие за сохранение/восстановление состояния FPU, также обрабатывают и состояние расширения MMX. Таким образом, MMX не добавило новых исключений или состояний процессора.

Инструкции

Инструкции MMX охватывают несколько областей:

1) базовые арифметические операции, такие как add – сложение, sub – вычитание, mul – умножение и mul-add – умножение-сложение;

2) операции сравнения;

3) операции преобразования типов, упаковка/рапаковка данных;

4) логические операции типа AND, AND NOT,OR и XOR;

5) операции сдвига;

6) операции пересылки данных (MOV).

Арифметические и логические инструкции поддерживают различные типы упакованных данных. С учетом этого все расширение MMX включает 57 кодов операций.

Краткий обзор набора инструкций MMX™

Инструкции и соответствующие мнемоники в таблице сгруппированы по категориям.

Если инструкция поддерживает тип данный байт (B), слово (W), двойное слово (DW) или учетверенное слово (QW), соответствующие аб-

бревиатуры указаны в квадратных скобках. Например, для базовой мнемоники PADD (packed add) возможны следующие варианты: PADDB, PADDW и PADDD.

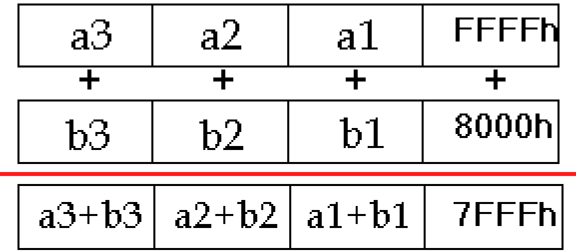
Категория	**Мнемоника**	**Описание**
Арифметические операции	PADD[B,W,D]	Сложение (*Add with wrap-around on [byte, word, doubleword]*)
	PADDS[B,W]	Сложение знаковое с насыщением (*Add signed with saturation on [byte, word]*)
	PADDUS[B,W]	Сложение беззнаковое с насыщением (*Add unsigned with saturation on [byte, word]*)
	PSUB[B,W,D]	Вычитание (*Subtract with wrap-around on [byte, word, doubleword]*)
	PSUBS[B,W]	Вычитание знаковое с насыщением (*Subtract signed with saturation on [byte, word]*)
	PSUBUS[B,W]	Вычитание беззнаковое с насыщением (*Subtract unsigned with saturation on [byte, word]*)
	PMULHW	Packed multiply high on words
	PMULLW	Packed multiply low on words
	PMADDWD	Умножение упакованных байт и суммирование полученных пар слов (*Packed multiply on words and add resulting pairs*)
Операции сравнения	PCMPEQ[B,W,D]	Сравнение упакованных на равенство (*Packed compare for equality [byte, word,doubleword]*)
	PCMPGT[B,W,D]	Сравнение упакованных на больше (*Packed compare greater than [byte, word, doubleword]*)
Преобразование разрядности	PACKUSWB	Упаковка слов в байты (беззнаковое с насыщением) *Pack words into bytes (unsigned with saturation)*
	PACKSS[WB,DW]	Упаковка [слов в байты, двойных слов в слова] (знаковое с насыщением) *Pack [words into bytes, doublewords into words] (signed with saturation)*
	PUNPCKH [BW,WD,DQ]	Распаковка старших [байт, слов, двойных слов] из MMX регистра. *Unpack (interleave) high-order [bytes, words, doublewords] from MMX register*
	PUNPCKL [BW,WD,DQ]	Распаковка младших [байт, слов, двойных слов] из MMX регистра. *Unpack (interleave) low-order [bytes, words, doublewords] from MMX register*
Логические	PAND	Битовый (*битwise*) AND
	PANDN	Битовый (*битwise*) AND NOT
	POR	Битовый (*битwise*) OR
	PXOR	Битовый (*битwise*) XOR
Сдвиги	PSLL[W,D,Q]	Логический сдвиг влево упакованных [слов, двойных слов, учетверенного слова] на значе-

Категория	Мнемоника	Описание
		ние указанное в MMX регистре или в команде *Packed shift left logical [word, doubleword, quadword] by amount specified in MMX register or by immediate value*
	PSRL[W,D,Q]	Логический сдвиг вправо упакованных [слов, двойных слов, учетверенного слова] на значение указанное в MMX регистре или в команде *Packed shift right logical [word, doubleword, quadword] by amount specified in MMX register or by immediate value*
	PSRA[W,D]	Арифметический сдвиг вправо упакованных [слов, двойных слов, учетверенного слова] на значение указанное в MMX регистре или в команде *Packed shift right arithmetic [word, doubleword] by amount specified in MMX register or by immediate value*
Пересылки данных	MOV[D,Q]	Пересылка [двойного или учетверенного] слова в/из MMX регистра *Move [doubleword, quadword] to MMX register or from MMX register*
Управление состоянием FPU и MMX	EMMS	Сброс состояния FPU, переключение между MMX и FPU режимами *Empty MMX state*

Примеры инструкций

В этом разделе кратко описываются несколько примеров MMX инструкций. Для примера взяты операции с 16-битными словами, однако большинство этих операций существуют и для байт и двойных слов.

Пример 1 иллюстрирует сложение упакованных слов без насыщения. Вычисляются четыре суммы для восьми 16-битных элементов, все они выполняются независимо и параллельно. В данном случае результат самого правого сложения превышает значение, которое может быть представлено 16-ю битами, т. е. происходит переполнение. Результатом сложения FFFFh + 8000h будет 17-битное число. 17-й бит теряется из-за усечения результата, соответственно получается число 7FFFh.



Пример 1. PADD[W]: Сложение без насыщения

В примере 2 показано сложение упакованных слов с беззнаковым насыщением. Используются те же данные, что и в предыдущем примере. Самое правое сложение дает результат, не помещающийся в 16 бит, следовательно, происходит насыщение. Термин «насыщение» означает, что при переполнении сложения (или вычитания) результат ограничен максимальным (или минимальным) числом, представимым в данной разрядности. Для беззнаковых 16-битных слов максимальным и минимальным будут FFFFh и 0000h, для знаковых это 7FFFh и 8000h соответственно. Насыщение может оказаться важным при работе с пикселями изображения, когда переполнение может вызвать появление точек черного цвета.

a3	a2	a1	FFFFh
+	+	+	+
b3	b2	b1	8000h
a3+b3	a2+b2	a1+b1	FFFFh

Пример 2. PADDUS[W]: Сложение с беззнаковым насыщением

Пример 3 показывает работу инструкции PMADDWD, выполняющую умножить-сложить операцию. Эта операция часто используется в алгоритмах обработки сигналов. Эта команда меняет разрядность результата – ее операнды имеют размер слова (16 бит), а результат двойного слова (32 бита).

a3	a2	a1	a0
*	*	*	*
b3	b2	b1	b0
a3*b3+a2*b2		a1*b1+a0*b0	

Пример 3. PMADDWD: 16b x 16b -> 32b Операция умножить-сложить

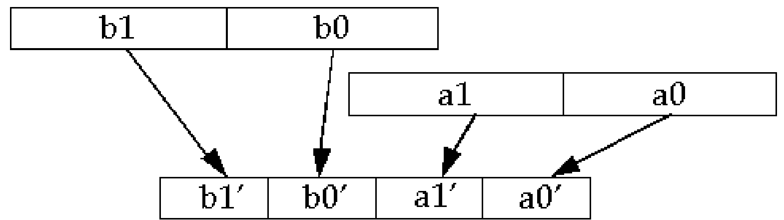
Пример 4 иллюстрирует параллельное сравнение упакованных данных. Команда сравнивает четыре пары 16-битных слов. В регистр результата заноситься true/истина (FFFFh) или false/ложь (0000h) отдельно для каждого сравнения. На рисунке можно видеть сравнение по условию «больше» («greater than»). Эта команда не изменяет флагов процессора.

23	45	16	34
gt ?	gt ?	gt ?	gt ?
31	7	16	67
0000h	FFFFh	0000h	0000h

Пример 4. PCMPGT[W]: Параллельные сравнения

Результат такого сравнения может быть использован как маска для выбора элементов из другого входного потока с использованием логических операций. Такой метод позволяет в какой-то мере сократить количество инструкций перехода.

Последний пример (пример 5) показывает механизм работы инструкции упаковки (сокращения разрядности). На вход подаются четыре 32-битных двойных слова в двух различных регистрах, которые упаковываются в четыре 16-битных слова в одном регистре. Если значение исходного операнда (например, b1) превосходит разрядность результата (b1′), преобразование выполняется с насыщением.



Пример 5. PACKSS[DW]: Инструкция упаковки

Инструкции упаковки/распаковки обычно используются, когда разрядность промежуточных результатов вычислений превосходит разрядность исходных данных.

Задание.

Написать программу на языке программирования C/C++ с использованием ассемблерных вставок, выполняющую следующие действия: произвести в цикле порядка одного миллиона операций сложения и замерить время выполнения, затем сделать все то же самое для операции вычитания и умножения с использованием MMX регистров; произвести те же действия с целочисленными операндами без использования MMX регистров по аналогии со второй лабораторной работой; сравнить времена расчетов, сделать выводы о преимуществе использования MMX регистров.

Допустимая среда разработки – Borland C++ Builder5.0, Borland C++ Builder6.0, Microsoft Visual C++6.0, Microsoft Visual C++7.0 и т. д.

Порядок выполнения работы

1. Написать программу, реализующую поставленную задачу.
2. Откомпилировать программу.
3. Запустить программу. Убедиться в ее корректной работе.
4. Уметь объяснить работу программы.
5. Оформить отчет.

Содержание отчета

1. Ф.И.О., группа, номер, название лабораторной работы и название дисциплины.

2. Цель работы.
3. Краткое описание алгоритма работы программы.
4. Результат работы программы.
5. Листинг программы.
6. Выводы.

Рекомендуемая литература

1. В. Юров «Assembler. Учебник»;
2. В. Юров «Assembler. Краткий справочник команд»;
3. MSDN.

Лабораторная работа 4

ОПРЕДЕЛЕНИЕ РАЗМЕРОВ КЭШ-ПАМЯТИ НА ОСНОВАНИИ ВЫЧИСЛЕННОГО ВРЕМЕНИ ДОСТУПА

Цель работы: Написать программу для определения размеров кэш-памяти различных уровней иерархии на основании вычисленного времени доступа.

Краткие теоретические сведения

Концепция кэш-памяти возникла довольно давно и сегодня кэш-память имеется практически в любом классе компьютеров, и чаще всего многоуровневая. В таблице 1 представлен типичный набор параметров, который используется для описания кэш-памяти.

Таблица 1

Типовые значения ключевых параметров для кэш-памяти персональных компьютеров

Параметр	Значение
Размер блока (строки)	4 – 128 байт
Время попадания (hit time)	1 – 4 такта синхронизации (обычно 1 такт)
Потери при промахе (miss penalty) (Время доступа – access time) (Время пересылки – transfer time)	8 – -32 такта синхронизации (6 – 10 тактов синхронизации) (2 – 22 такта синхронизации)
Доля промахов (miss rate)	1% – 20%
Размер кэш-памяти	4 Кбайт – 16 Мбайт

Рассмотрим организацию кэш-памяти более детально. Принципы размещения блоков в кэш-памяти определяют три основных типа их организации.

Если каждый блок основной памяти имеет только одно фиксированное место, на котором он может появиться в кэш-памяти, то такая кэш-память называется кэшем с прямым отображением (direct mapped). Это наиболее простая организация кэш-памяти, при которой для отображения адресов блоков основной памяти на адреса кэш-памяти просто используются младшие разряды адреса блока. Таким образом, все блоки основной

памяти, имеющие одинаковые младшие разряды в своем адресе, попадают в один блок кэш-памяти, т. е.

(адрес блока кэш-памяти) = (адрес блока основной памяти) mod (число блоков в кэш-памяти)

Если некоторый блок основной памяти может располагаться на любом месте кэш-памяти, то кэш называется полностью ассоциативным (fully associative).

Если некоторый блок основной памяти может располагаться на ограниченном множестве мест в кэш-памяти, то кэш называется множественно-ассоциативным (set associative). Обычно множество представляет собой группу из двух или большего числа блоков в кэше. Если множество состоит из n блоков, то такое размещение называется множественно-ассоциативным с n каналами (n-way set associative). Для размещения блока прежде всего необходимо определить множество. Множество определяется младшими разрядами адреса блока памяти (индексом):

(адрес множества кэш-памяти) =

(адрес блока основной памяти) mod (число множеств в кэш-памяти)

Далее, блок может размещаться на любом месте данного множества.

Диапазон возможных организаций кэш-памяти очень широк: кэш-память с прямым отображением есть просто одноканальная множественно-ассоциативная кэш-память, а полностью ассоциативная кэш-память с m блоками может быть названа m-канальной множественно-ассоциативной.

У каждого блока в кэш-памяти имеется адресный тег, указывающий, какой блок в основной памяти данный блок кэш-памяти представляет. Эти теги обычно одновременно сравниваются с выработанным процессором адресом блока памяти. Кроме того, необходим способ определения того, что блок кэш-памяти содержит достоверную или пригодную для использования информацию. Наиболее общим способом решения этой проблемы является добавление к тегу так называемого бита достоверности (valid бит).

Адресация множественно-ассоциативной кэш-памяти осуществляется путем деления адреса, поступающего из процессора, на три части: поле смещения используется для выбора байта внутри блока кэш-памяти, поле индекса определяет номер множества, а поле тега используется для сравнения. Если общий размер кэш-памяти зафиксировать, то увеличение степени ассоциативности приводит к увеличению количества блоков в множестве, при этом уменьшается размер индекса и увеличивается размер тега.

При возникновении промаха, контроллер кэш-памяти должен выбрать подлежащий замещению блок. Польза от использования организации с прямым отображением заключается в том, что аппаратные решения здесь наиболее простые. Выбирать просто нечего: на попадание проверяется только один блок и только этот блок может быть замещен. При полностью

ассоциативной или множественно-ассоциативной организации кэш-памяти имеются несколько блоков, из которых надо выбрать кандидата в случае промаха. Как правило, для замещения блоков применяются две основных стратегии: случайная и LRU.

В первом случае, чтобы иметь равномерное распределение, блоки-кандидаты выбираются случайно. В некоторых системах, чтобы получить воспроизводимое поведение, которое особенно полезно во время отладки аппаратуры, используют псевдослучайный алгоритм замещения. Во втором случае, чтобы уменьшить вероятность выбрасывания информации, которая скоро может потребоваться, все обращения к блокам фиксируются. Заменяется тот блок, который не использовался дольше всех (LRU – Least-Recently Used). Достоинство случайного способа заключается в том, что его проще реализовать в аппаратуре. Когда количество блоков для поддержания трассы увеличивается, алгоритм LRU становится все более дорогим и часто только приближенным. В таблице 2 показаны различия в долях промахов при использовании алгоритма замещения LRU и случайного алгоритма.

Таблица 2

Сравнение долей промахов для алгоритма LRU и случайного алгоритма замещения при нескольких размерах кэша и разных ассоциативностях при размере блока 16 байт

Ассоциативность:	2-канальная		4-канальная		8-канальная	
Размер кэш-памяти	LRU	Random	LRU	Random	LRU	Random
16 KB	5.18%	5.69%	4.67%	5.29%	4.39%	4.96%
64 KB	1.88%	2.01%	1.54%	1.66%	1.39%	1.53%
256 KB	1.15%	1.17%	1.13%	1.13%	1.12%	1.12%

При обращениях к кэш-памяти на реальных программах преобладают обращения по чтению. Все обращения за командами являются обращениями по чтению и большинство команд не пишут в память. Обычно операции записи составляют менее 10% общего трафика памяти. Желание сделать общий случай более быстрым означает оптимизацию кэш-памяти для выполнения операций чтения, однако при реализации высокопроизводительной обработки данных нельзя пренебрегать и скоростью операций записи. К счастью, общий случай является и более простым. Блок из кэш-памяти может быть прочитан в то же самое время, когда читается и сравнивается его тег. Таким образом, чтение блока начинается сразу, как только становится доступным адрес блока. Если чтение происходит с попаданием, то блок немедленно направляется в процессор. Если же происходит промах, то от заранее считанного блока нет никакой пользы, правда нет и никакого вреда. Однако при выполнении операции записи ситуация коренным образом меняется. Именно процессор определяет размер записи (обычно от 1 до 8 байтов), и только эта часть блока может быть изменена. В

общем случае это подразумевает выполнение над блоком последовательности операций чтение-модификация-запись: чтение оригинала блока, модификацию его части и запись нового значения блока. Более того, модификация блока не может начинаться до тех пор, пока проверяется тег, чтобы убедиться в том, что обращение является попаданием. Поскольку проверка тегов не может выполняться параллельно с другой работой, то операции записи отнимают больше времени, чем операции чтения.

Очень часто организация кэш-памяти в разных машинах отличается именно стратегией выполнения записи. Когда выполняется запись в кэш-память, имеются две базовые возможности. Сквозная запись (write through, store through) – информация записывается в два места: в блок кэш-памяти и в блок более низкого уровня памяти. Запись с обратным копированием (write back, copy back, store in) – информация записывается только в блок кэш-памяти. Модифицированный блок кэш-памяти записывается в основную память только тогда, когда он замещается. Для сокращения частоты копирования блоков при замещении обычно с каждым блоком кэш-памяти связывается так называемый бит модификации (dirty бит). Этот бит состояния показывает, был ли модифицирован блок, находящийся в кэш-памяти. Если он не модифицировался, то обратное копирование отменяется, поскольку более низкий уровень содержит ту же самую информацию, что и кэш-память.

Когда процессор ожидает завершения записи при выполнении сквозной записи, то говорят, что он приостанавливается для записи (write stall). Общий прием минимизации остановки по записи связан с использованием буфера записи (write buffer), который позволяет процессору продолжить выполнение команд во время обновления содержимого памяти. Следует отметить, что остановки по записи могут возникать и при наличии буфера записи.

При промахе во время записи имеются две дополнительные возможности:

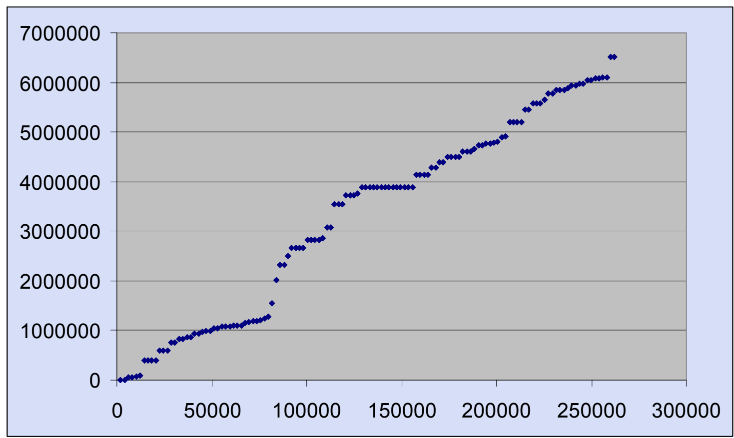
1. Разместить запись в кэш-памяти (write allocate) (называется также выборкой при записи (fetch on write)). Блок загружается в кэш-память, вслед за чем выполняются действия аналогичные выполняющимся при выполнении записи с попаданием. Это похоже на промах при чтении.

2. Не размещать запись в кэш-памяти (называется также записью в окружение (write around)). Блок модифицируется на более низком уровне и не загружается в кэш-память.

Обычно в кэш-памяти, реализующей запись с обратным копированием, используется размещение записи в кэш-памяти (в надежде, что последующая запись в этот блок будет перехвачена), а в кэш-памяти со сквозной записью размещение записи в кэш-памяти часто не используется (поскольку последующая запись в этот блок все равно пойдет в память).

В данной лабораторной работе необходимо определить размер кэш-памяти различных уровней иерархии на основании вычисленного времени доступа. Фактически это означает следующее. Необходимо выделить не-

кий объем данных, например 1кбайт, и скопировать его при помощи функции memcpy() в цикле несколько раз (количество итераций цикла определяется экспериментально в зависимости от производительности компьютера, на котором будет запускаться программа). После первой итерации копируемые данные занесутся в кэш-память первого уровня, и при всех последующих операциях копирования будет происходить «попадание» в кэш-память первого уровня, а это значит, что копируемые данные будут извлекаться из кэш-памяти. Так как кэш-память работает очень быстро, то время работы цикла копирования будет очень мало. После окончания работы цикла копирования необходимо увеличить размерность копируемых данных, например, на 1кбайт и снова запустить цикл копирования. Увеличение размерности копируемых данных естественно повлечет за собой увеличение времени копирования, однако прирост времени будет незначителен, т. к. данные все еще берутся из кэш-памяти. Наращивая размерность копируемых данных, мы постепенно превысим размерность кэш-памяти первого уровня, т. е. все копируемые данные уже не будут помещаться в кэш-память первого уровня и, следовательно, обращение за ними будет происходить в кэш-память второго уровня. Обращение к кэш-памяти второго уровня требует на порядок больше времени, чем обращение к кэш-памяти первого уровня, а это значит, что время выполнения цикла копирования начнет резко возрастать. Подобная закономерность будет наблюдаться и при переходе из кэш-памяти второго уровня к оперативной памяти. Следовательно, необходимо наращивать объем копируемых данных от 1кбайта до 1000кбайт (т. к. некоторые современные процессора имеют кэш-память второго уровня от 512кбайт и выше) и сохранять все значения времен выполнения циклов копирования в текстовый файл. Затем необходимо построить в Excel график зависимости времен выполнения циклов копирования от размерности копируемых данных. Приблизительный пример подобного графика представлен на рисунке внизу.



Глядя на этот график можно увидеть, что резкий прирост времени выполнения начинается, где-то в районе 70кбайт. Из стандартных значе-

ний размерности кэш-памяти первого уровня подходит значение 64кбайта, следовательно, кэш-память первого уровня равна 64кбайта. Так как больше резких скачков на графике не наблюдается можно сделать вывод, что кэш-память второго уровня превышает 250кбайт, а это означает в свою очередь необходимость дальнейшего наращивания размерности копируемых данных.

Задание.

Необходимо написать программу, которая будет замерять времена выполнения операций копирования данных различных объемов с определенным шагом наращивания размерности копируемой информации и сохранение этих величин в файл на языке программирования C/C++. Затем, используя полученные значения времен копирования построить в Excel график зависимости времен копирования от объемов копируемых данных. На основании построенного графика, сделать выводы о размерности кэш-памяти различных уровней иерархии.

Порядок выполнения работы

1. Написать программу, реализующую поставленную задачу.
2. Откомпилировать программу.
3. Запустить программу. Убедиться в ее корректной работе.
4. Уметь объяснить работу программы.
5. Оформить отчет.

Содержание отчета

1. Ф.И.О., группа, номер, название лабораторной работы и название дисциплины.
2. Цель работы.
3. Краткое описание алгоритма работы программы.
4. Результат работы программы.
5. Листинг программы.
6. Выводы.

8 СЕМЕСТР.

Лабораторная работа 1

ОСНОВЫ РАБОТЫ С DSP ПРОЦЕССОРОМ TMS320VC5510

1. Общие сведения

Реализация любой системы цифровой обработки сигналов (ЦОС) на базе цифровых сигнальных процессоров (ЦСП) включает:

1) разработку и отладку программного кода на языке высокого уровня или ассемблере в соответствии с выбранным алгоритмом. Результатом является набор текстовых файлов с программным кодом, описывающим реализуемый алгоритм;

2) оптимизацию программного кода с учетом архитектуры выбранного ЦСП. Результатом является один бинарный файл, содержащий цифровое двоичное представление разработанного программного код;

3) тестирование разработанного программного код;

4) разработку платы для проектируемой системы ЦО;

5) тестирование программного кода на разработанной плате.

Для выполнения всех вышеперечисленных этапов необходимо иметь комплекс программных и аппаратных средств проектирования систем ЦОС (программное обеспечение и средства аппаратной поддержки). Программное обеспечение (ПО) включает:

1. Текстовый редактор для написания программного кода.
2. Компилятор для трансляции файлов с кодом программы в бинарный файл.
3. Симулятор ЦСП для отладки кода в виртуальном режиме.
4. Загрузчик для переноса полученного бинарного кода программы из файла в память процессора.
5. Отладчик для тестирования программы.
6. Средства аппаратной поддержки (САП) представляют собой:

1) персональный компьютер (ПК) для реализации возможностей ПО;

2) отладочный модуль для тестирования бинарного файла с кодом программы;

3) программатор для переноса разработанного кода в память ЦОС;

4) эмулятор для тестирования разработанной платы.

Для реализации систем ЦОС фирма Texas Instruments (TI) предлагает ряд технологических решений, позволяющих существенно ускорить и облегчить процесс разработки:

1) единый, интегрированный в каждый ЦСП фирмы TI отладочный интерфейс Joint Test Action Group (JTAG), позволяющий подключить не-

сколько ЦСП к одному отладчику всего по 6 проводам и обеспечивающий высокую скорость объмена данными;

2) технологию обмена данными в реальном времени – Real Time Data Exchange (RTDX), базирующуюся на интерфейсе JTAG;

3) библиотеку функций DSP/BIOS, использующую технологию RTDX и позволяющую реализовать алгоритмы обмена данными в реальном времени;

4) стандарт eXpresDSP, определяющий основные правила написания программного кода для реализации алгоритмов ЦОС на ЦСП, позволяющий упростить разработку программ и увеличить повторное использование кода.

Программная часть этих технологий воплотилась в интегрированной среде разработки CCS (Code Composer Studio), которая включает:

1) текстовый редактор;

2) менеджер проектов;

3) оптимизирующий компилятор;

4) симулятор ЦСП;

5) загрузчик программного кода в ЦСП;

6) мультипроцессорный отладчик, оптимизированный для приложений ЦОС, который содержит уникальный набор средств анализа и отладки программ в реальном времени, базирующийся на RTDX и DSP/BIOS.

Аппаратным элементом являются платы двух типов:

1) внутрисхемный эмулятор;
2) отладочный модуль.

Внутрисхемный эмулятор представляет модуль, устанавливаемый в слот PCI или подключаемый к персональному компьютеру при помощи порта USB или LPT, и кабель для подключения к отлаживаемой плате. Все ЦСП фирмы TI имеют единый интерфейс внутрисхемного эмулятора JTAG и работают с единым внутрисхемным эмулятором XDS-510 (или SDSP-510). Через отладочный интерфейс доступны как внутренние регистры процессора, так и вся память и периферия. Эмулятор позволяет:

1) производить загрузку кода программы и данных во внутреннюю и внешнюю память;

2) устанавливать любое количество точек остановки;

3) производить контроль и модификацию содержимого памяти, регистров процессора и регистров периферийных устройств;

4) проводить пошаговое выполнение программы;

5) измерять время выполнения программы или ее частей;

6) работать с несколькими цифровыми процессорами для обработки сигналов (ЦПОС).

Подключение внутрисхемного эмулятора абсолютно «прозрачно» для исполняемой программы и не оказывает на выполнение никакого

влияния, при этом программа исполняется в реальном времени, без каких-либо задержек и ограничений по производительности. Такой подход радикально отличается от традиционного, при котором предполагается подключение на место процессора либо специальной микросхемы – прототипа, либо специального устройства – эмулятора ЦСП.

Отладочный модуль выполняется в виде отдельной платы, которая содержит:

1) один из типов ЦПОС;
2) внешнюю память;
3) источник питания;
4) JTAG контроллер;
5) разъемы для подключения дочерних плат;
6) разъемы подключения к ПК;
7) дополнительные элементы, такие как АЦП/ЦАП, контроллеры сети, аудио-, видео- кодеки, адаптеры PCI, шины и т. д.

Отладочный модуль может подключаться к ПК как через внутрисхемный эмулятор, так и напрямую через порты LPT или USB. Однако во втором случае обмен данными между ПК и отладочным модулем происходит значительно медленнее.

Ассортимент предлагаемых отладочных модулей очень широк: от плат начального уровня до специализированных устройств, позволяющих построить программно-аппаратные комплексы реализации алгоритмов обработки аудио и видеоданных, сетевые протоколы и т. д. На рисунке 1.1 показаны возможные структуры программно-аппаратного комплекса.

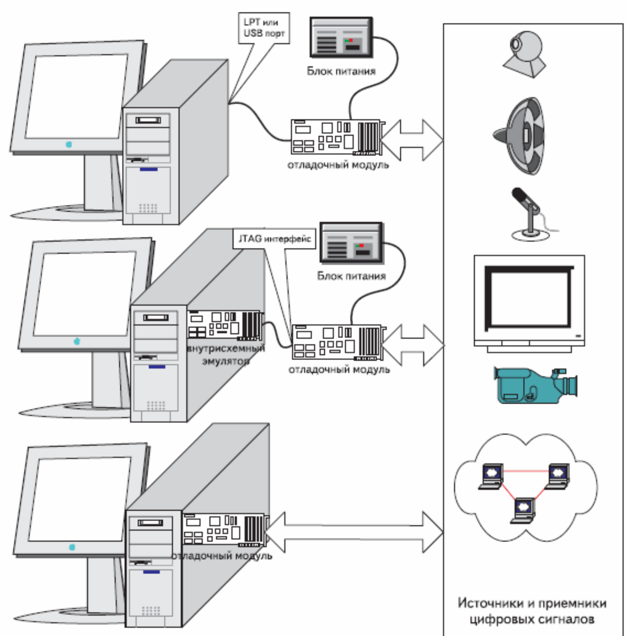


Рис. 1.1 Структуры программно-аппаратного комплекса

Предлагаемые ПО и САП позволяют объединить весь процесс разработки систем ЦОС и выделить всего три этапа:

1) первый – разработка и отладка программного кода с использованием ИСР CCS и одного из наиболее соответствующих поставленной задаче отладочных модулей;

2) второй – разработка платы для проектируемой системы ЦОС;

3) третий – отладка программного обеспечения на разработанной плате с использованием CCS и внутрисхемного эмулятора.

Таким образом, для полноценной работы с ЦСП фирмы TI необходимы только три компонента:

1) один из отладочных модулей, в наибольшей степени пригодный для решения поставленной задачи;

2) один наиболее подходящий по характеристикам внутрисхемный эмулятор;

3) интегральная среда разработки (ИСР) – Code Composer Studio (CCS).

Использование ПО и САП фирмы TI позволяет оценить разрабатываемую систему ЦОС и отработать ее решения на реальном ЦСП на самых ранних стадиях проектирования. В лабораторных работах будут использоваться TMS320VC5510 DSK (DSK5510) (рис. 1.2).

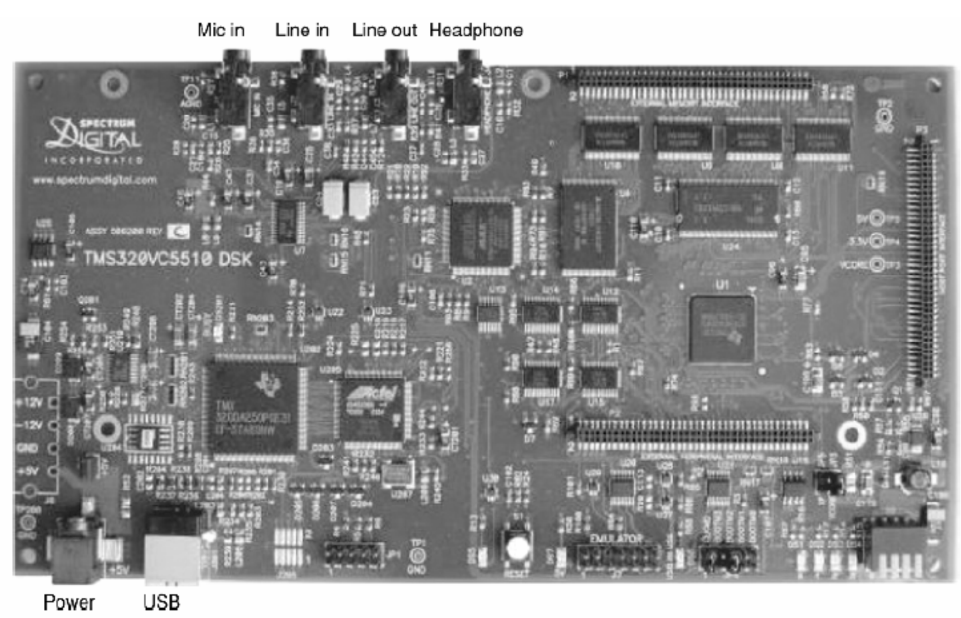


Рис. 1.2 TMS320VC5510 DSK (DSK5510)

2. Настройка и запуск Code Composer Studio (simulation)

После инсталляции CCS при первом запуске программы вначале необходимо произвести настройку ИСР для работы в режиме симуляции ЦСП для отладки кода в виртуальном режиме:

1) проверить установлены ли драйвера для платы DSK5510, которые располагаются в папке: C:\CCStudio_v3.1\specdig\ xds510usb\;

2) запустить утилиту «Setup CCStudio v3.1»(ярлык расположен на рабочем столе), (рис. 2.1). Эта утилита предлагает выбрать плату отладочного модуля, которая будет установлена в системе, и с которой будет работать ИСР CCS;

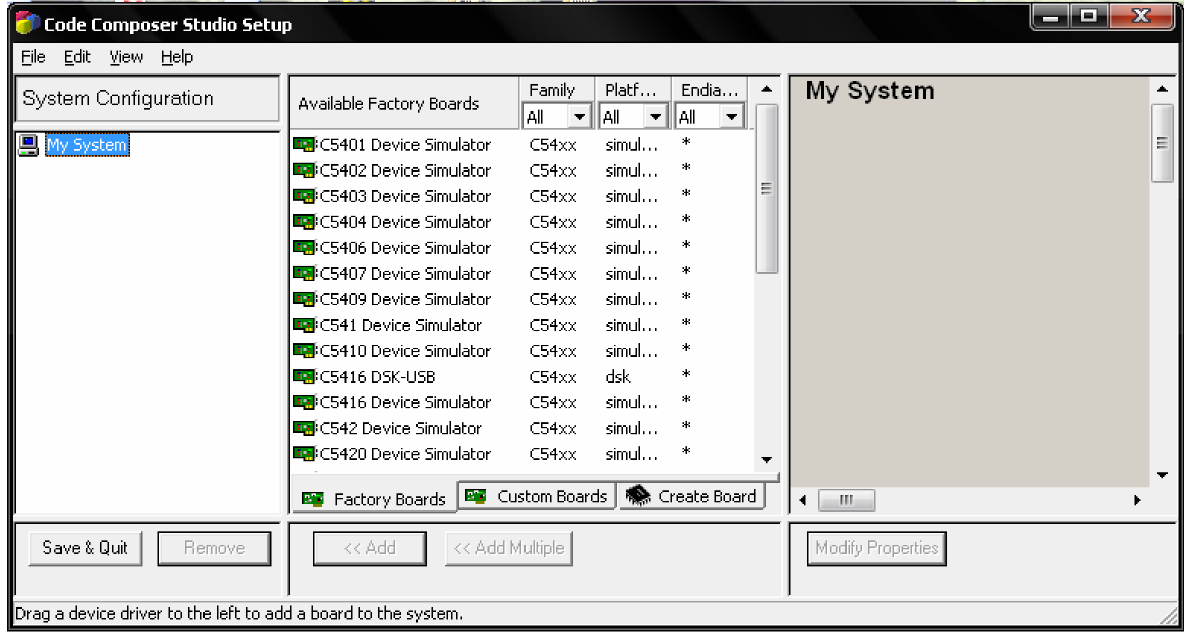


Рис. 2.1 утилита «Setup CCStudio v3.1»

3) из списка выбрать плат: «C5510 Device Simulator» и нажать кнопку: «<<Add» (рис. 2.2). Этот отладочный модуль является виртуальным симулятором отладочной платы DSK5510, и именно он будет установлен в систему для работы с ИСР CCS;

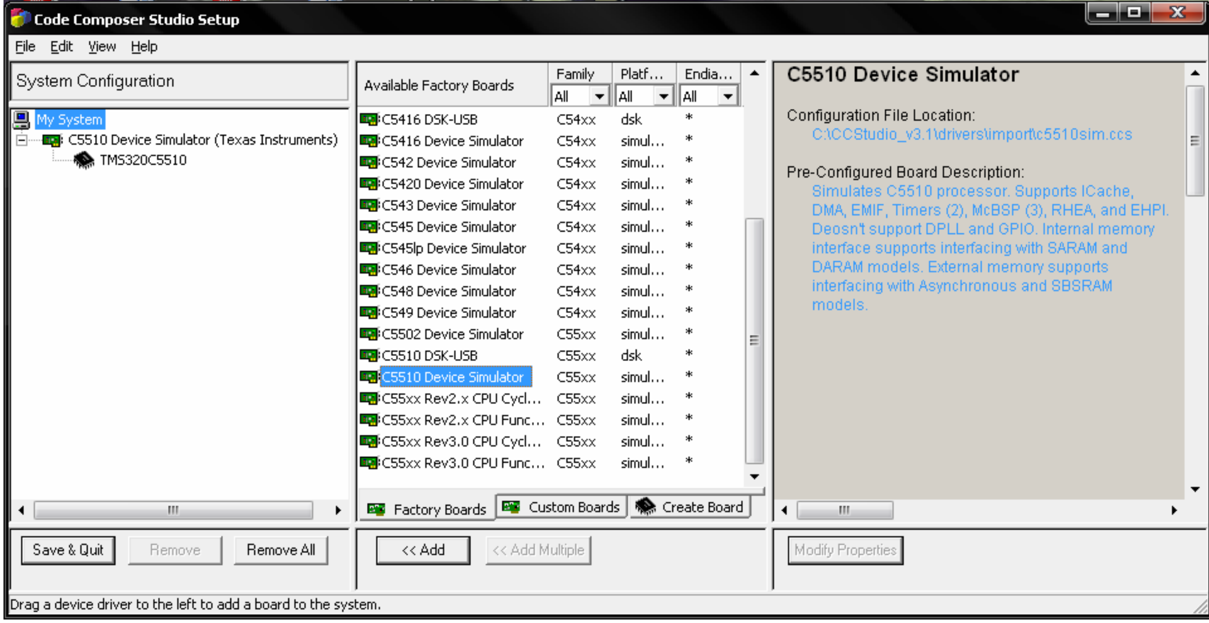


Рис. 2.2 Добавление DSK5510 в систему

4) После добавление нажать кнопку «Save & Quit» для сохранения настроек системы и выхода из утилиты. В появившемся диалоге нажать кнопку «Да» (рис. 2.3);

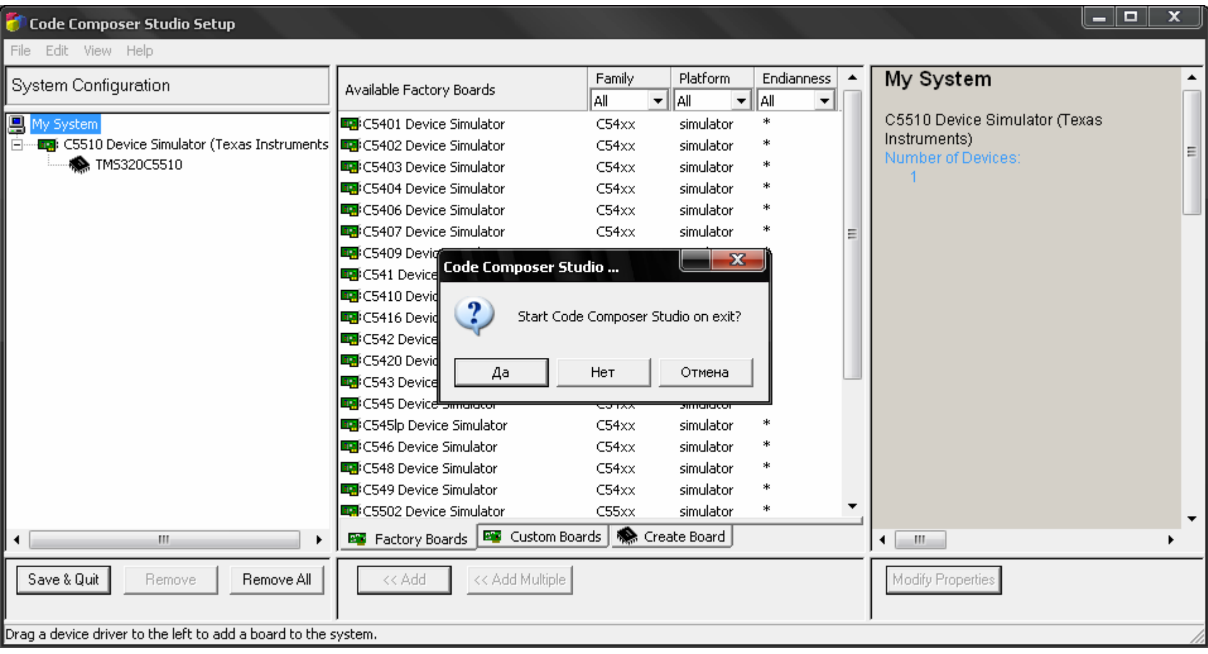


Рис. 2.3 Запрос на запуск ИСР CCS

5) Автоматически запуститься среда разработки (рис. 2.4):

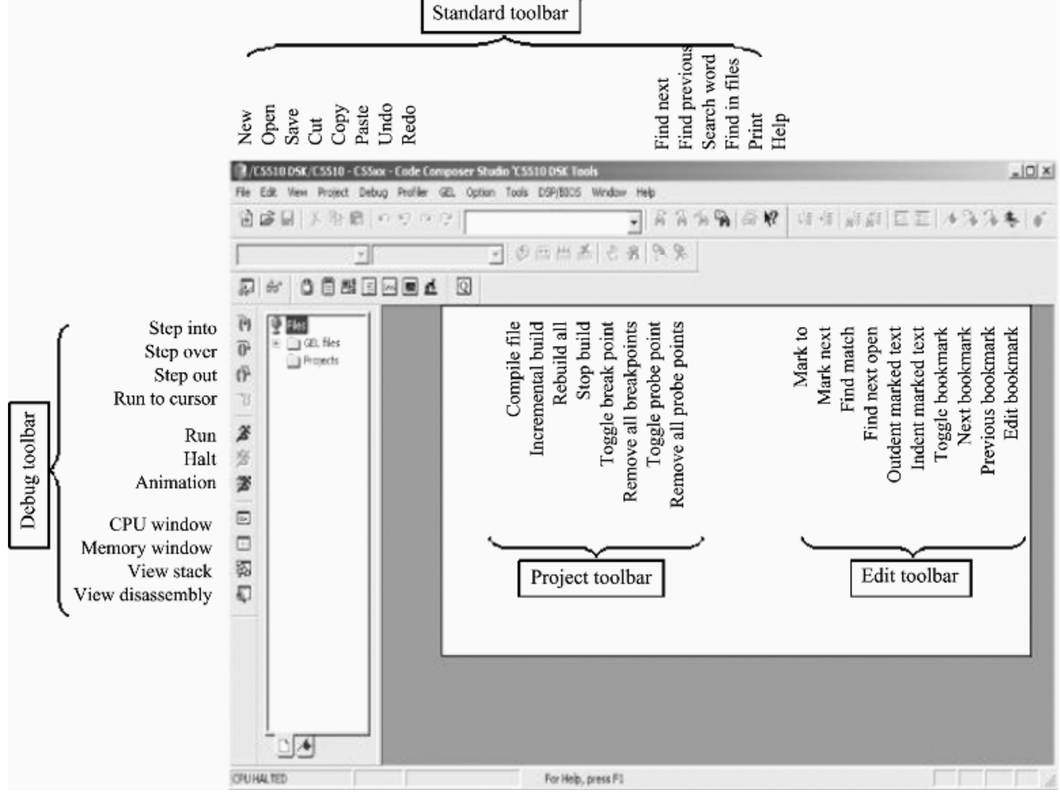


Рис. 2.4 Внешний вид ИСР CCS

3. Особенности проектирования в ИСР Code Composer Studio

Разработка Программного Обеспечения и Концепция COFF

Для стандартизации процесса разработки программного обеспечения Texas Instruments использует «Общий Формат Объектных Файлов» (Common Object File Format (COFF)). COFF наиболее эффективен в задачах, когда разработка программного обеспечения разделена между несколькими программистами.

Каждый файл, содержащий программный код, может быть написан независимо и называется модулем, включая спецификацию всех ресурсов, необходимых для правильного выполнения модуля. Модули могут быть написаны в Code Composer Studio или текстовом редакторе, который может сохранять файлы в ASCII-формате. Предполагаемые расширения исходных файлов на языке ассемблера – .asm, и .c для языка Си.

Множество модулей объединяются в завершенную программу с помощью компоновщика. Компоновщик эффективно распределяет ресурсы доступные в устройстве для каждого модуля проекта. Компоновщик пользуется командным файлом .cmd, для определения ресурсов памяти и правил расположения в них различных секций, входящих в состав модуля. Выходом процесса компоновки становится скомпонованный объектный файл .out, который может быть выполнен на DSP, и может быть получен .map файл, который показывает, куда были размещены все секции проекта. Схематически этот процесс отражен на рисунке 3.1.

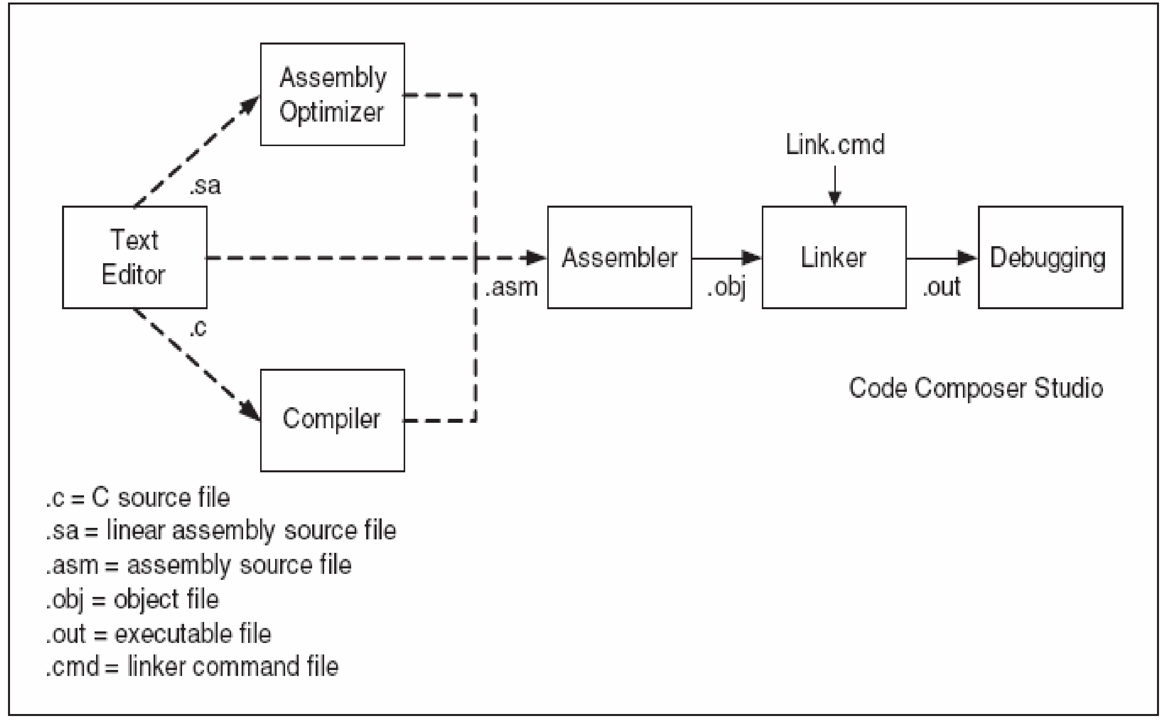


Рис. 3.1 Этапы сборки проекта

Концепция COFF обеспечивает модульную разработку программного обеспечения, независимую от аппаратной базы. Отдельно взятый ассемблерный файл может быть написан для выполнения определенной задачи и может быть скомпонован с несколькими другими задачами для получения более сложных систем.

Код разрабатывается аппаратно-независимым, что позволяет программисту модуля не думать о распределении памяти, так как эта работа будет сделана на этапе компоновки проекта. При изменении любого из модулей производится новая сборка проекта, и вопросы распределения памяти решаются заново, исключая вероятность появления конфликтов.

Проекты

Code Composer работает с проектами, которые создаются для решения задач ЦОС на базе ЦСП. Проект содержит в себе всю необходимую информацию для того, чтобы создать запускаемый файл. Например, он описывает такие вещи как: исходные файлы, заголовочные файлы, карту памяти конечного устройства и опции сборки программы. Информация проекта хранится в файле с расширением *.PJT, который создается и обслуживается средой Code Composer (рис. 3.2).

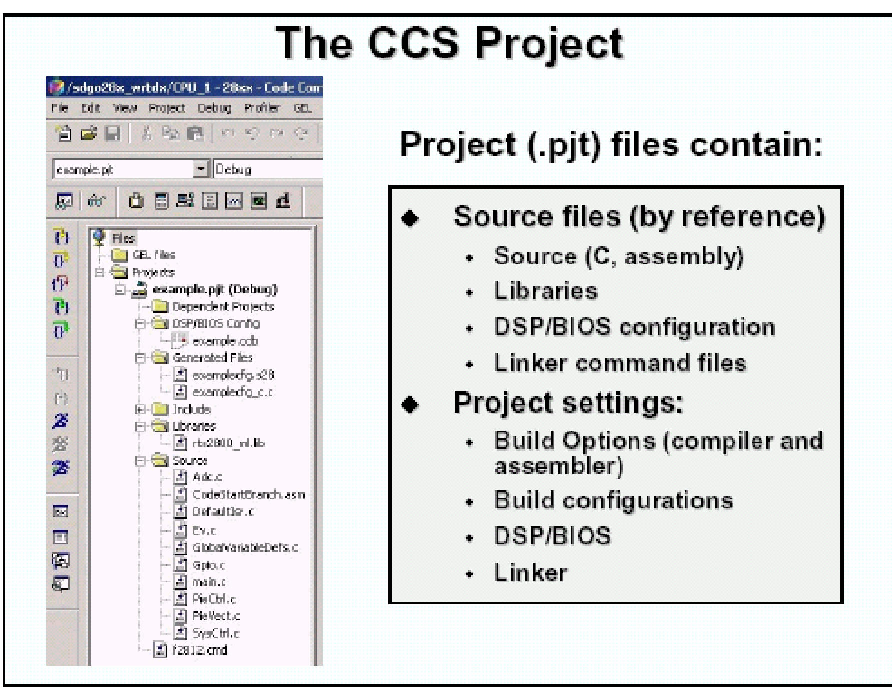


Рис. 3.2 Дерево проекта в ИСР CCS

Опции сборки

Опции проекта указывают компилятору, ассемблеру и компоновщику, как собрать код в соответствии с требованиями системы. Когда создается новый проект, Code Composer формирует два набора опций, называе-

мых конфигурациями: одна называется Debug – Отладочная, а вторая Release – Конечная (Оптимизированная).

Чтобы облегчить процесс настройки опций, ИСР CCS предлагает графический пользовательский интерфейс для различных настроек компилятора, открыть которые можно выбрав раздел Project → Build Options… после чего откроется окно как показано на рисунке 3.3.

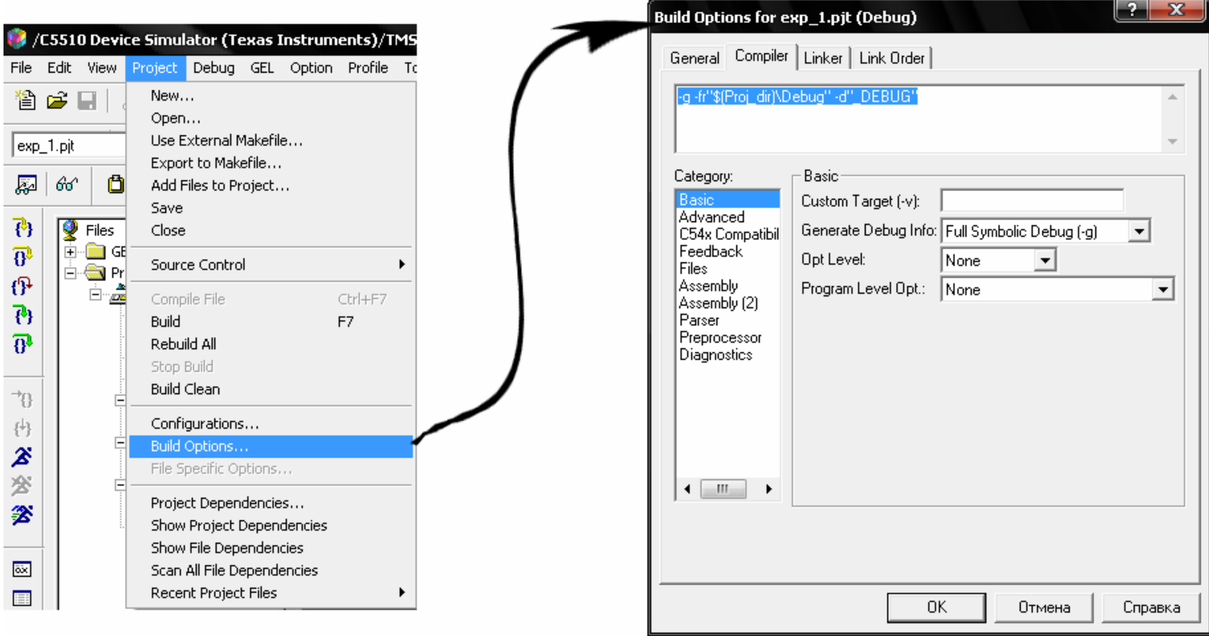


Рис. 3.3 Опции сборки проекта

Содержимое строки компилятора полностью соответствует тому, что выбрано через пользовательский интерфейс.

Существует ряд опций компоновщика, но следующие четыре являются основными:

1) «–o <filename>» – определяет имя выходного (пускового) файла;

2) «–m <filename>» – указывает компоновщику создавать файл с картой проекта (map-файл;

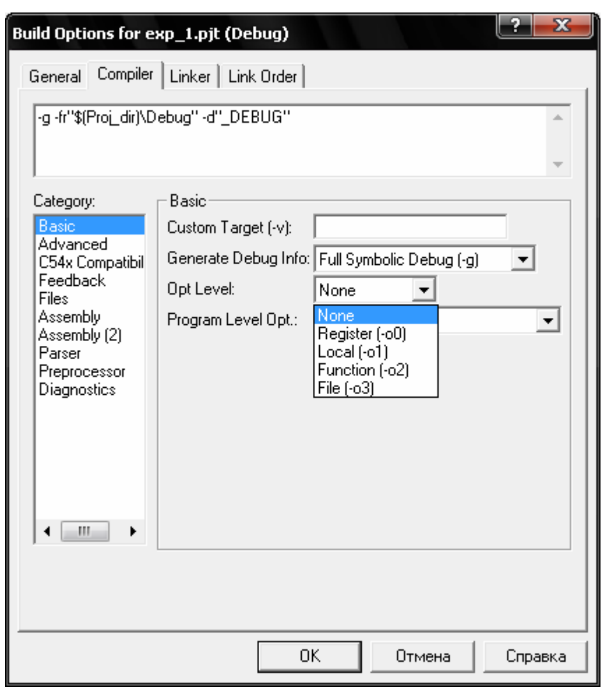
3) «–c» – указывает компилятору, что он должен автоматически инициализировать глобальные и статические переменны;

4) «–x» – указывает компоновщику перечитывать библиотеки. Без этой опции библиотеки прочитываются один раз, поэтому ссылки из одной библиотеки на другую могут не быть разрешены.

Среди ряда параметров сборки, особое внимание следует уделить выбору уровню оптимизации (Program Level Opt. рисунок 3.4). Есть четыре уровня оптимизации (-o0, -o1, -o2, -o3), которые управляют типом и степенью оптимизации:

1. Уровень 0 оптимизации выполняет упрощение контрольного потокового графа, распределяет переменные для регистров, исключает неиспользованный код, и упрощает выражения и утверждения.

2. Уровень 1 оптимизации выполняет все оптимизации Уровня 0, исключает неиспользованные определения, и исключает местные общие выражения.



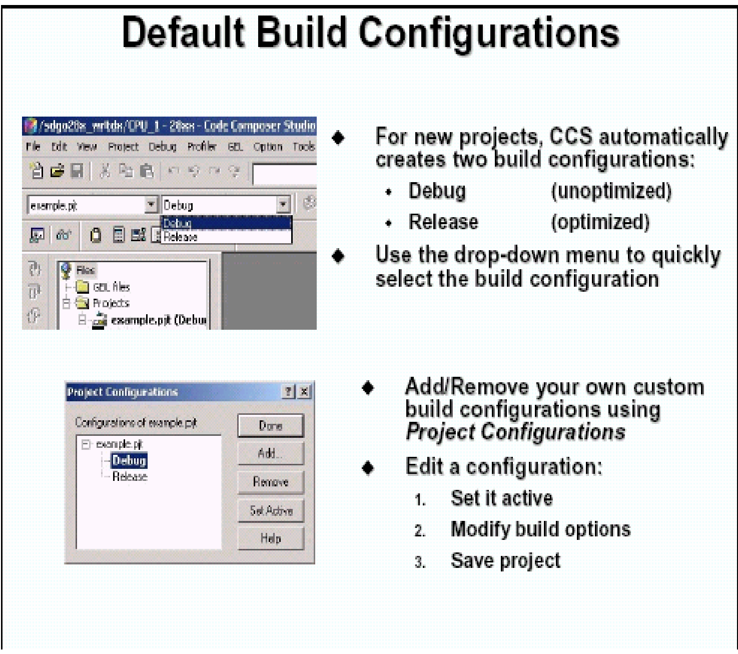


Рис. 3.4 Опции сборки и оптимизации

3. Уровень 2 оптимизации выполняет все оптимизации Уровня 1, плюс конвейерная обработка программного обеспечения, оптимизация циклов, и разворачивание циклов. Также исключает глобальные общие подвыражения и неиспользованные назначения.

4. Уровень 3 выполняет все оптимизации Уровня 2, исключает все функции, к которым никогда не обращаются, и упрощает функции возвращающие значения, которые никогда не используются.

Создание Командного Файла Компоновщика

Секции

Программа, написанная на языке Си, содержит одновременно и код, и разные типы данных (глобальные, локальные и т.д.). Texas Instruments и в принятом формате COFF эти различные части программы называются секциями – Sections. Разделение кода и данных на различные секции обеспечивает гибкость, так как позволяют размещать кодовые секции в ПЗУ, а переменные в ОЗУ. В таблице 3.1 показаны секции, создаваемые компилятором.

Таблица 3.1

Имена секций компилятора

Инициализированные секции		
Имя	Описание	Адрес размещения
.text	Код программы	Программа
.cinit	Инициализированные глобальные и статические переменные	Программа
.econst (.const)	Содержит данные (например, const int k = 3;)	Данные
.switch	Таблицы для оператора switch	Данные (программа для -mt)
.pinit	Таблицы глобальных конструкторов языка Си	Программа
Неинициализированные секции		
Имя	Описание	Адрес размещения
.ebss (.bbs)	Глобальные и статические переменные	Данные
.stack	Пространство стека	Младшие 64K данных
.esysmem (.sysmem)	Память для функций определения памяти far malloc	Данные

Примечание: в скобках указаны секции для модели памяти .small (рис. 3.5).

Секции программы на языке Си должны размещаться в разных областях памяти процессорной системы. Раздельные секции для кода, констант и переменных позволяют достичь большей производительности. В этом случае секции могут быть размещены точно в нужной области памяти процессорной системы. Секции размещают исходя из назначения (рис. 3.6), руководствуясь следующими правилами:

Программный Код (.text)

Программный код состоит из последовательности инструкций для обработки данных, инициализации системных настроек и так далее. Про-

граммный код должен быть определен во время сброса (Reset) или включения питания. Поэтому необходимо помещать программный код в энергонезависимую память такую как Flash или EPROM.

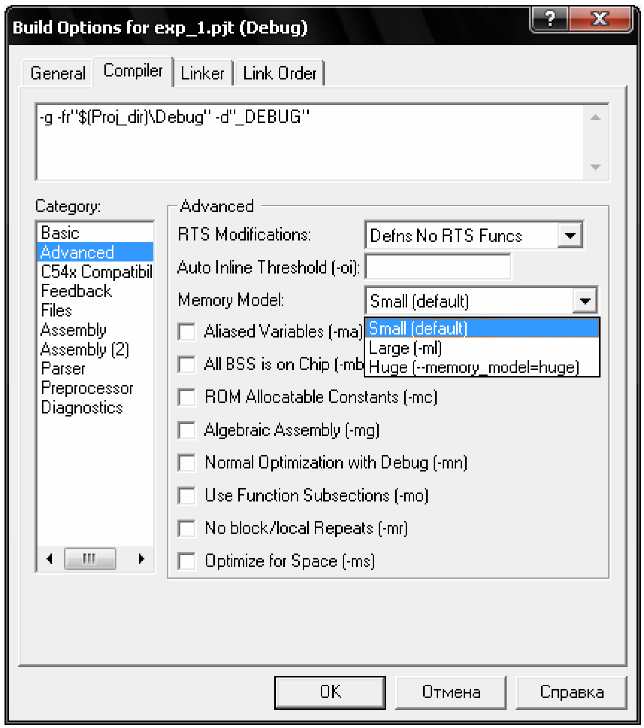


Рис. 3.5 Опции компилятора

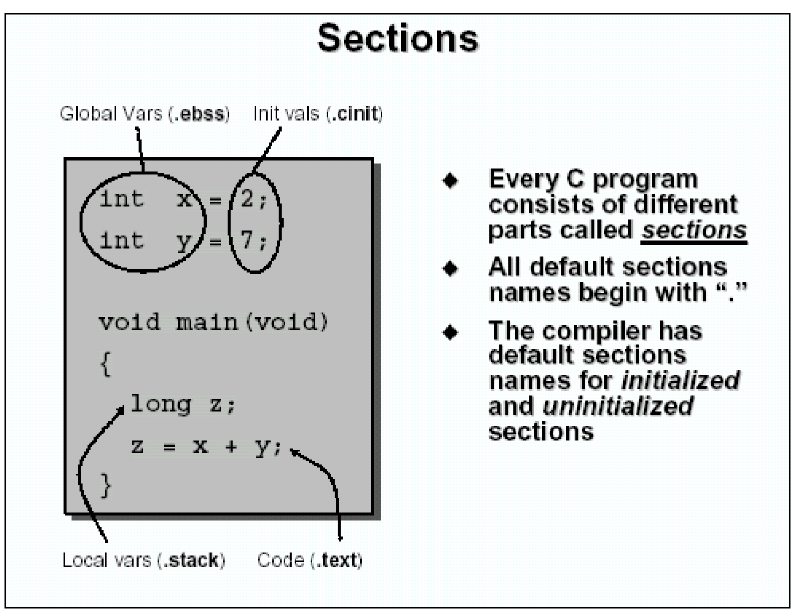


Рис. 3.6 Разделение кода программы по секциям памяти

Константы (.cinit – инициализированные данные)

Инициализированными называются те области памяти данных, которые должны быть определены во время сброса (Reset) или включения питания. Константы содержат начальные значения переменных или значения

по умолчанию. Константы подобно программному коду определяют в энергонезависимой памяти.

Переменные (.ebss – неинициализированные данные)

Неинициализированные области памяти содержат данные, которые могут изменяться в процессе выполнения программного кода. Эти данные, в отличие от программного кода и констант, должны находиться в ОЗУ. Эти области памяти могут модифицироваться и обновляться, как это происходит в языках высокого уровня при вычислении математических формул. Каждая переменная объявляется директивой, чтобы зарезервировать память, содержащую ее значение. В соответствии с их определением, им не присваивается какое-либо значение, а изменить их может только работающая программа.

Процесс компоновки кода состоит из трех шагов:

1) определение различных областей памяти (встроенная SARAM, Flash, внешняя память);

2) описание распределения секций по областям памяти;

3) запуск компоновщика в процессе сборки (build) или пересборки (rebuild).

Командный Файл Компоновщика (.cmd)

Компоновщик объединяет секции из всех входных файлов, определяя память каждой секции исходя из ее длины и положения определяемого командами MEMORY и SECTIONS в командном файле компоновщика.

Область команды MEMORY описывает конфигурацию памяти процессорной системы для компоновщика.

Формат описания следующий:

```
Name:     origin = 0x????, length = 0x????
```

Например, если разместить 128К Flash-памяти начиная с адреса 0x3D8000, то содержимое файла будет следующим:

```
MEMORY
{
        FLASH:      origin = 0x3D8000, length = 0x20000
}
```

Таким же образом определяются другие сегменты памяти. Если определить M0SARAM и M1SARAM, то это будет записано как:

```
MEMORY
{
        M0SARAM:    origin = 0x000000, length = 0x0400
        M1SARAM:    origin = 0x000400, length = 0x0400
}
```

Микроконтроллер имеет два типа памяти: память программ и память данных (Гарвардская архитектура). Поэтому описание каждого типа памя-

ти должно быть раздельно. Компоновщик использует следующий синтаксис для их описания:

Страница компоновщика	Определение Texas Instruments
Page 0	Program (память программ)
Page 1	Data (память данных)

Командный файл будет следующим:

```
MEMORY
{
          PAGE 0:     /* Память программ */
                FLASH:      origin = 0x3D8000, length = 0x20000
          PAGE 1:     /* Память данных */
                M0SARAM:    origin = 0x000000, length = 0x0400
                M1SARAM:    origin = 0x000400, length = 0x0400
}
```

Директива SECTIONS определяет, как будут распределены секции в описанной памяти. Следующий код используется для связи секций с памятью определенной в предыдущем примере:

```
SECTIONS
{
          .text:>     FLASH PAGE 0
          .ebss:>     M0SARAM     PAGE 1
          .cinit:>    FLASH PAGE 0
          .stack:>    M1SARAM     PAGE 1
}
```

Компоновщик будет собирать все кодовые секции из всех файлов проекта вместе. Также он поступает с остальными секциями.

Начиная с первой секции, компоновщик поместит их в указанный сегмент памяти.

```
MEMORY
{
          PAGE 0:     /* Память программ */
                FLASH:      origin = 0x3D8000, length = 0x20000
          PAGE 1:     /* Память данных */
                M0SARAM:    origin = 0x000000, length = 0x0400
                M1SARAM:    origin = 0x000400, length = 0x0400
}
SECTIONS
{
          .text:>     FLASH PAGE 0
          .ebss:>     M0SARAM     PAGE 1
          .cinit:>    FLASH PAGE 0
          .stack:>    M1SARAM     PAGE 1
}
```

Карта памяти для TMS320VC5510

Все 16М байты памяти адресуемы как пространство программ или данных (рис. 3.7). Когда ЦП использует пространство программ для чтения программного кода из памяти, он применяет адреса в 24 бита для обращения к байтам. Когда программа получает доступ к пространству данных, она использует адреса в 23 бита для обращения к 16 - разрядным словам. В обоих случаях шины адреса переносят значения в 24 бита, но в течение доступа к пространству данных, младший бит на шине адреса приравнивается к 0.

Пространство данных делится на 128 основных страниц данных (с 0 по 127). Каждая основная страница данных имеет 64К адресов. Инструкция, ссылающаяся на основную страницу данных, объединяет 7- разрядную страницу данных с 16 - разрядным офсетом.

На странице данных 0 первые 96 адресов (00 0000h-00 005Fh) зарезервированы под регистры, отображенные в карте памяти (MMR). Существует соответствующий блок из 192-х адресов (00 0000h-00 00BFh) в пространстве программ. Не рекомендуется хранить (запоминать) программный код в пространстве памяти.

Для того, чтобы посмотреть, как адреса распределяются между внутренней памятью и внешней памятью, и, чтобы получить информацию о внутренней памяти, обратитесь к технической документации вашего C55x DSP в частности SPRU371D (перевод).

	Адреса пространства данных (Шестнадцатир. диапазон)	Память программ/данных	Адреса пространства программ (Шестнадцатиричный диапазон)
Главная стр. данных 0	MMRs 00 0000–00 005F		00 0000–00 00BF
	00 0060–00 FFFF		00 00C0–01 FFFF
Главная стр. данных 1	01 0000–01 FFFF		02 0000–03 FFFF
Главная стр. данных 2	02 0000–02 FFFF		04 0000–05 FFFF
. . .	. . .		. . .
Главная стр. данных 127	7F 0000–7F FFFF		FE 0000–FF FFFF

Рис. 3.7 Карта памяти TMS320VC5510

Обращение к памяти процессора может осуществляться по адресам. Аппаратно доступ к блокам, описанным в таблице 3.2, может осуществлять-

ся параллельно. Т. е. два доступа за один цикл (два чтения, две записи или чтение и запись). DARAM - Dual-Access RAM (Память с двойным доступом).

Таблица 3.2

Пространство памяти DARAM

BYTE ADDRESS RANGE	MEMORY BLOCK
000000h – 001FFFh	DARAM 0†
002000h – 003FFFh	DARAM 1
004000h – 005FFFh	DARAM 2
006000h – 007FFFh	DARAM 3
008000h – 009FFFh	DARAM 4
00A000h – 00BFFFh	DARAM 5
00C000h – 00DFFFh	DARAM 6
00E000h – 00FFFFh	DARAM 7

† First 192 bytes are reserved for Memory-Mapped Registers (MMRs).

Внутри кристалла также имеются области памяти с одиночным доступом (SARAM). В таблице 3.3 обозначено деление этих областей по адресам.

Таблица 3.3

Пространство памяти SARAM

BYTE ADDRESS RANGE	MEMORY BLOCK	BYTE ADDRESS RANGE	MEMORY BLOCK
010000h – 011FFFh	SARAM 0	030000h – 031FFFh	SARAM 16
012000h – 013FFFh	SARAM 1	032000h – 033FFFh	SARAM 17
014000h – 015FFFh	SARAM 2	034000h – 035FFFh	SARAM 18
016000h – 017FFFh	SARAM 3	036000h – 037FFFh	SARAM 19
018000h – 019FFFh	SARAM 4	038000h – 039FFFh	SARAM 20
01A000h – 01BFFFh	SARAM 5	03A000h – 03BFFFh	SARAM 21
01C000h – 01DFFFh	SARAM 6	03C000h – 03DFFFh	SARAM 22
01E000h – 01FFFFh	SARAM 7	03E000h – 03FFFFh	SARAM 23
020000h – 021FFFh	SARAM 8	040000h – 041FFFh	SARAM 24
022000h – 023FFFh	SARAM 9	042000h – 043FFFh	SARAM 25
024000h – 025FFFh	SARAM 10	044000h – 045FFFh	SARAM 26
026000h – 027FFFh	SARAM 11	046000h – 047FFFh	SARAM 27
028000h – 029FFFh	SARAM 12	048000h – 049FFFh	SARAM 28
02A000h – 02BFFFh	SARAM 13	04A000h – 04BFFFh	SARAM 29
02C000h – 02DFFFh	SARAM 14	04C000h – 04DFFFh	SARAM 30
02E000h – 02FFFFh	SARAM 15	04E000h – 04FFFFh	SARAM 31

В TMS320VC5510 есть вектор прерываний, который обслуживает все внутренние и внешние прерывания. Пространство памяти отведенное вектору прерывания представлено в таблице 3.4.

Таблица 3.4

Пространство памяти SARAM

BYTE ADDRESS RANGE	DESCRIPTION
FF8000h – FF8FFFh	Bootloader
FF9000h – FFF9FFh	Reserved
FFFA00h – FFFBFFh	Sine look-up table
FFFC00h – FFFEFFh	Factory Test Code
FFFF00h – FFFFFBh	Vector Table
FFFFFCh – FFFFFFh	ID Code

4. Реализация проекта в ИСР Code Composer Studio

Создание и запуск проекта

Создаваемый проект будет реализовывать копирование статически заданного массива констант в буфер. Массив констант это заданные отсчеты для одного периода синусоиды. Данные в буфер будут скопированы в противофазе.

Процедура создания проекта:

1. Запустить ИСР Code Composer Studio в режиме Simulator.

2. Выбирать раздел Project → New, в появившемся диалоговом окне ввести имя будущего проекта и место где он будет сохранен, рекомендуется сохранять проект в папке MyProjects, в частности: “laba_1” и нажать кнопку «Готово» (рис. 4.1):

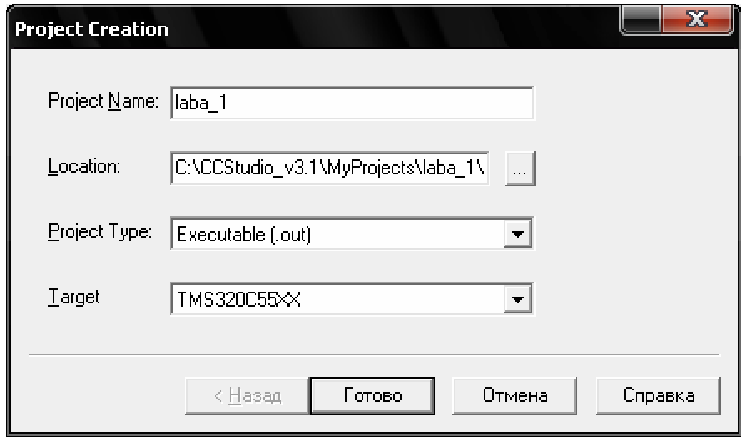


Рис. 4.1 Окно создания нового проекта

3. Выбрать раздел File → New → Source File или начать CNTR+N. В появившемся редакторе ввести код из листинга 4.1, который объявляет массив размером BUF_SIZE, перед началом цикла выводит соответствующее сообщение. В цикле происходит копирование в массив данных, объявленных в файле «arr.h», и вывод на экран. По завершении цикла будет выведено соответствующее сообщение.

Листинг 4.1.

```
#include <stdio.h>
#include "arr.h"

short outBuffer[BUF_SIZE];

void main()
{
          short i;
          printf("BEGIN\n");
                for (i=0; i<BUF_SIZE;i++)
                      {
                      outBuffer [i] = 0 - sineTable[i]; //
<- Set breakpoint
                      printf("%d, ",outBuffer [i]);
                      }
          printf("\nEND\n");
}
```

После вставки кода в редакторе выбрать раздел File → Save As и сохранить файл с именем main.c в папку, где находится проект, в частности в папку laba_1.

4. Создать и сохранить файл arr.h из листинга 4.2 (аналогично п. 3) в котором объявлен массив констант, представляющих собой отсчеты синусоиды.

Листинг 4.2.

```
#define BUF_SIZE 40
const short sineTable[BUF_SIZE]=
{0x0000, 0x000f, 0x001e, 0x002d, 0x003a, 0x0046, 0x0050,
0x0059,0x005f, 0x0062, 0x0063, 0x0062, 0x005f, 0x0059, 0x0050,
0x0046,0x003a, 0x002d, 0x001e, 0x000f, 0x0000, 0xfff1, 0xffe2,
0xffd3,0xffc6, 0xffba, 0xffb0, 0xffa7, 0xffa1, 0xff9e, 0xff9d,
0xff9e,0xffa1, 0xffa7, 0xffb0, 0xffba, 0xffc6, 0xffd3, 0xffe2,
0xfff1};
```

5. Создать и сохранить командный файл lnk.cmd листинг 4.3 (аналогично п. 3), все пояснения изложены в разделе: «Создание Командного Файла».

Листинг 4.3.

```
/* Specify the system memory map */
MEMORY
{
RAM  (RWIX) : o = 0x000100, l = 0x00feff /* Data memory */
RAM0 (RWIX) : o = 0x010000, l = 0x008000 /* Data memory */
RAM1 (RWIX) : o = 0x018000, l = 0x008000 /* Data memory */
RAM2 (RWIX) : o = 0x040100, l = 0x040000 /* Program memory */
ROM  (RIX)  : o = 0x020100, l = 0x020000 /* Program memory */
VECS (RIX) : o = 0xffff00, l = 0x000100 /* Reset vector */
}
/* Specify the sections allocation into memory */
SECTIONS
{
          vectors     > VECS /* Interrupt vector table */
          .text       > ROM /* Code */
          .switch     > RAM /* Switch table info */
          .const      > RAM /* Constant data */
          .cinit      > RAM2 /* Initialization tables */
          .data       > RAM /* Initialized data */
          .bss        > RAM /* Global & static vars */
          .stack      > RAM /* Primary system stack */
          .sysstack   > RAM /* Secondary system stack */
          expdata0    > RAM0 /* Global & static vars */
          «expdata1»  > RAM1 /* Global & static vars */
}
```

Примечание: Атрибуты, устанавливаемые для доступа к памяти в секции MEMORY: R – Пространство памяти открыто для чтения;
W – Пространство памяти открыто для записи;
X – Пространство памяти содержит рабочий код;
I – Пространство памяти может быть инициализировано.

6. Выбрать раздел Project → Add Files to Project… в появившемся окне обозревателя зайти в папку с проектом, установить просмотр для всех типов файлов, и выбрать файлы main.c, arr.h и lnk.cmd. Выбрать одновременно, несколько файлов, можно зажав клавишу CTRL. После этого снова выбрать раздел Project → Add Files to Project… и добавить к проекту библиотеку rts55.lib, она находится: C:\CCStudio_v3.1\C5500\cgtools\lib\. По окончании всех действий Code Composer Studio примет вид, как показано на рис. 4.2.

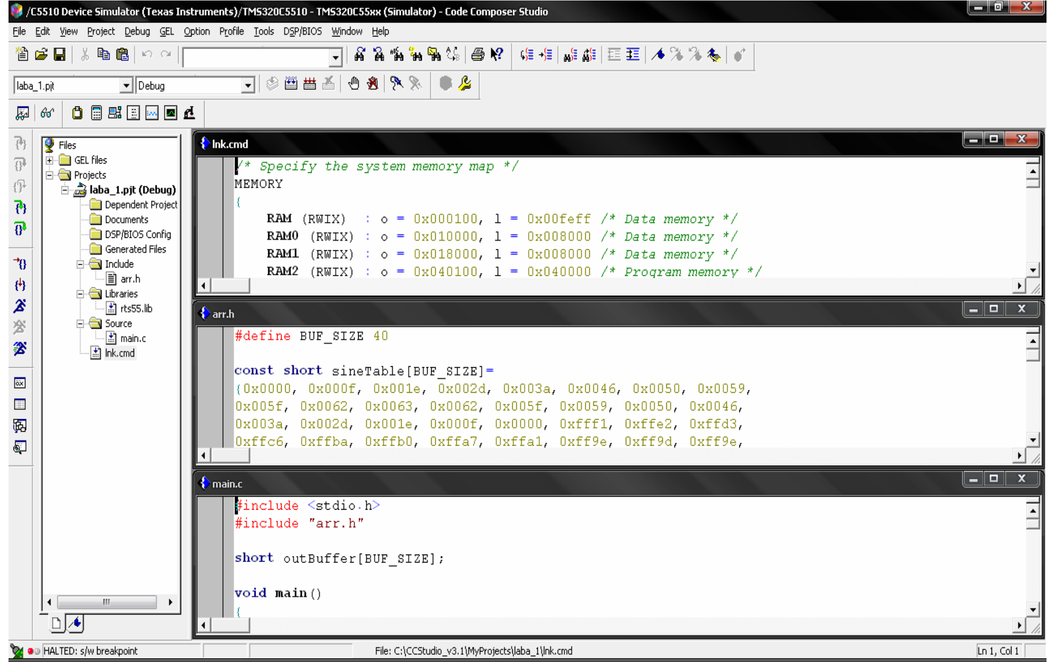


Рис. 4.2 Вид ИСР CCS с открытым проектом

7. Выбрать раздел Option → Customize и перейти на вкладку Program/Project Load и отметить пункт напротив Load Program After Build (рис. 4.3), эта функция необходима для загрузки выходного файла после компиляции в симулятор или плату DSK5510:

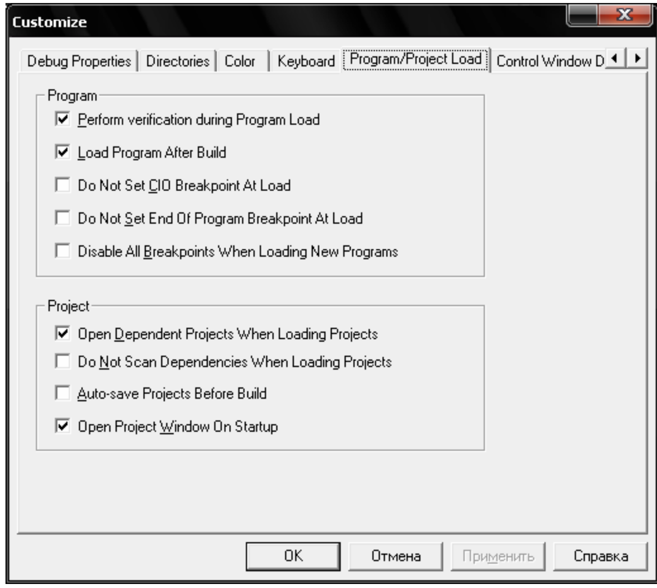


Рис. 4.3 Настройка загрузки программы/проекта

8. Выбрать раздел Project → Rebuild All или нажмите кнопку . После этого проект скомпилируется, и выходной файл будет загружен в симулятор. Для запуска программы после загрузки необходимо нажать «Alt+F5», что соответствует вызову функции Animate, которую можно также вызвать, нажав кнопку на панели инструментов ИСР CCS (рис. 4.4).

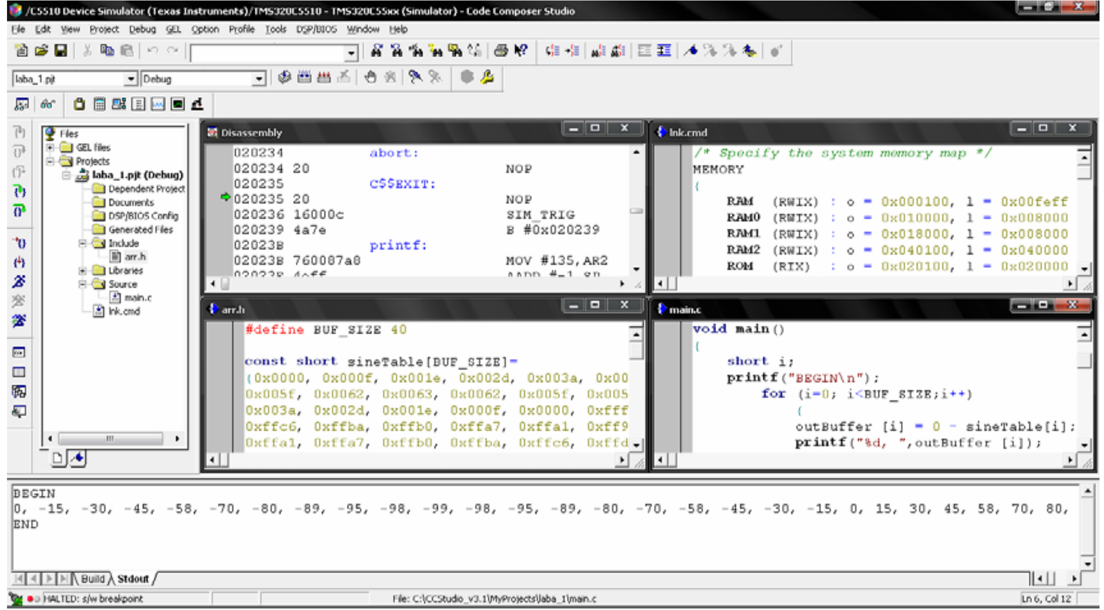


Рис. 4.4 Результаты выполнения программы в симуляторе

9. Сохранить все изменения в проекте и закрыть среду разработки.

Примечание: Очистить содержимое окна стандартного ввода/вывода (***stdout)*** *можно вызвав контекстное меню этого окна и выбрать в нем опцию* ***Clear****.*

5. Тестирование проекта в ИСР Code Composer Studio

Для облегчения процесса отладки CCS имеет возможность отображать отдельные области памяти в виде осциллограммы, изображения, спектра и оценки производительности.

Просмотр содержимого памяти

1) запустить среду разработки и открыть проект laba_1. выбрав раздел Project → Open…;

2) нажать кнопку на панели инструментов, после этого откроется окно Disassembly (его можно закрыть);

3) выбрать раздел View → Memory… В появившемся диалоговом окне внести изменения как показано на рисунке 5.1 и нажать кнопку OK;

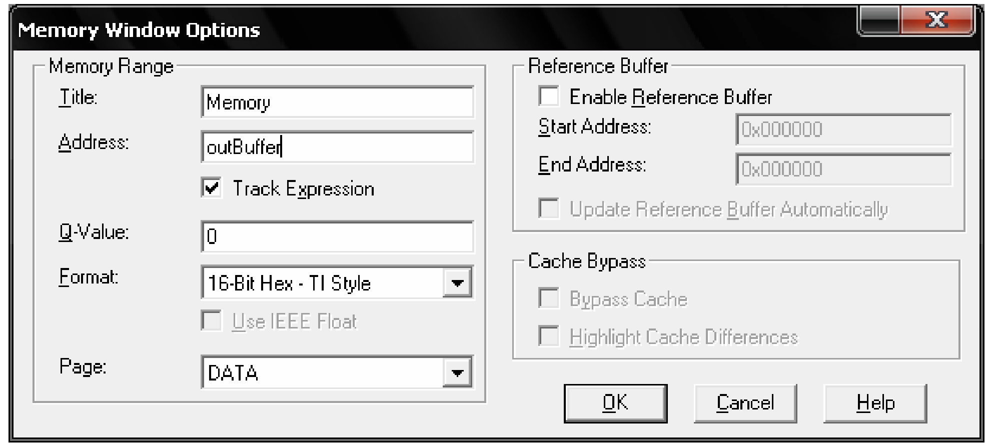


Рис. 5.1 Настройка параметров просмотра содержимого памяти

4) после этого откроется окно, в котором будет отображаться содержимое памяти. Если это первый запуск проекта, то содержимое памяти по указанному адресу будет содержать нули (рис. 5.2);

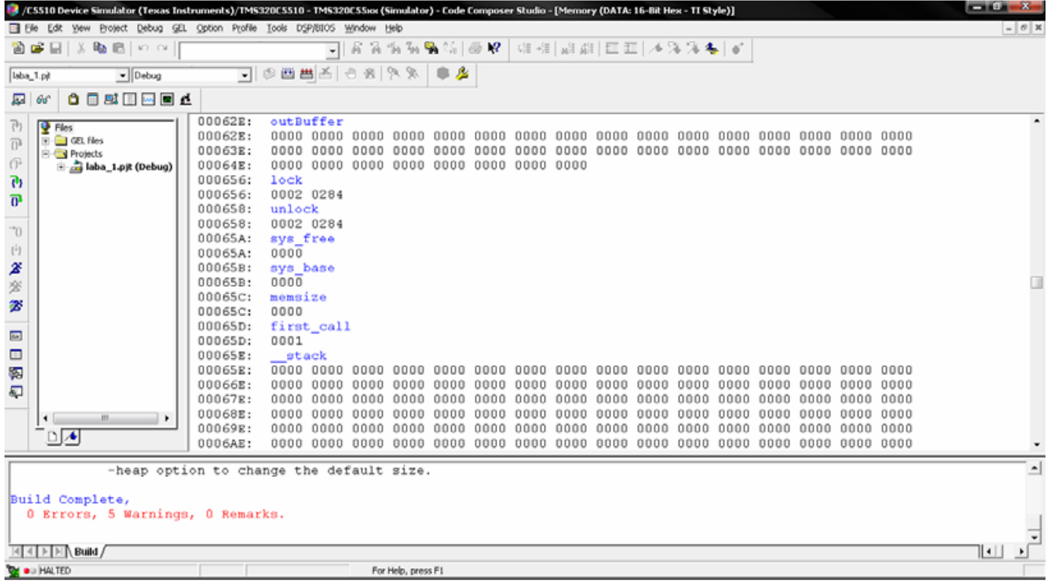


Рис. 5.2 Содержимое памяти по адресу outBuffer

5) теперь необходимо запустить режим Animate, нажав на кнопку , после этого опять откроется окно дизассемблера, закрыв его, можно будет увидеть, как изменилось содержание памяти (рис. 5.3);

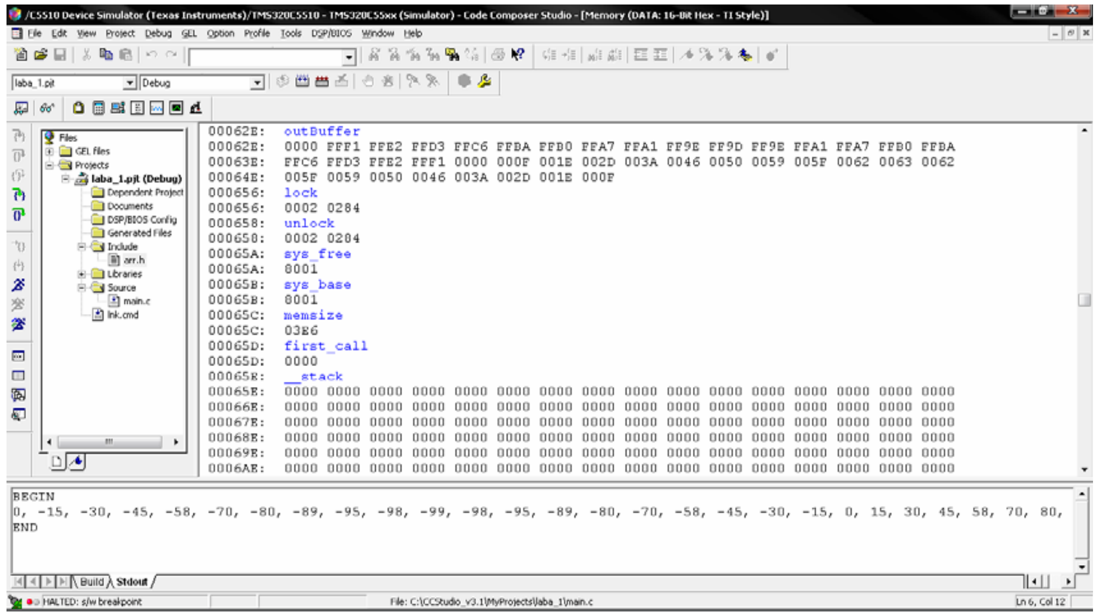


Рис. 5.3 Содержимое памяти по адресу outBuffer

Построение графиков

1) выбрать раздел View → Graph → Time/Frequency… в появившемся диалоговом окне внести изменения, как показано на рис. 5.4, и нажать кнопку OK:

a. Start Address – адрес начала построения графика, т. е. с этого места система начнет построение.

b. Acquisition Buffer Size – размер буфера.

c. Display Data Size – количество отображаемых значений.

d. DSP Data Type – тип данных, которые будут считаны.

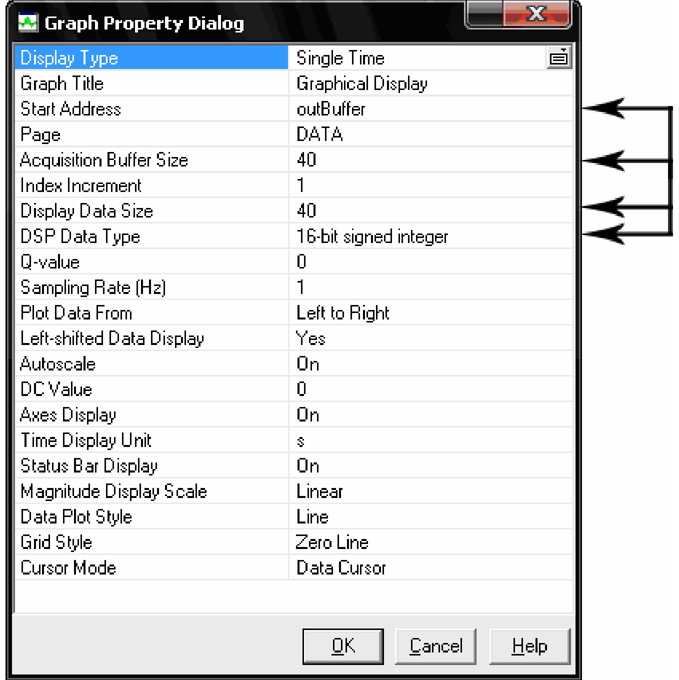


Рис. 5.4 Меню настройки окна визуализации

2) откроется окно Graphical Display (окно визуализации) (рис. 5.5), располагается оно не совсем удобно для работы, поэтому правым щелчком мыши в области этого окна вызовите контекстное меню и выберите опцию Float In Main Window. Затем ИСР CCS примет вид как показано на рис. 5.6;

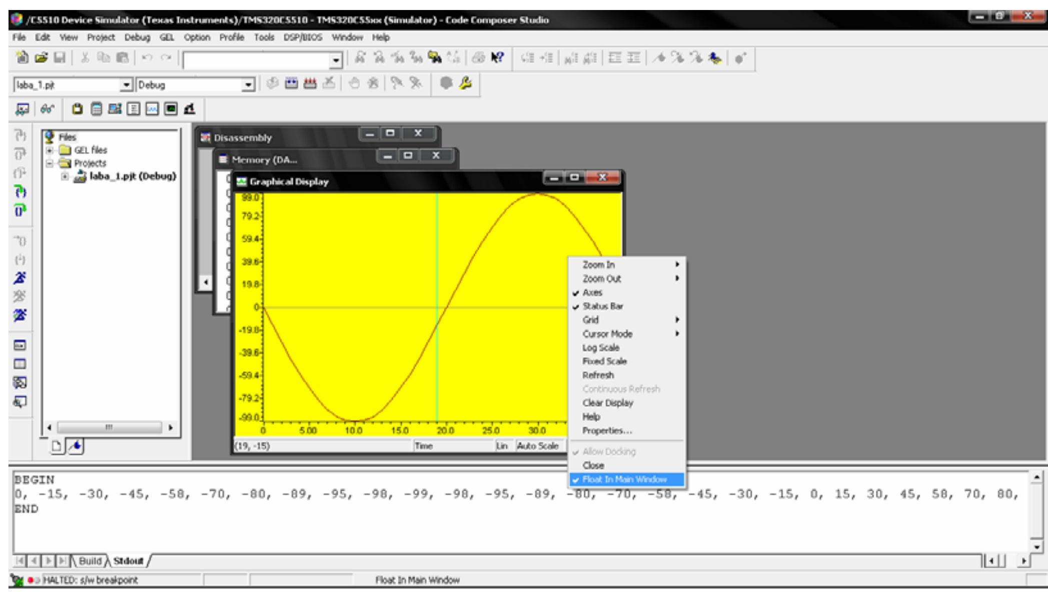


Рис. 5.5 Окно визуализации встроенное в главное окно

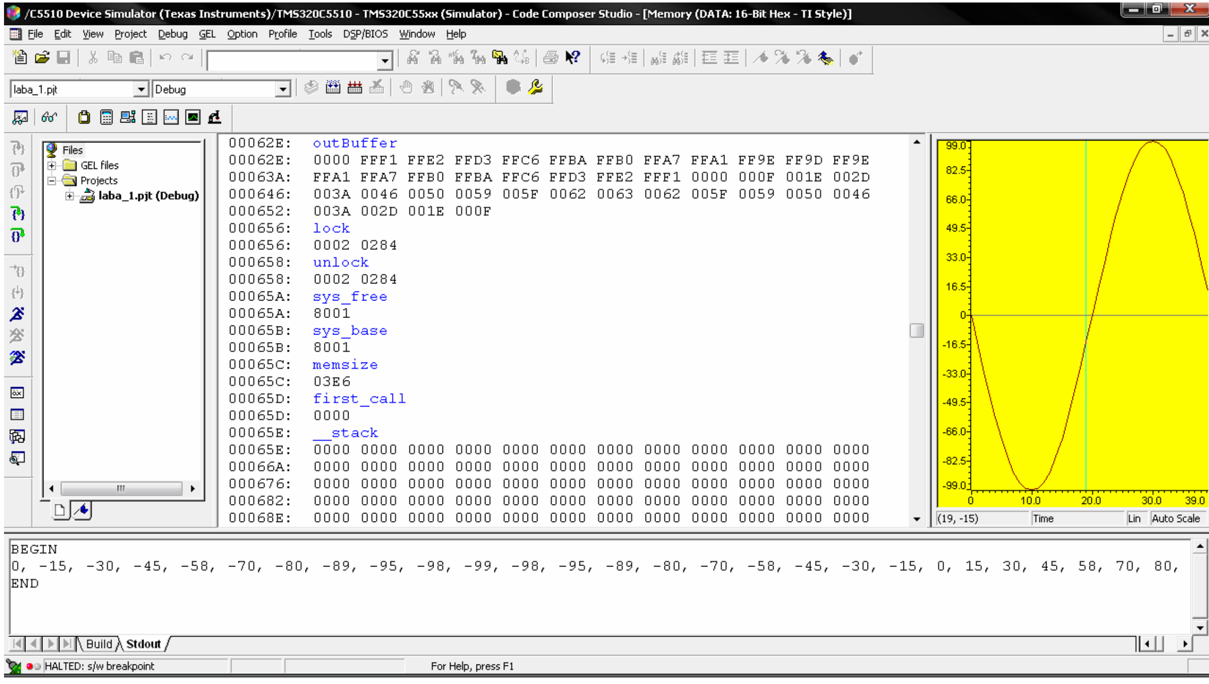


Рис. 5.6 Окно визуализации не встроенное в главное окно

3) изменения данных в окне визуализации происходит при остановке процесса выполнения программы. Для реализации данного положения нужно установить точку остановки в процессе выполнения программного кода: установить курсор в том месте программы, где необходимо прервать ее выполнение (рис. 5.7). Затем нажать кнопку на панели быстрых клавиш. Напротив выбранного места остановки появится красная точка (рис. 5.8);

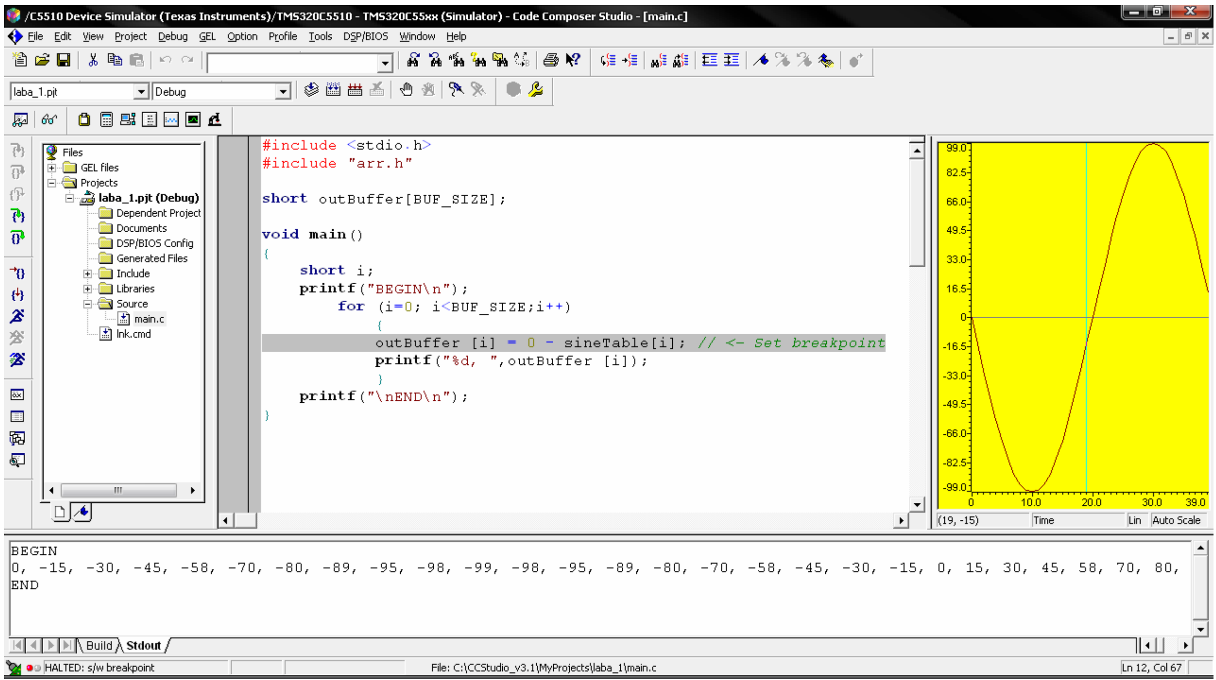


Рис. 5.7 Установка курсора в место прерывания программного кода

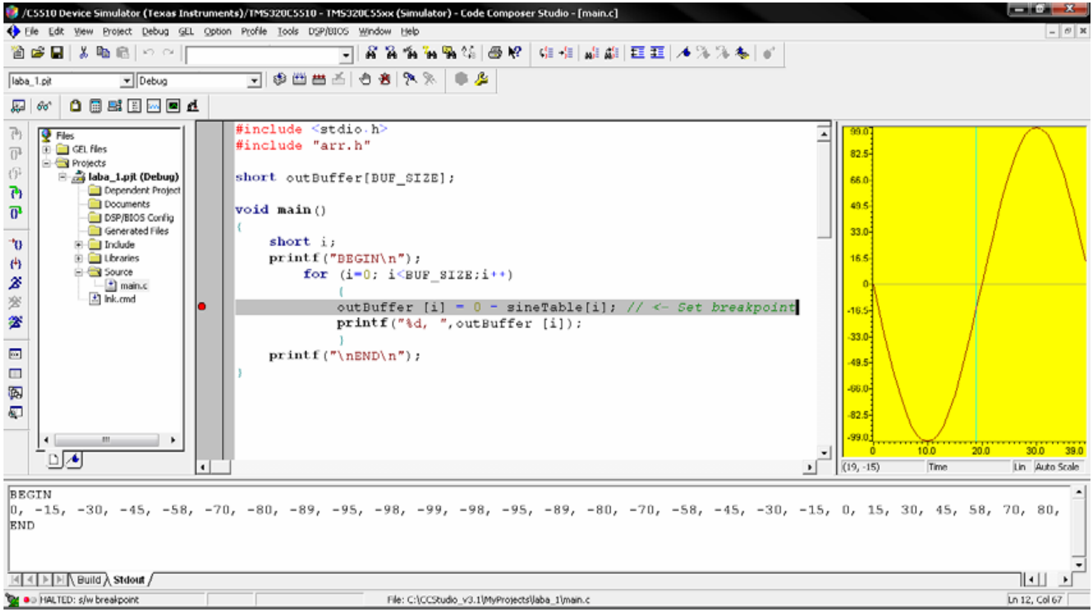


Рис. 5.8 Отображение прерывания программного кода

4) для того чтобы не проводить каждый раз компиляцию и перезагрузку программы, когда код не изменялся, необходимо установить программный указатель на начало программы:Debug → Restart;

5) выбрать раздел View → Graph → Time/Frequency… в появившемся диалоговом окне выбрать FFT Magnitude – отображение спектра, как показано на рис. 5.9, и нажать кнопку OK;

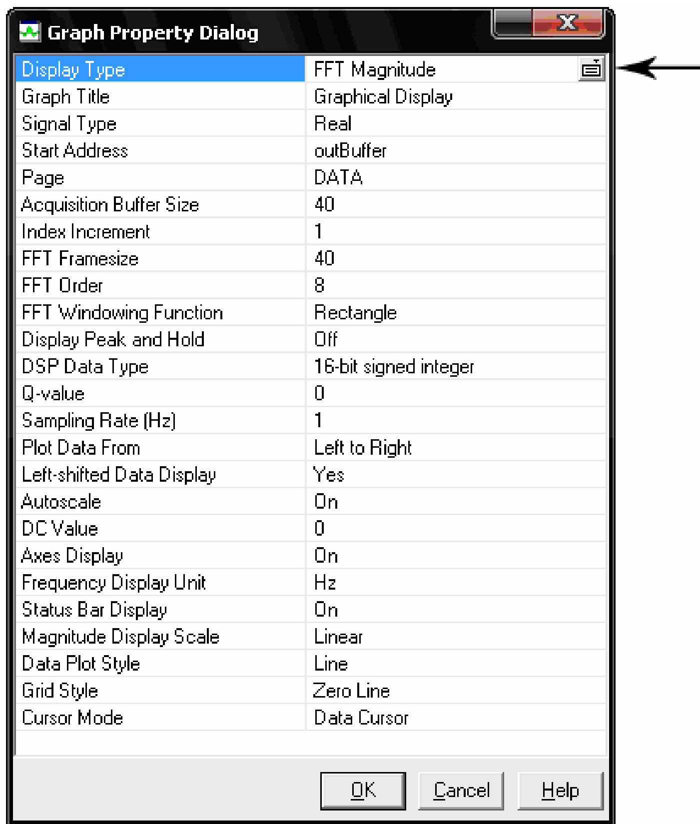


Рис. 5.9 Меню настройки окна визуализации

6) появившееся окно визуализации соответствует спектрограмме сигнала (рис. 5.10).

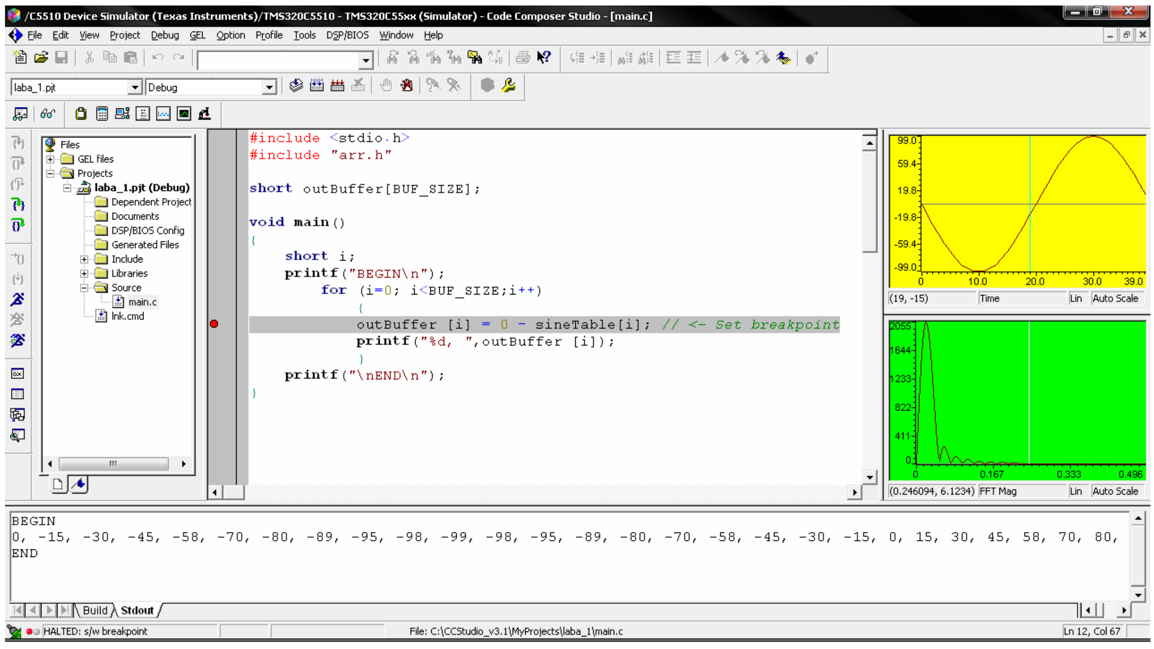


Рисунок 5.10 – Визуализация спектра.

Оценка производительности

Для оценки производительности в CCS реализован механизм профилирования (определения времени выполнения) отдельных участков кода и функций.

1. Выбрать раздел Profile → Setup в CCS появится окно настройки профилирования (рис. 5.11) и нажатием кнопки ⏱, это включит режим профилирования.

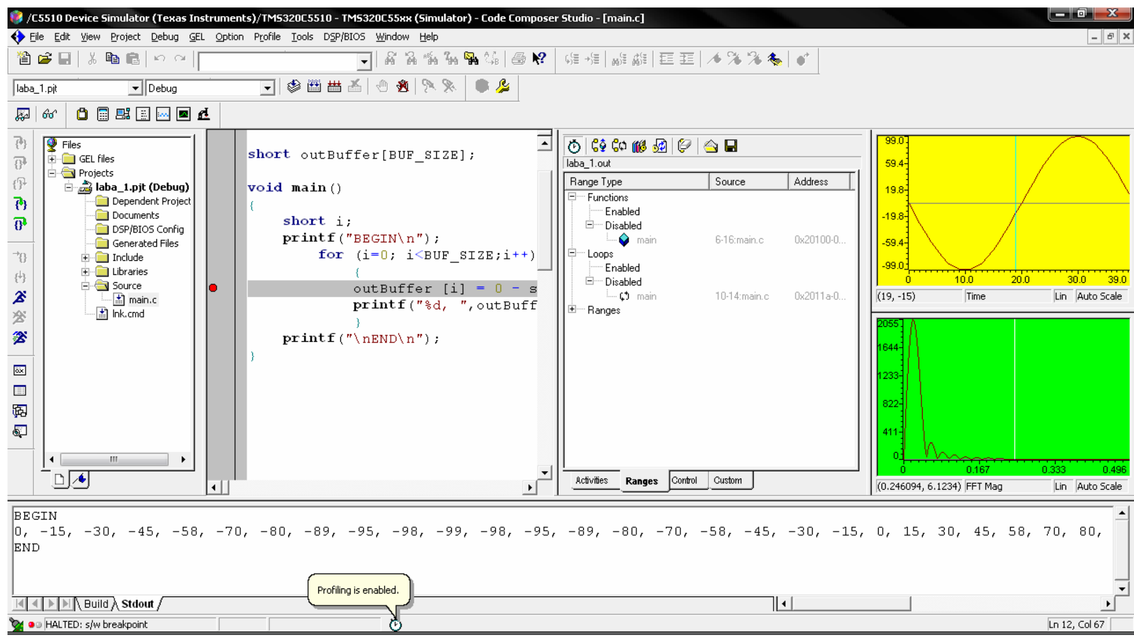


Рис. 5.11 Настройка режима профилирования

2. Перейти на вкладку Ranges (см. рис. 5.11) и перетащить мышкой функции, для которых будет производиться профилирование, из папки Disabled в папку Enabled (рис. 5.12).

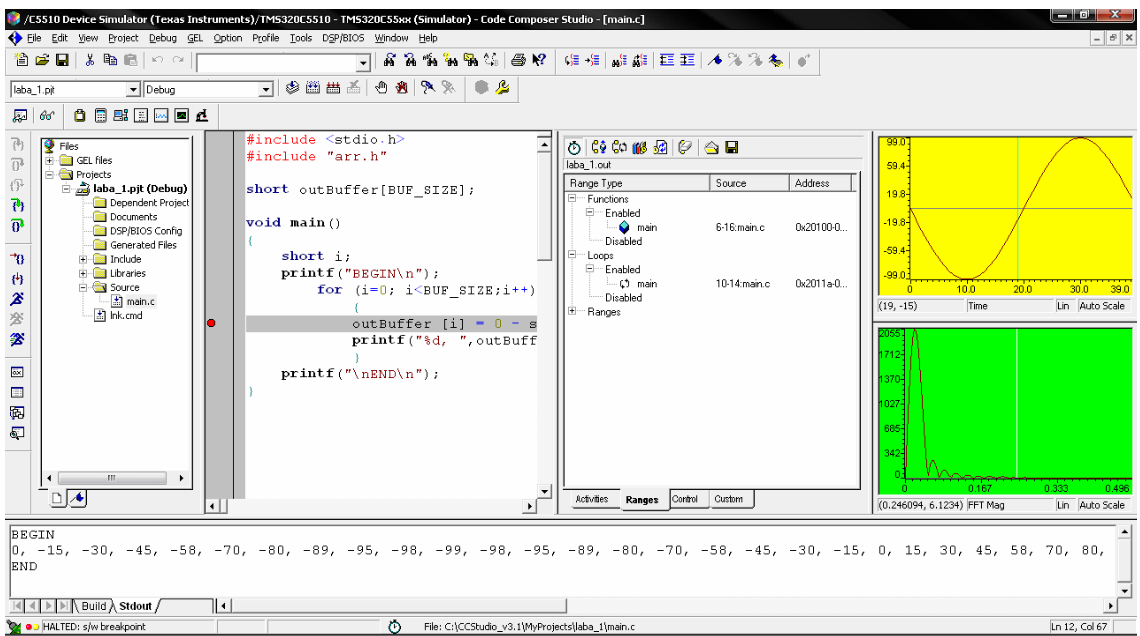


Рис. 5.12 Выбор функций и циклов для профилирования

3. На вкладке Custom выбрать размерность профилирования. В рассматриваемом примере профилирование будет производиться в тактах процессора. Для этого ставится галочка напротив позиции Cycles (рис. 5.13).

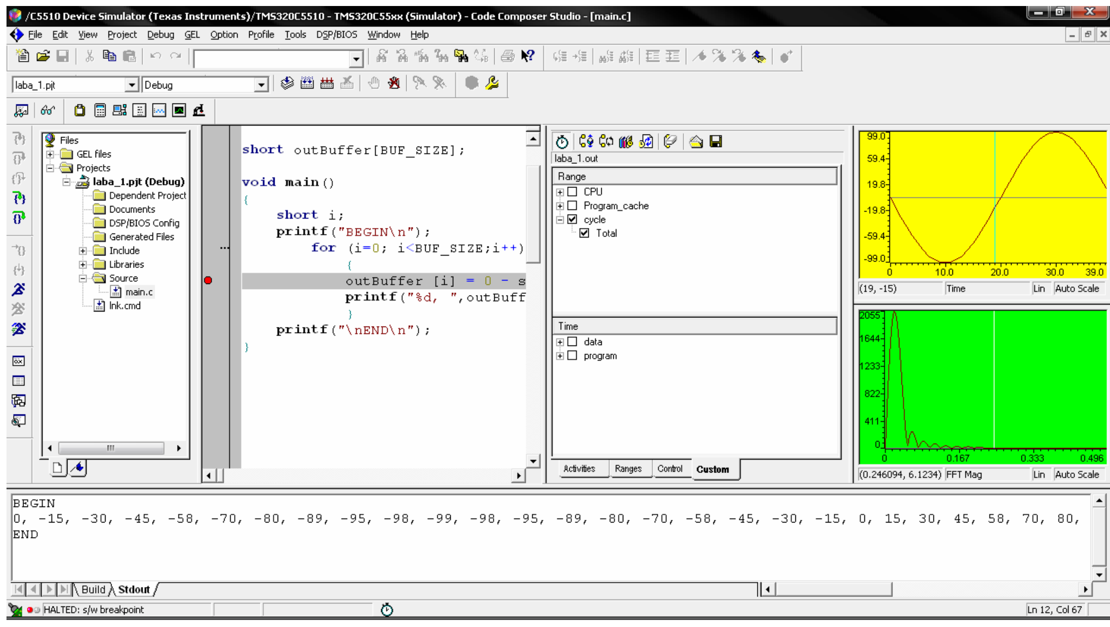


Рис. 5.13 Выбор размерности профилирования

4. Выбрать раздел Profile → View в ИСР CCS появится окно, в котором будет отображаться заданные параметры профилирования (рис. 5.14). Можно выбрать параметры, которые будут отображаться в окне профили-

рования. Для этого надо кликнуть правой кнопкой мыши в этом окне и из контекстного меню выбрать опцию Columns And Rows Setting.

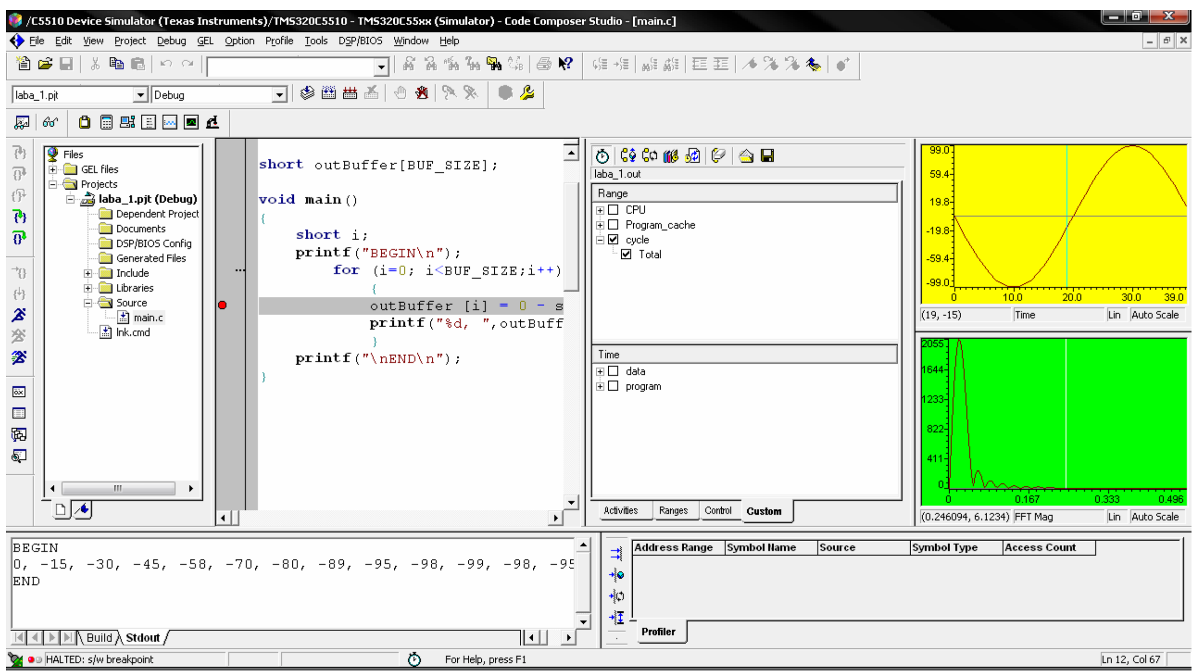


Рис. 5.14 Просмотр данных профилирования

5. Для получения данных необходимо заново перекомпилировать проект и запустить в режим Animate рис. 5.15.

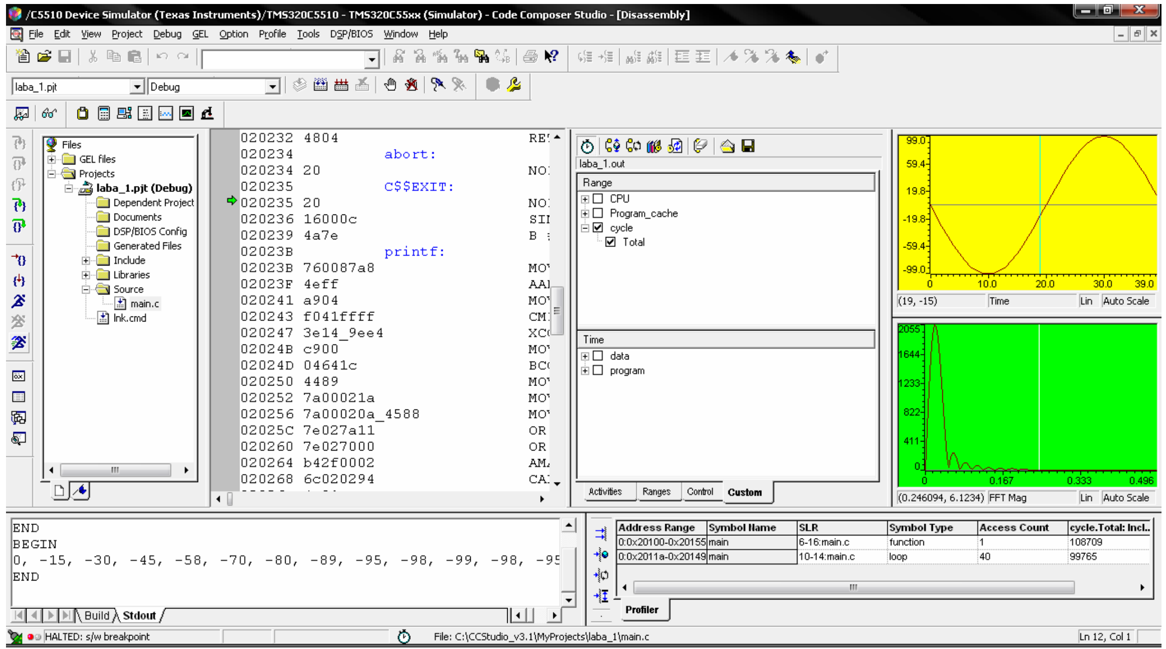


Рис. 5.15 Результаты профилирования

6. Выключается режим профилирования повторным нажатием на кнопку ⏱.

7. Закрыть ИСР CCS.

6. Аппаратная реализация проекта в ИСР Code Composer Studio

После того как программа прошла тестирование на симуляторе, необходимо протестировать на реальной плате, а именно DSK5510. Для этого необходимо выполнить ряд действий который позволит скомпилировать и загрузить проект в плату DSK5510.

1. Запускаем утилиту «Setup CCStudio v3.1». Выбираем плату «C5510 Device Simulator», которая установлена в нашей системе и нажимаем кнопку «Remove» или просто нажимаем кнопку «Remove All», как показано на рис. 6.1 и подтверждаем удаление (очистку системы).

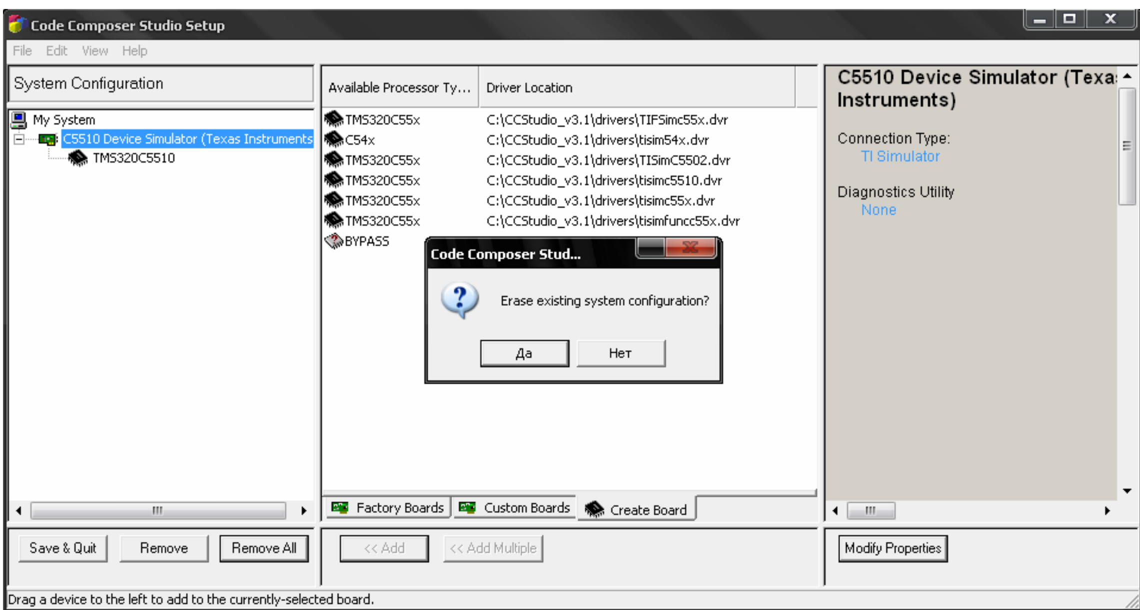


Рис. 6.1 Удаление плат установленных в системе

2. После очистки из списка предложенных плат выбираем «C5510 DSK-USB» и нажимаем кнопку «<<ADD», как показано на рис. 6.2 и 6.3.

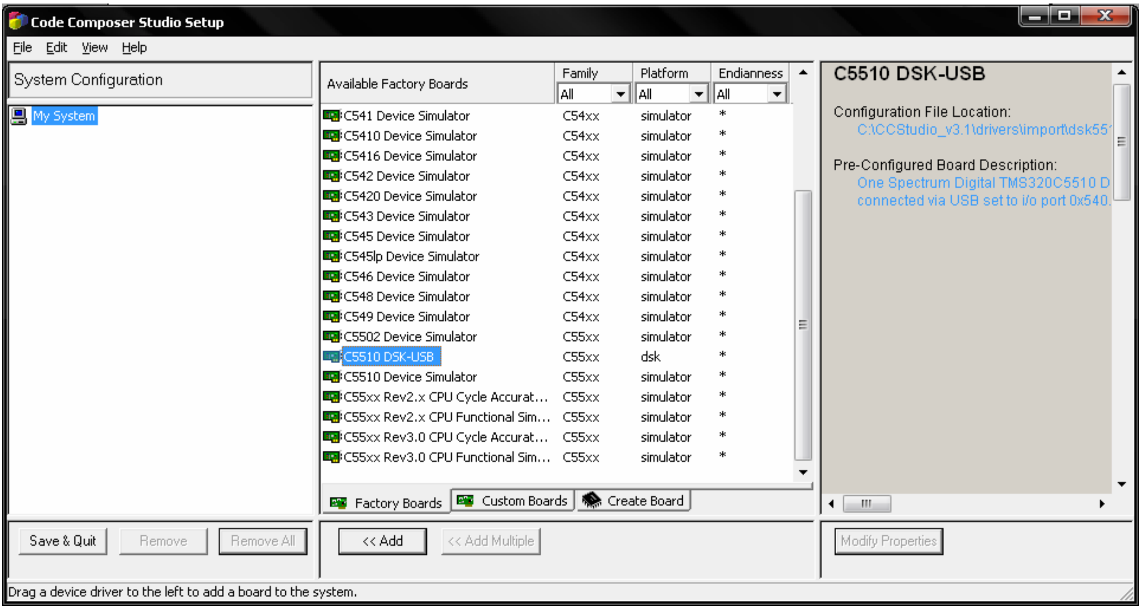


Рис. 6.2 Выбор реальной платы подключаемой через USB

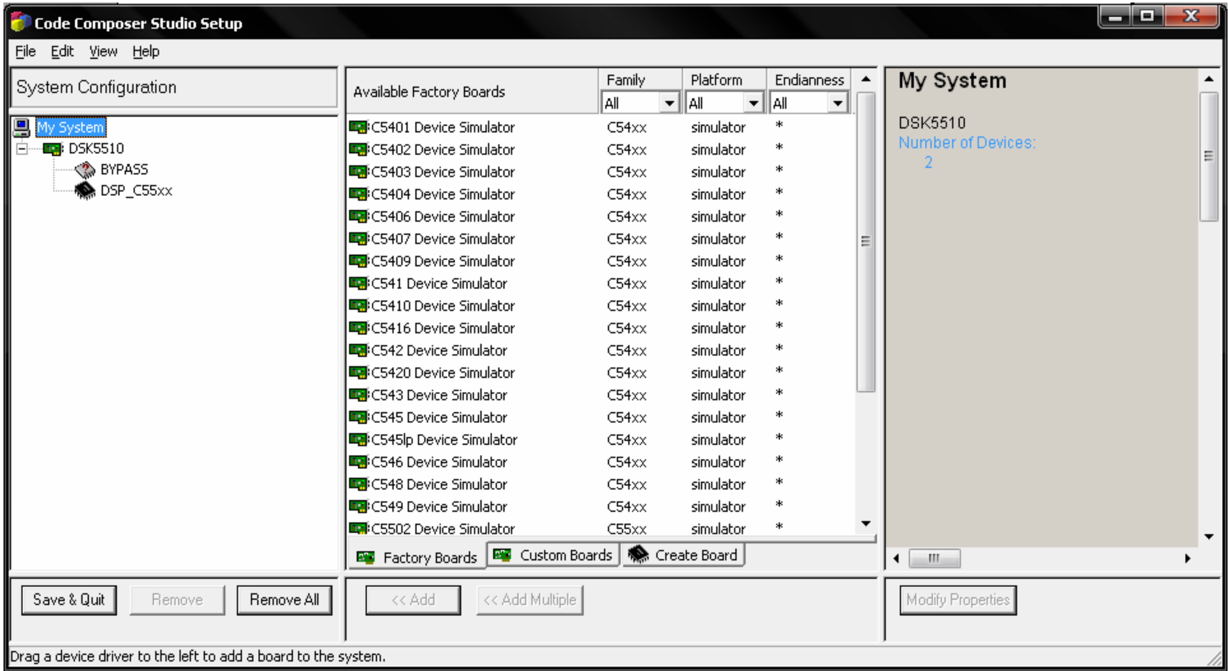


Рис. 6.3 Результат добавления выбранной платы

3. Теперь необходимо сохранить изменения, а для этого необходимо нажать кнопку «Save & Quit» (рис. 6.4) после чего подтвердить запуск CCS.

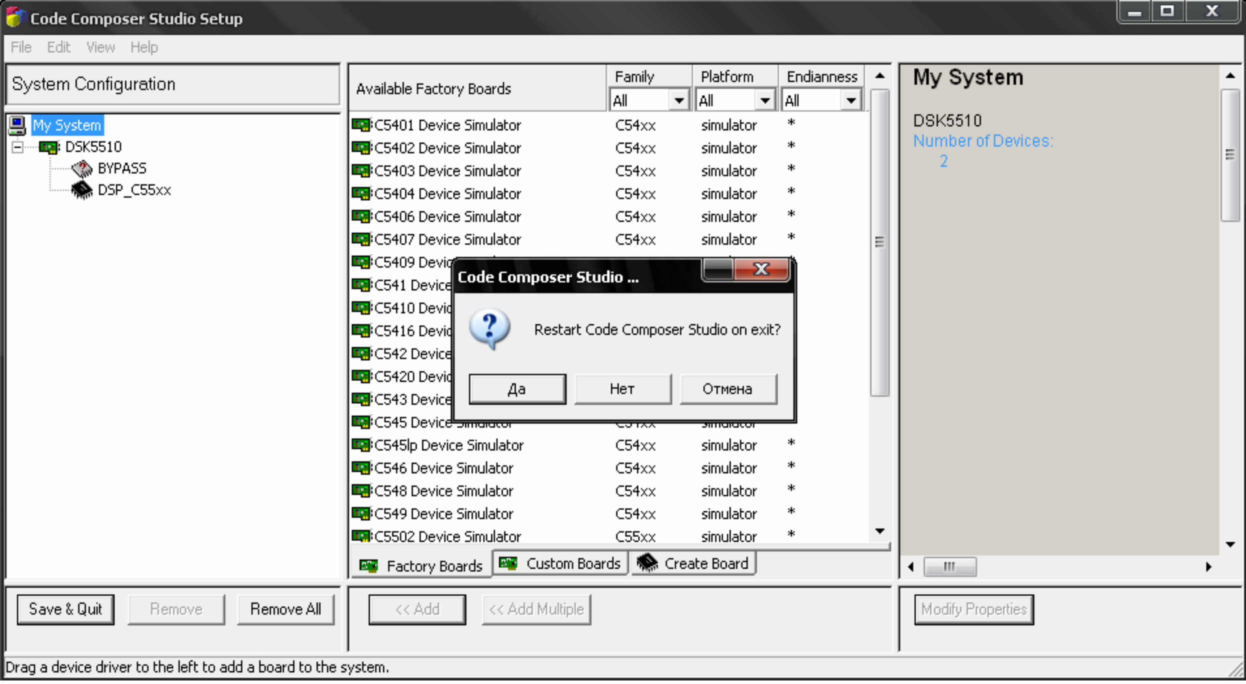


Рис. 6.4 Сохранение изменений и запуск ИСР CCS

4. После предыдущего шага запуститься ИСР CCS, как показано на рис. 6.5.

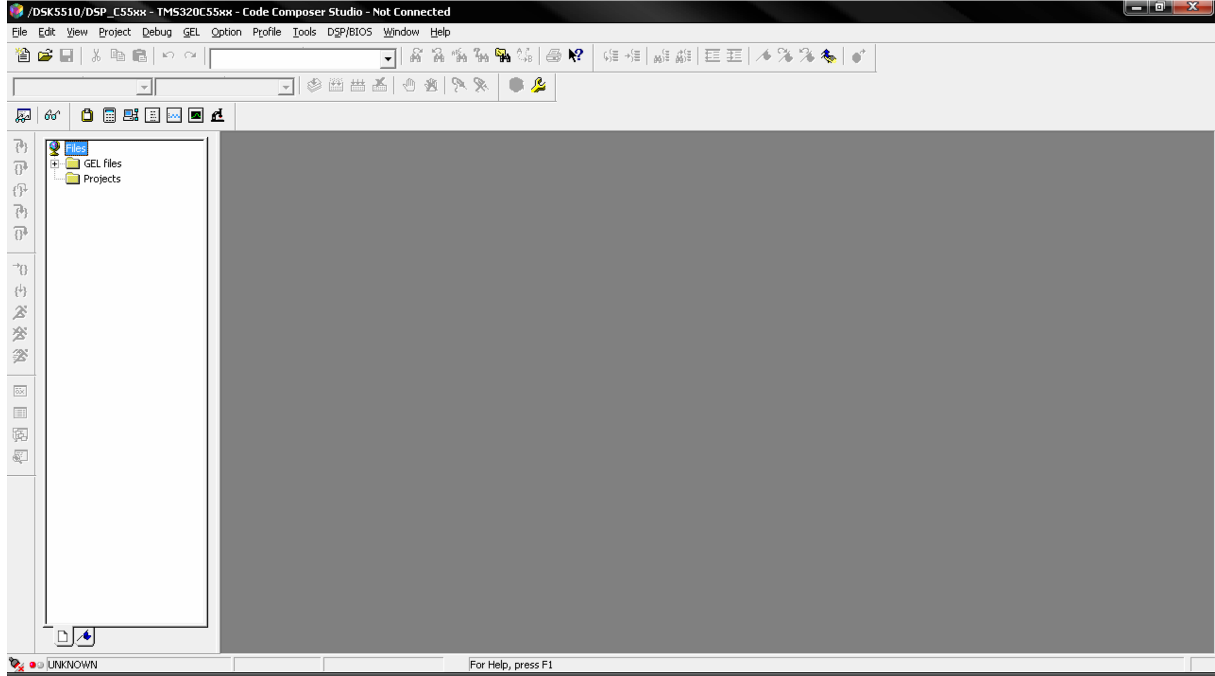


Рис. 6.5 ИСР CCS

Если этого не произошло и открылось окно как показано на рис. 6.6, то это означает, что или на плату DSK5510 не подключено питание, или компьютер не соединен с платой кабелем USB. После устранения этих неисправностей нажмите кнопку «Retry».

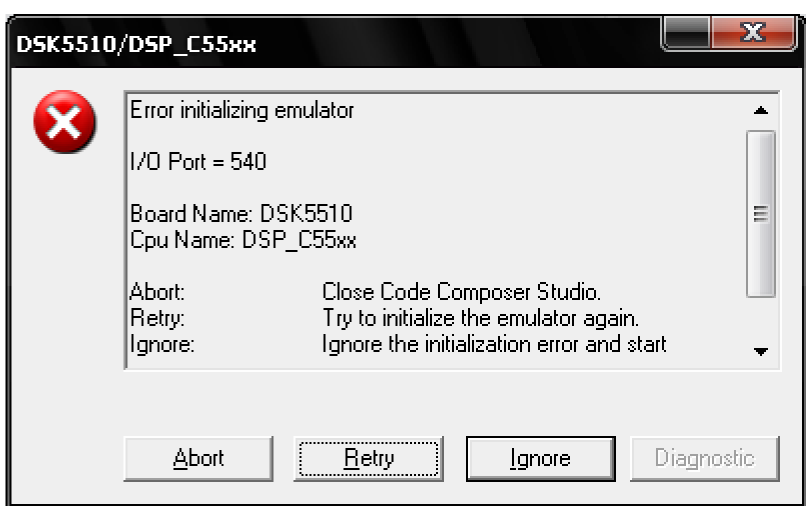


Рис. 6.6 Ошибка инициализации платы

Значок в нижнем левом углу окна ИСР CCS свидетельствует о том, что плата DSK5510 программно не подключена. Для подключения ИСР CCS к плате необходимо нажать сочетание клавиш «Alt+C» или зайти в раздел Debug →Connect. После этих действий окно ИСР CCS станет таким, как показано на рис. 6.7.

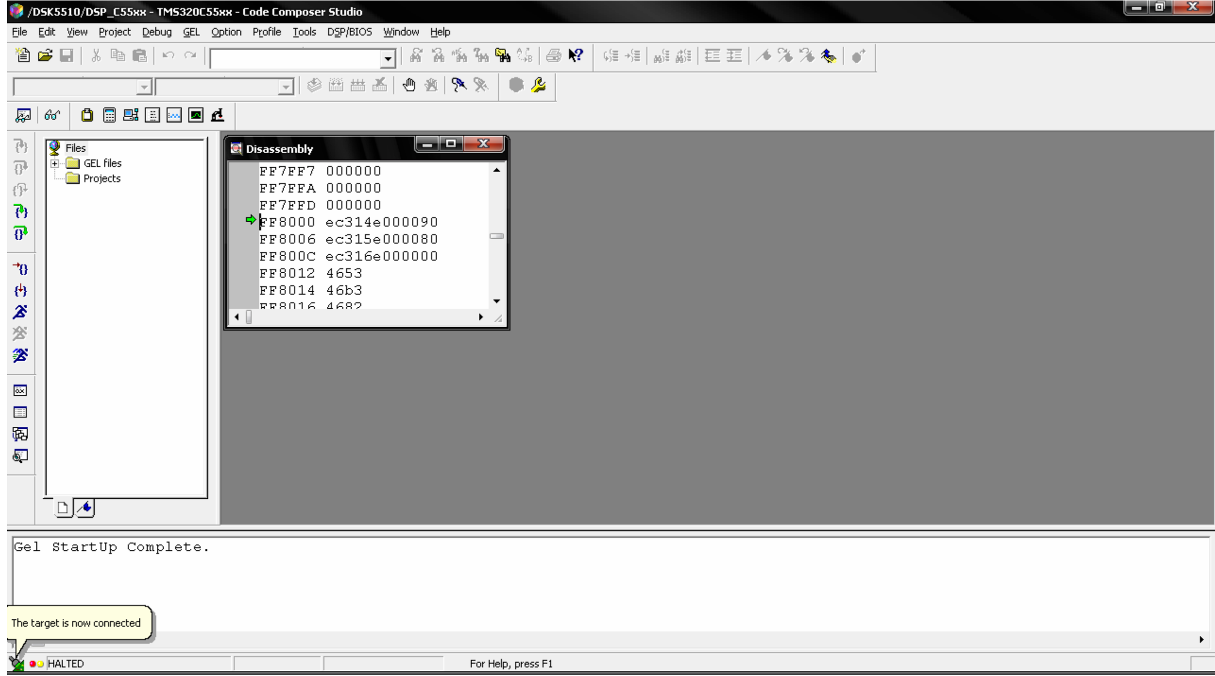


Рис. 6.7 Результат подключения ИСР CCS к плате DSK5510

5. После запуска ИСР CCS необходимо открыть проект «laba_1» (рис. 6.8).

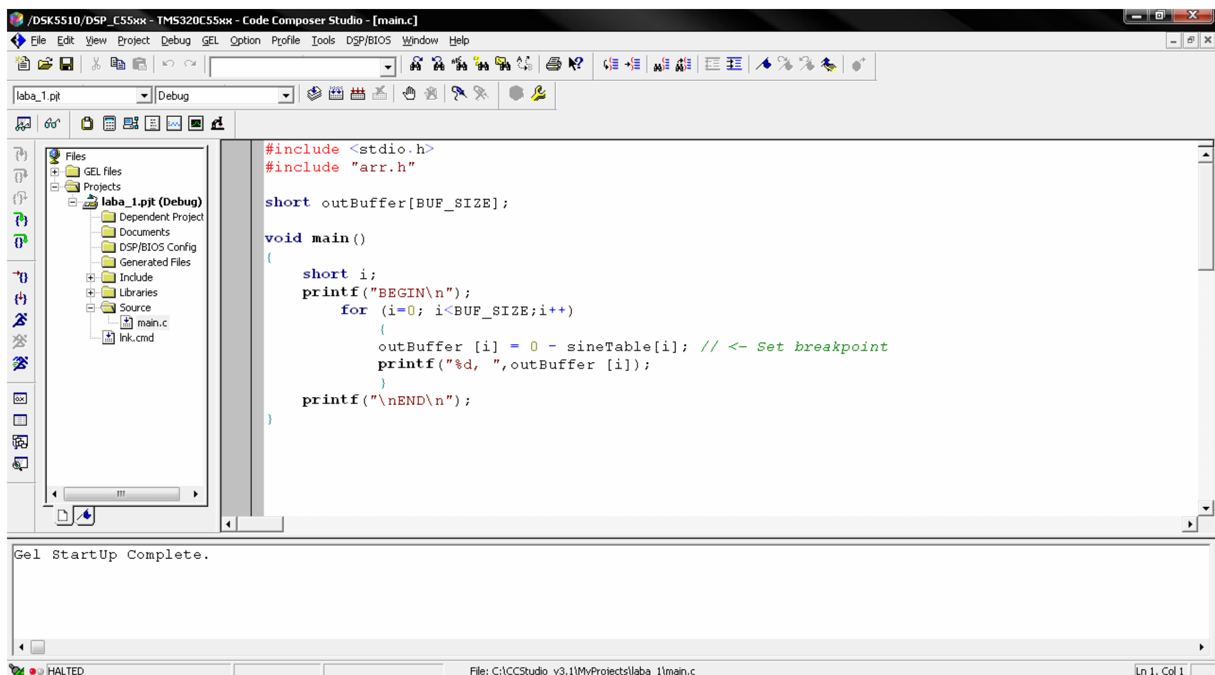


Рис. 6.8 Окно проекта с главным исполняемым файлом

6. Никаких изменений в проект вносить не потребуется, поэтому сразу нажмите кнопку . По окончании компиляции проекта на фоне окна ИСР CCS появится окно как показано на рис. 6.9, которое показывает про-

цесс загрузки выходного файла проекта (файл с расширением *.out) в память платы DSK5510.

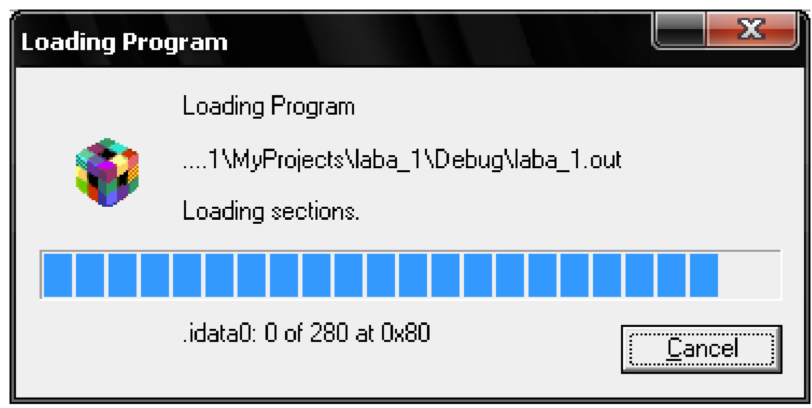


Рис. 6.9 Загрузка проекта в плату DSK5510

7. Теперь необходимо запустить режим Animate, нажав на кнопку , и окно ИСР CCS станет таким, как показано на рис. 6.10.

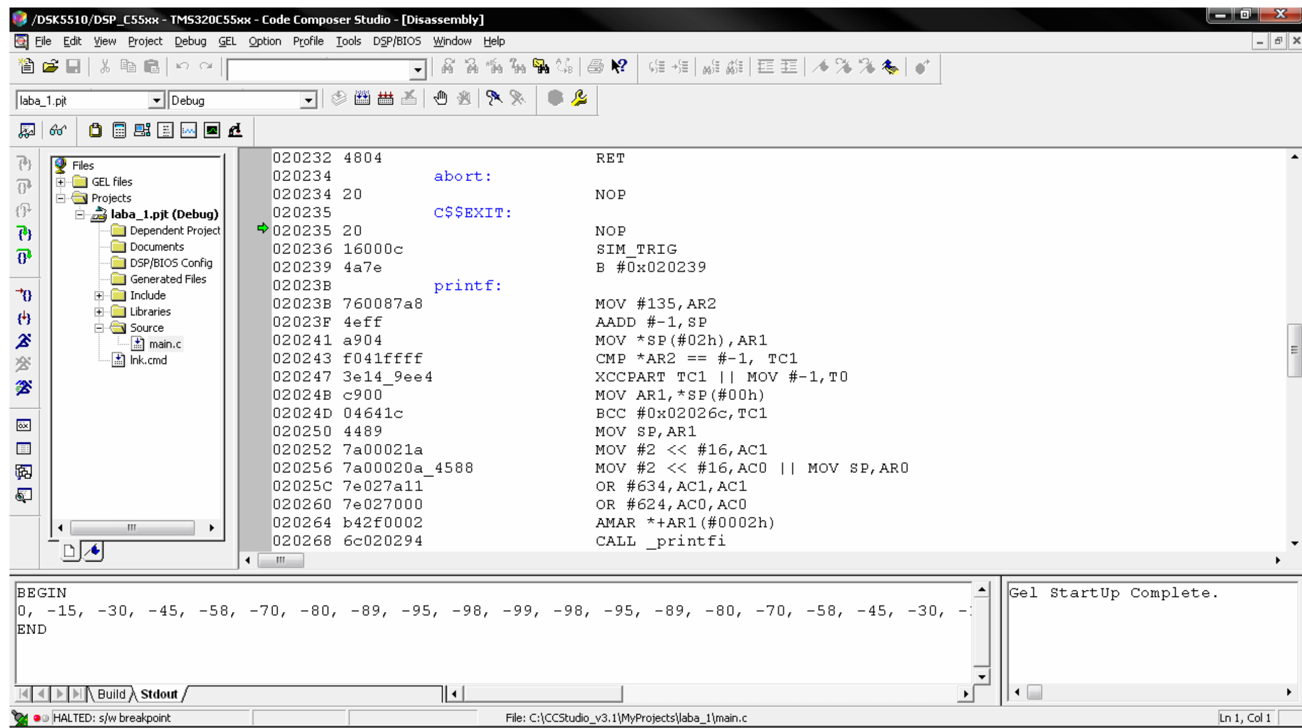


Рис. 6.10 Окно дизассемблера после запуска программы

8. Теперь можно проводить тестирование проекта аналогично тому, как это делалось на симуляторе.

Индивидуальные задания студентам выдаются во время лабораторной работы преподавателем.

Лабораторная работа 2

РАБОТА С DSP/BIOS ДЛЯ ГЕНЕРАЦИИ ЗВУКОВОГО СИГНАЛА ПЛАТОЙ DSK5510

1. Подключение файлов ввода/вывода с помощью точек зондирования

Необходимо запустить ИСР CCS в режиме симуляции. Создать новый проект, для этого, предварительно, необходимо в папке «MyProject» создать папку «laba_2» и в этой папке сохранить новый проект под именем «exp_1».

Теперь нужно создать и добавить к проекту исходный код программы, который находится в файле «main.c» (листинг 1.1). Суть исходного кода состоит в том, что он копирует данные из входного буфера в выходной. С помощью директивы #pragma DATA_SECTION(outBuffer, "expdata0") и #pragma DATA_SECTION(inBuffer, "expdata1") происходит определения (назначения) области памяти, где будут храниться входной и выходной буфер, в частности это области (секции) "expdata0" и "expdata1".

Листинг 1.1.

```
#pragma DATA_SECTION(outBuffer, "expdata0")
#pragma DATA_SECTION(inBuffer, "expdata1")
#define DATALENGTH 1000
int inBuffer[DATALENGTH];
int outBuffer[DATALENGTH];

void main()
{
          short i;
          for(i=0;i<DATALENGTH;i++)
                 {
                 outBuffer[i] =inBuffer[i];
                 }
}
```

Следующим шагом нужно подключить к проекту командный файл «lnk.cmd» из первой лабораторной работы (никаких изменений вносить не требуется). Именно в этом файле и определены секции памяти "expdata0" и "expdata1".

После того как файлы «main.c» и «lnk.cmd» добавлены к проекту, необходимо, добавить к проекту библиотеку «rts55.lib», находится: C:\CCStudio_v3.1\C5500\cgtools\lib\.

Прежде чем приступить к тестированию проекта, нужно создать в папке с проектом папку «data». В эту папку скопировать файлы «InputData.dat» и «OutputData.dat». Файл «OutputData.dat» пустой и его можно просто создать как текстовый, а «InputData.dat» содержит четное количество 32-

разрядных отсчетов синусоиды и первой строкой этого файла является заголовок. Структуру заголовка файла ввода/вывода можно увидеть на рис. 1.1.

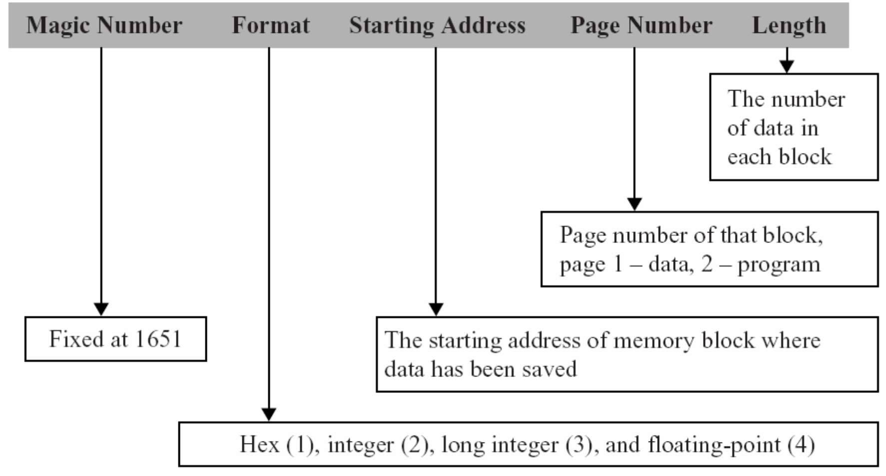


Рис. 1.1 Заголовок файла ввода/вывода

Файлы ввода/вывода подключаются при помощи точек зондирования, которые необходимо вначале определить. Для этого необходимо установить курсор на нужную строку кода и нажать кнопку , на панели быстрых кнопок. В текущем проекте будет две точки зондирования для файла ввода и для файла вывода, установлены они должны быть так, как показано на рис. 1.2.

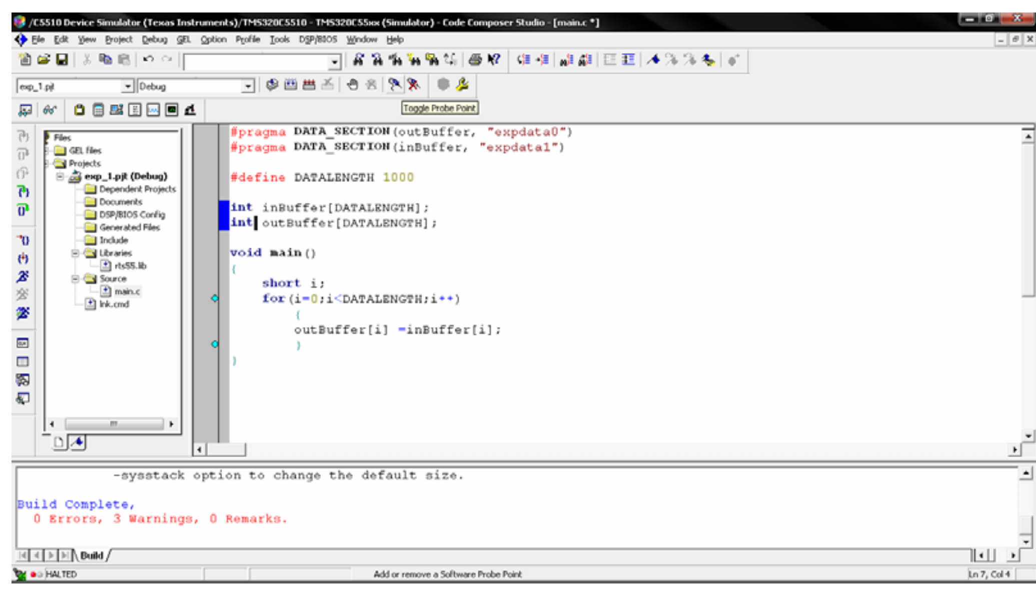


Рис. 1.2 Установка точек зондирования

После того как точки зондирования уставлены, обязательно, нужно нажать кнопку , для того, чтобы информация о месте нахождения точек зондирования сохранилась.

Для добавления файлов ввода/вывода следует выполнить следующую последовательность действий:

1. Выбрать раздел File → File I/O… как показано на рис. 1.3.

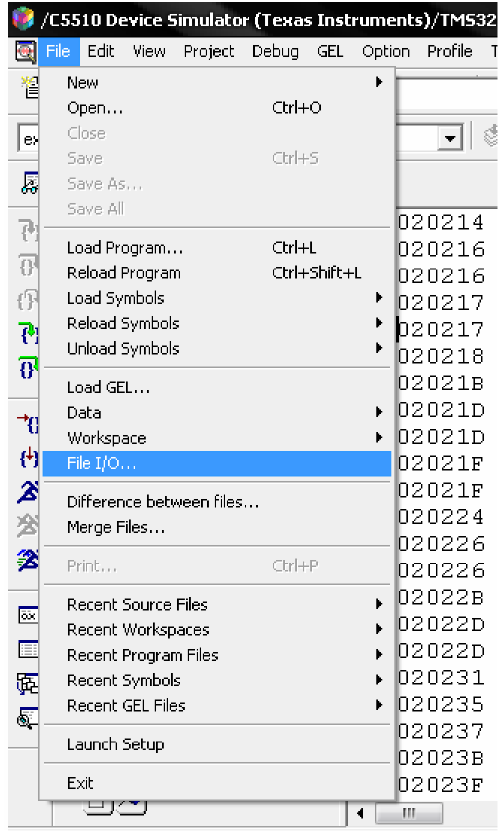


Рис. 1.3 Меню добавления файлов ввода/вывода

2. Откроется диалоговое окно настройки как показано на рис. 1.4. На вкладке «File Input» нажать кнопку «Add File» и открыть файл «InputData.dat». Диалоговое окно примет вид как показано на рис. 1.5.

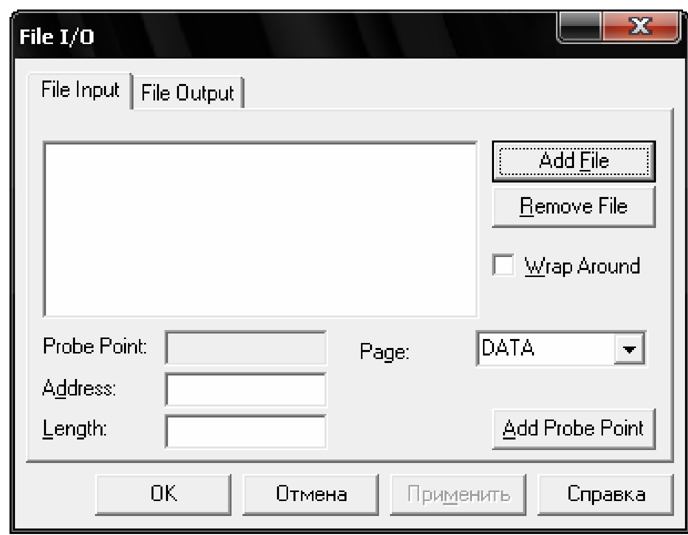


Рис. 1.4 Настройка подключаемых файлов

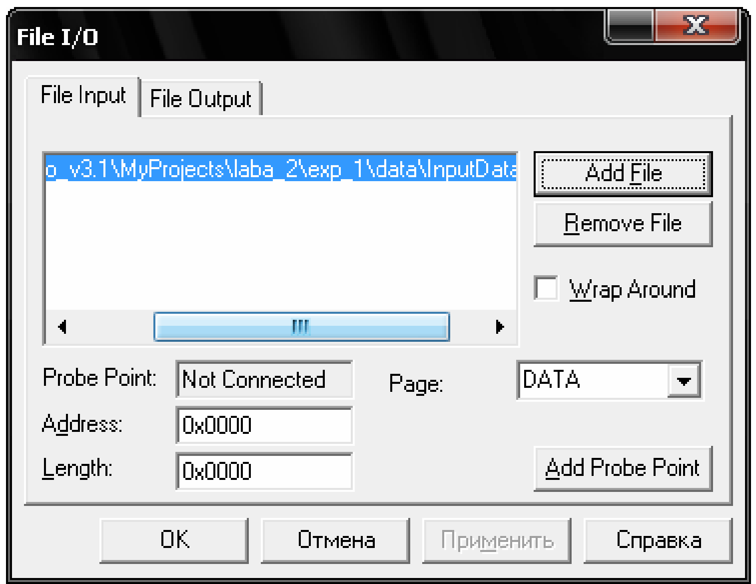


Рис. 1.5 Добавление файла ввода

3. После добавления файла нажать кнопку «Add Probe Point», откроется диалоговое окно как показано на рис. 1.6, которое позволит связать точку зондирования с файлом ввода.

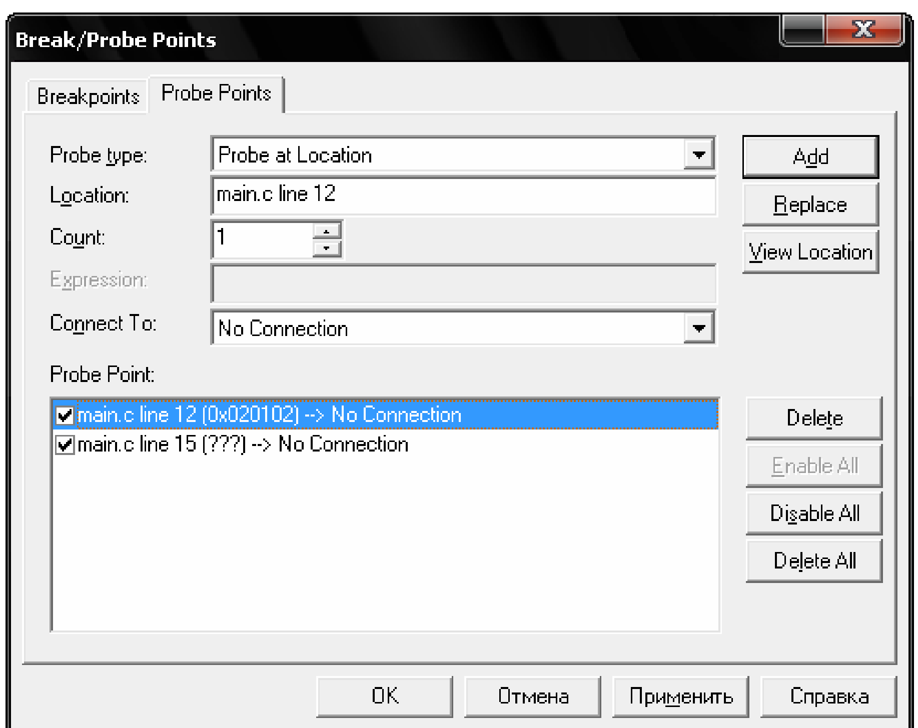


Рис. 1.6 Связывание точек зондирования с файлом

Для связывания выделите строку «main.c line12 …….» из списка «Probe Point:», после этого из списка «Connect To:» выберете файл «InputData.dat», как показано на рис. 1.7.

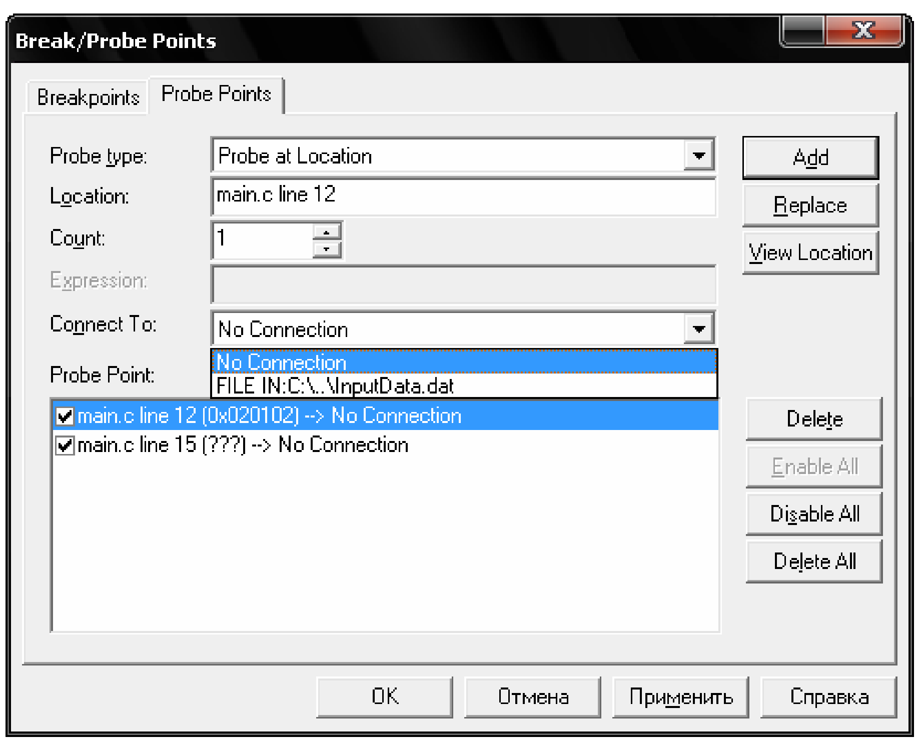


Рис. 1.7 Выбор файла для связывания

Теперь нажать кнопку «Replace», для сохранения настроек (рис. 1.8).

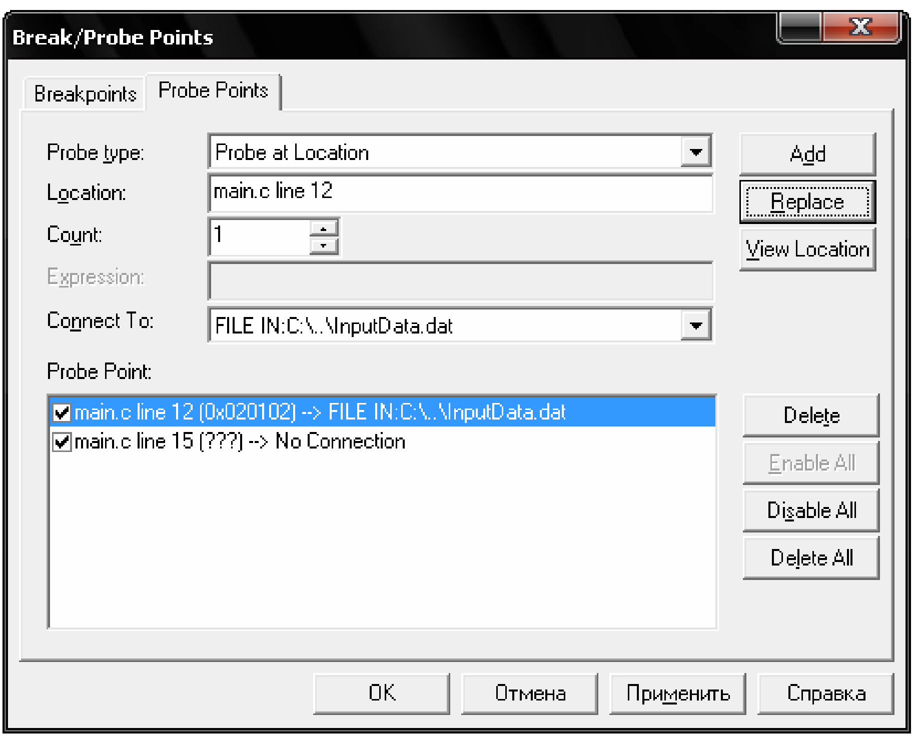


Рис. 1.8 Результат связывания файла ввода с точкой зондирования

Теперь нажать кнопку «OK».

4. В окне «File I/O» установить флажок в позицию «Warp Around» для циклического считывания данных из входного файла (рис. 1.9). Здесь также необходимо указать адрес приемного буфера, который вводится в поле «Address», и количество считываемых из файла отсчетов, которые указываются в поле «Length». В частности указать соответственно «inBuffer» и «1000», как показано на рис. 1.9.

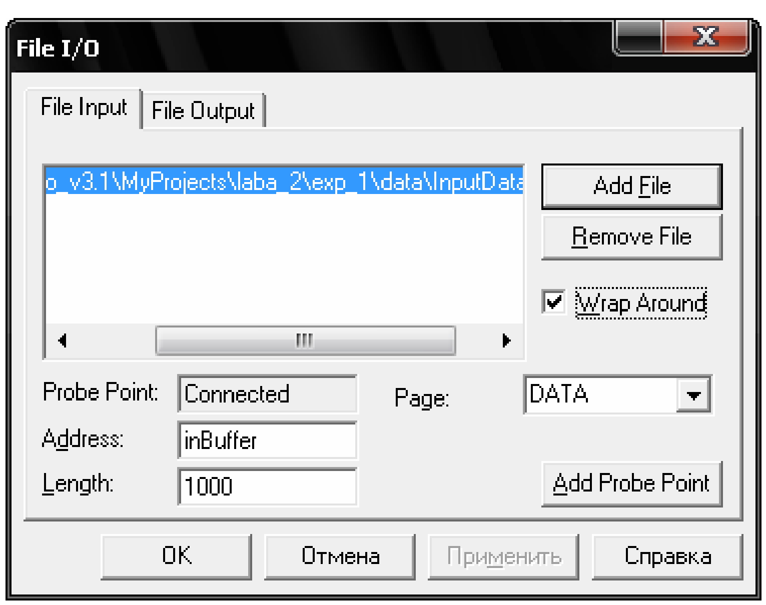


Рис. 1.9 Настройка подключаемых файлов

В итоге ИСР CCS примет вид как показано на рис. 1.10.

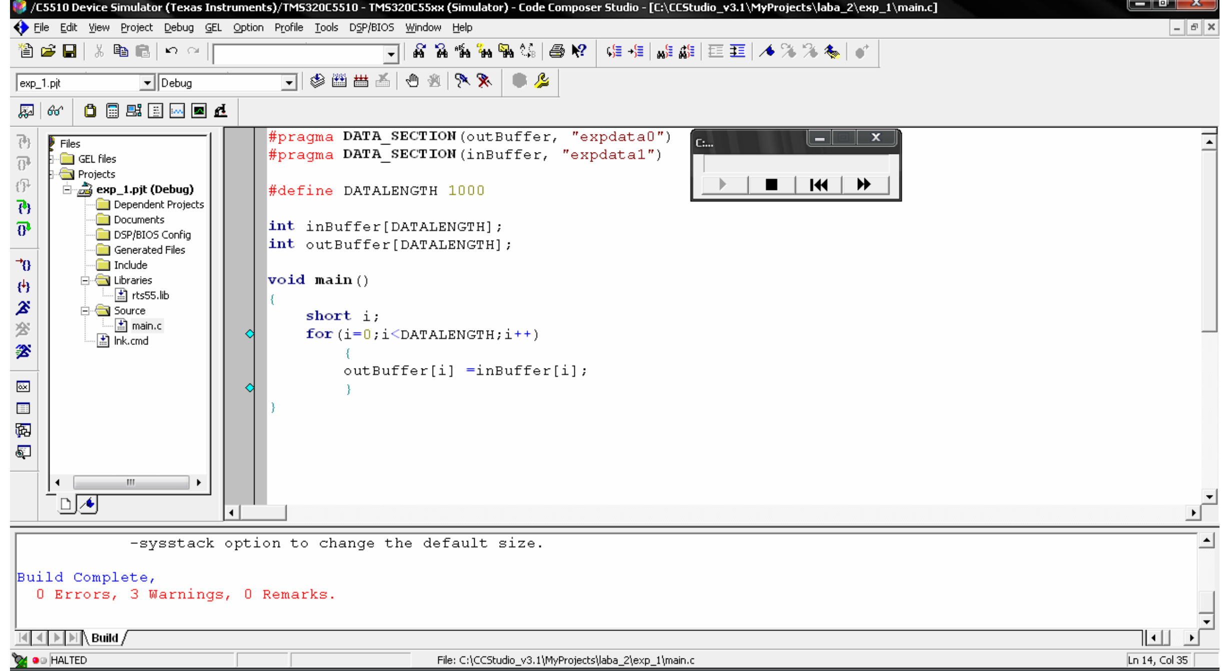


Рис. 1.10 ИСР CCS с подключенным файлом ввода

5. Следующим шагом нужно подключить файл вывода, для этого опять выбирать раздел File → File I/O… и перейти на вкладку «File Output» как показано на рисунке 1.11 нажать кнопку «Add File» и открыть файл «OutputData.dat».

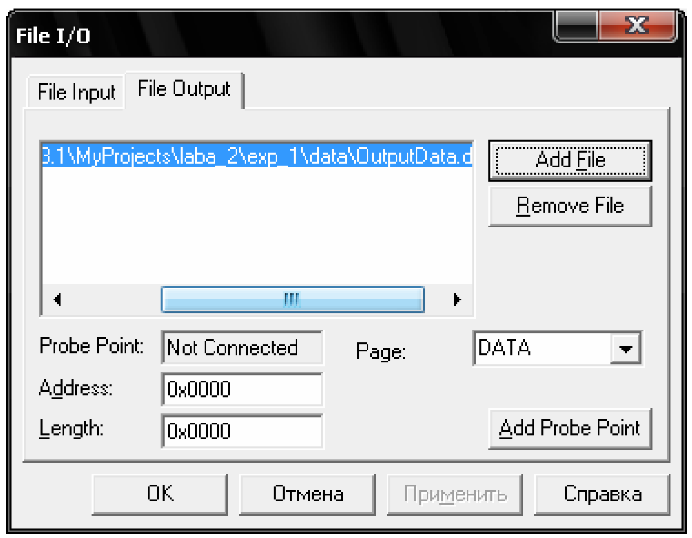


Рис. 1.11. Настройка подключаемых файлов

6. Также как и для файла ввода, файл вывода нужно связать с точкой зондирования, процедура отличается лишь выбором точки зондирования и файла, все действия показаны на рис. 1.12 и 1.13. По окончании действий нажать кнопку «OK».

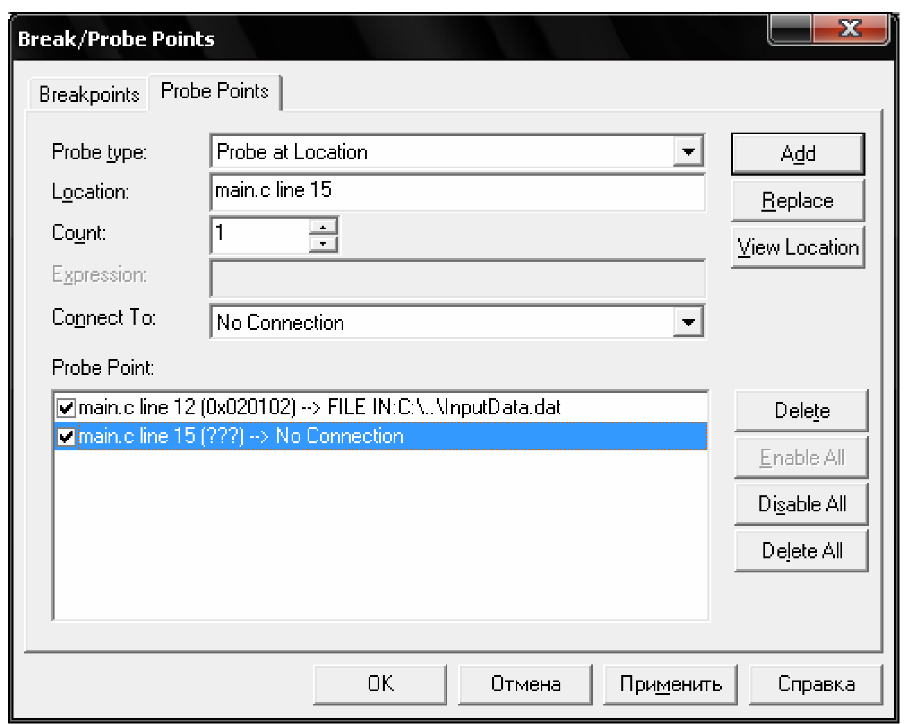


Рис. 1.12 Выбор точки зондирования

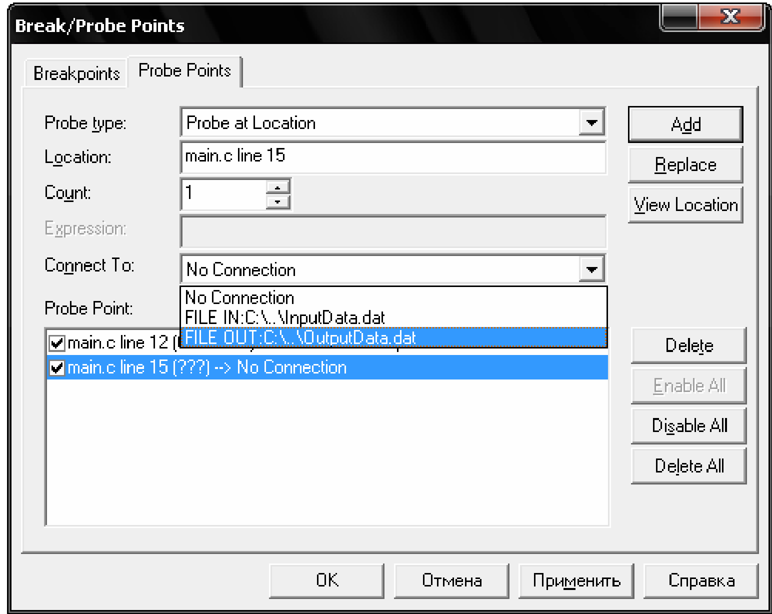


Рис. 1.13 Выбор файла вывода для связывания

7. Указать адрес приемного буфера, который вводится в поле «Address», и количество записываемых в файл отсчетов, которые указываются в поле «Length». В частности указать соответственно «outBuffer» и «1000», как показано на рис. 1.14, и нажать кнопку «OK».

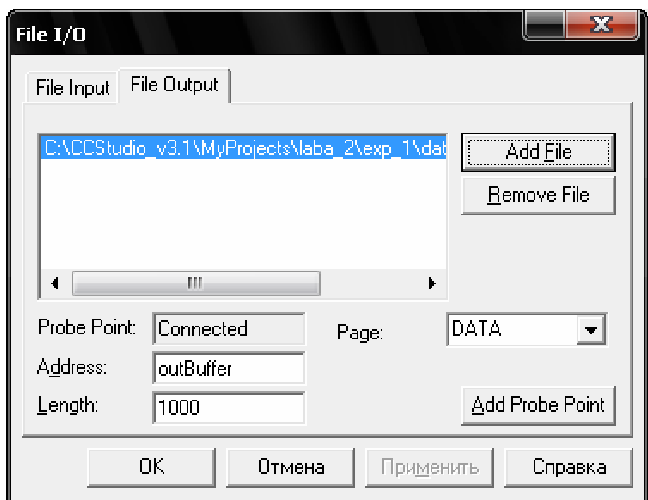


Рис. 1.14 Настройка подключаемых файлов

В итоге в ИСР CCS появится два окна – это счетчики входного и выходного файлов (рис. 1.15). В ни отображается процесс чтения входного и запись выходного файлов.

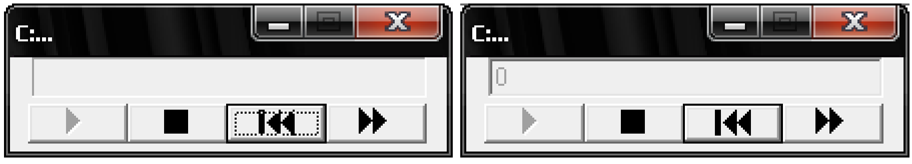


Рис. 1.15 Счетчики файлов

8. Теперь нужно запустить программу на выполнение (нажав). Вид ИСР CCS после выполнения программы показан на рис. 1.16. Выходной файл после выполнения будет содержать отсчеты из входного файла.

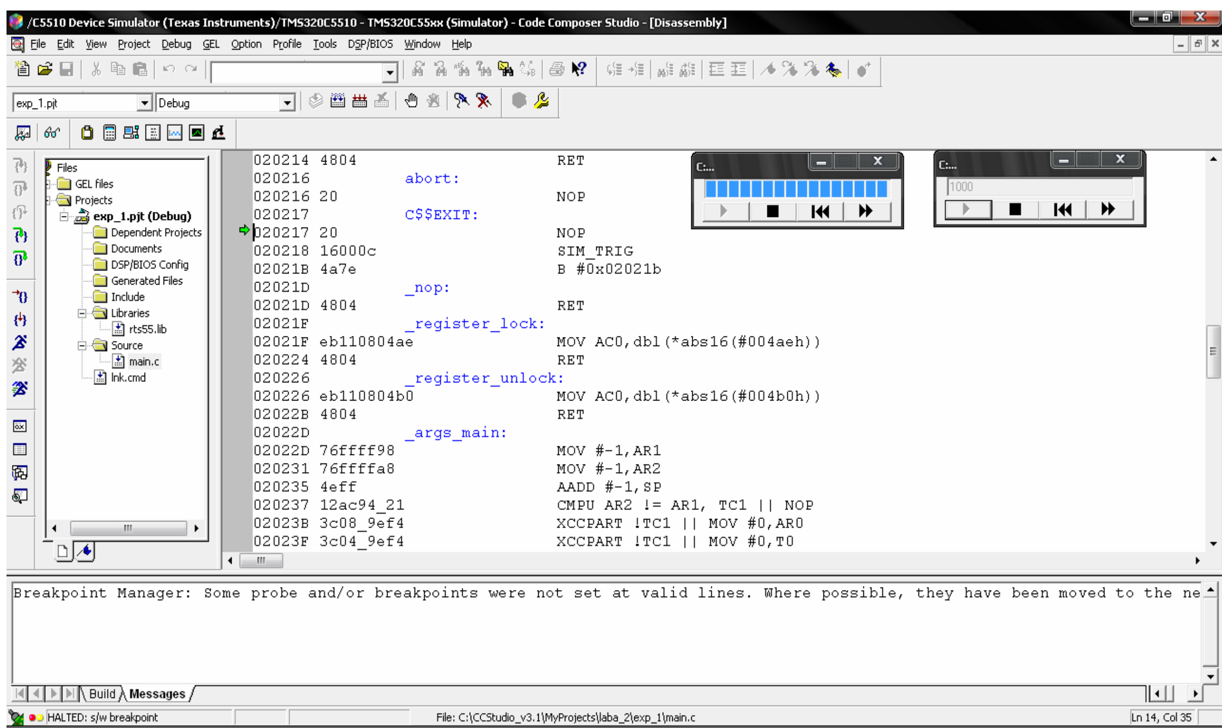


Рис. 1.16 Результат выполнения программы

9. Для повторного запуска программы нужно установить программный указатель на начало программы и на счетчиках файлов нажать клавишу , что соответствует установке на начало файла, причем выходной файл в этом случае будет создан заново.

10. Аппаратная реализация этого проекта не предусматривает внесение, каких-либо изменений, поэтому не рассматривается.

2. Работа с файлами по средствам функций языка C

Необходимо запустить ИСР CCS в режиме симуляции. Создать новый проект под именем «exp_2» в папке «laba_2».

Теперь нужно создать и добавить к проекту исходный код программы, который находится в файле «main.c» (листинг 2.1). Суть исходного кода состоит в том, что входной файл «inStereo.wav» (необходимо создать папку «Data» и скопировать этот файл в нее) открывается для чтения, а в выходные файлы «outLeftCh.wav» и «outRightCh.wav», будет записан звуковой моносигнал, полученный разложением входного файла на левый и правый канал.

Листинг 2.1.

```
#include <stdio.h>
#include "fiel_wav.h"

void main()
{
 FILE *inFile; // File pointer of input signal
 FILE *outLeftFile; // File pointer of left channel output
signal
 FILE *outRightFile; // File pointer of right channel output
signal

 short x[4];
 char wavHd[44];

 inFile = fopen("..\\data\\inStereo.wav", "rb");

 if (inFile == NULL)
          {
          printf("Can't open inStereo.wav");
          exit(0);
          }

 outLeftFile = fopen("..\\data\\outLeftCh.wav", "wb");
 outRightFile = fopen("..\\data\\outRightCh.wav", "wb");

 // Skip input wav file header
 fread(wavHd, sizeof(char), 44, inFile);

 // Add wav header to left and right channel output files
 fwrite(wavHeader, sizeof(char), 44, outLeftFile);
 fwrite(wavHeader, sizeof(char), 44, outRightFile);

 // Read stereo input and write to left/right channels
 while( (fread(x, sizeof(char), 4, inFile) == 4) )
          {
          fwrite(&x[0], sizeof(char), 2, outLeftFile);
          fwrite(&x[2], sizeof(char), 2, outRightFile);
          }

 fclose(inFile);
 fclose(outLeftFile);
 fclose(outRightFile);
}
```

Перед тем как записывать данные в wav файл, в него нужно записать заголовок, состоящий из 44 байт, этот заголовок по своей цели аналогичен заголовку подключаемого файла через точку зондирования, и содержит

основную информацию о «теле» файла. Описание wav заголовка приведено в табл. 2.1.

Таблица 2.1.

Номера байт	Описание
12 байтный заголовок RIFF файла	
0 – 3	Коды символов 'R' 'I' 'F' 'F' в ASCII
4 – 7	Длина файла минус первые 8 байт заголовка RIFF (4 байта для слова "WAVE" + 24 байтный заголовок, описывающий формат записанных данных + 8 байт для блока, описывающего данные + актуальный размер блока данных)
8 – 11	Коды символов 'W' 'A' 'V' 'E' в ASCII
24 байтный заголовок, описывающий формат записанных данных	
0 – 3	Коды символов 'f' 'm' 't' ' ' в ASCII (последний пробел)
4 – 7	Длина блока, описывающего формат записанных данных (всегда 16)
8 – 9	Заполнение файла. Всегда 1
10 – 11	Число каналов. 1 - моно, 2 - стерео
12 – 15	Частота дискретизации
16 – 19	Число байт в секунду
20 – 21	Байт в образце (sample). 1 для 8 бит моно, 2 для 8 бит стерео или 16 бит моно, 4 для 16 бит стерео
22 – 23	Количество бит которыми кодируется при оцифровки сигнала
Блок данных	
0 – 3	Коды символов 'd' 'a' 't' 'a' в ASCII
4 – 7	Длина данных в байтах

Заголовок файлов «outLeftCh.wav» и «outRightCh.wav» описывается в заголовочном файле «fiel_wav.h» и представлен в листинге 2.2. Файл «fiel_wav.h» должен находится в одной директории с прокутом.

Листинг 2.2.

```
// This wav file header is pre-calculated
// It can only be used for this experiment
short wavHeader[44]={
0x52, 0x49, 0x46, 0x46, // RIFF
0x2E, 0x8D, 0x01, 0x00, // 101678 (36 bytes + 101642 bytes
data)
0x57, 0x41, 0x56, 0x45, // WAVE
0x66, 0x6D, 0x74, 0x20, // Formatted
0x10, 0x00, 0x00, 0x00, // PCM audio
0x01, 0x00, 0x01, 0x00, // Linear PCM, 1-channel
0x40, 0x1F, 0x00, 0x00, // 8 kHz sampling
0x80, 0x3E, 0x00, 0x00, // Byte rate = 16000
0x02, 0x00, 0x10, 0x00, // Block align = 2, 16-бит/sample
0x64, 0x61, 0x74, 0x61, // Data
0x0A, 0x8D, 0x01, 0x00}; // 101642 data bytes
```

Следующим шагом нужно подключить к проекту командный файл «lnk.cmd» из первой лабораторной работы (никаких изменений вносить не требуется).

После того как файлы «main.c», «lnk.cmd» и «fiel_wav.h» добавлены к проекту, необходимо, добавить к проекту библиотеку «rts55.lib», находится: C:\CCStudio_v3.1\C5500\cgtools\lib\.

После всех действий проект компилируется и запускается на выполнения. Работа программы будет продолжаться некоторое время; о завершении работы будет свидетельствовать изменение надписи «ANIMATING» в нижнем левом углу на ту, что показана на рис. 2.1.

Рис. 2.1 Завершение выполнения программы

3. Работа с DSP/BIOS для генерации звукового сигнала платой DSK5510

Введение в DSP/BIOS

DSP/BIOS представляет собой ядро реального времени, которое предоставляет разработчику следующие сервисы:

1) гибкий планировщик задач с возможностью мультизадачной работы;
2) уровень аппаратной абстракции для работы с периферийными устройствами процессора;
3) устройство независимого ввода/вывода для передачи потоков данных в реальном времени;
4) средства анализа поведения ЦОС-приложения и обмена с ним данными в реальном времени.

Практически, это оптимизированная для ЦОС-приложений мультизадачная операционная система реального времени.

В отличие от большинства существующих коммерческих ядер, DSP/BIOS поставляется бесплатно как составная часть среды разработчика Code Composer Studio, что весьма немаловажно, с учетом цен на продукты такого класса. При этом DSP/BIOS включает в свой состав оптимизированные для работы с конкретным ЦСП библиотеки и компоненты.

Рассматривая применение операционных систем в ЦОС-приложениях, необходимо помнить три очень важных момента. Первый – ограниченность ресурсов встроенной системы, которая заставляет очень жестко подходить к дополнительным расходам, например, памяти. Второй момент –

необходимость минимизации временных накладных расходов, поскольку мы работаем в реальном времени и желательно максимально использовать всю производительность ЦСП для решения основной задачи. Третий момент относится именно к системам ЦОС и заключается в следующем: чем сложнее операционная среда, тем сложнее предсказать ее поведение, что никак не допустимо в алгоритмах ЦОС. Следовательно, надо иметь средства однозначного задания поведения системы и средства контроля ее поведения.

Для реализации этих достаточно противоречивых требований при построении DSP/BIOS была выбрана следующая модель: DSP/BIOS является статически конфигурируемым ядром реального времени со статически определяемой приоритетной моделью исполнения процессов. Что же достигается такой моделью? Во-первых, минимизация дополнительных расходов памяти, поскольку в исполняемый код включаются только модули, необходимые при реализации данной задачи, а не вся операционная среда. Во-вторых, достигается оптимизация производительности процессора, поскольку большинство статических вызовов после компиляции упаковываются в несколько команд. Далее, поскольку время выполнения команд ЦСП известно, конфигурация системы задается статически и тоже известна, модель исполнения и система приоритетов также задается статически, то мы имеем полностью предсказуемое и однозначно определенное поведение системы.

Отметим, что наравне со статическими средствами конфигурации, в стандартные средства DSP/BIOS включаются и средства динамического создания процессов и распределения памяти, что позволяет разработчику наравне с гарантированной моделью поведения основных критических узлов использовать и все преимущества полноформатной операционной системы при создании сложных и гибких систем.

Как уже говорилось, при построении любой операционной системы важным вопросом является дополнительный расход ресурсов на предоставляемые ей сервисы. В операционной системе реального времени, кроме расхода ресурсов процессора и памяти, добавляется еще и фактор расхода времени. С учетом достаточно ограниченных ресурсов систем ЦОС, а также очень жестких временных рамок, очень большое значение имеет правильный выбор операционной среды. С одной стороны, ОС должна предоставлять достаточно гибкий и удобный набор функциональных возможностей, а с другой – расходовать как можно меньше ресурсов. При построении DSP/BIOS была выбрана многоуровневая модель построения системных сервисов. Для каждой области имеются несколько типов вызовов, аналогичных по назначению, но различных по функциональным возможностям и, соответственно, по расходуемым ресурсам. Так, существует

нить типа «прерывание» с базовыми возможностями и очень малым временем реакции и существует нить типа «задача» с гораздо большим набором предоставляемых сервисов, но и с другими временными параметрами переключения и отработки. Аналогично, существует статическое распределение памяти, которое вообще не потребляет ресурсов процессора, так как задается утилитами конфигурации. Кроме того, существует динамическое распределение памяти, использование которого требует включения в исполняемый код соответствующего модуля и работает через системные вызовы, но позволяет разным процессам использовать ОЗУ совместно. Такая модель, совместно с моделью конфигурирования самого ядра DSP/BIOS, когда в выходной код добавляются только используемые компоненты, позволяет сочетать быстрое время реакции и малый объем дополнительного кода с широкими функциональными возможностями операционной системы реального времени.

Одной из задач, решаемых DSP/BIOS, является предоставление разработчику уровня аппаратной абстракции, т. е. единого логического интерфейса работы с аппаратной частью системы ЦОС, при этом учет особенностей работы аппаратных узлов того или иного ЦСП возлагается на инструментальные средства. При этом имеется как интерфейс к программированию узлов ядра процессора, таким как таймер и контроллер ПДП, так и стандартный интерфейс для организации каналов ввода/вывода, включающий в себя драйверы периферийных устройств, драйверы портов, к которым они подключаются, например McBSP, и средства организации стандартных потоков ввода/вывода.

При правильном использовании средств DSP/BIOS вполне достижим уровень аппаратной абстракции, при котором получается полная междуплатформенная переносимость приложения. Фактически, сбывается очень давняя мечта разработчиков – если на конечной стадии разработки выяснится, что ресурсов выбранного ЦСП не хватает или, что еще более болезненно, законченная и с таким трудом упакованная задача требует расширения, то можно просто перейти на более мощный ЦСП, при этом можно даже поменять платформу, ничего не меняя в написанном, проверенном и отлаженном коде. Еще одно преимущество переносимости – обратный процесс оптимизации. Можно взять мощный ЦСП, спокойно написать и отладить задачу, имея запас производительности для сервиса и отладки, на обкатку вариантов реализации и на оптимизацию участков кода, а затем, отладив и оптимизировав задачу, перенести ее на оптимальный по параметрам и цене ЦСП для серийного выпуска устройства. С учетом наличия ЦСП различной мощности и объема памяти в совместимых корпусах, возможности практически безболезненного переноса позволяют производить

как функционально-стоимостную оптимизацию, так и модернизацию устройства без дорогостоящих затрат на модернизацию аппаратных составляющих устройства, например, на переразводку печатных плат. Как пример аппаратной совместимости, приведем ЦСП семейства C5000, где в одном корпусе и совместимыми по выводам выпускаются ЦСП от ‘VC5401 (50 MIPS, 8 Кслов ОЗУ, цена менее $6) до ‘VC5416 (160 MIPS, 128 Кслов ОЗУ), причем параллельно с ЦСП выпускается и совместимая по выводам номенклатура компонентов поддержки, таких как микросхемы источников и супервизоров питания.

Использование библиотек поддержки микросхем (Chip Support Library), абстрагирующих аппаратный уровень основных функций, библиотек ЦОС-функций (DSP Library), предоставляющих оптимизированные для каждой платформы основные ЦОС-алгоритмы с единым интерфейсом, интерфейсов DSP/BIOS для управления выполнением приложения, а также оптимизирующих компиляторов с языка С для написания алгоритмов позволяет создать реально абстрагированное, «отвязанное» от конкретной платформы ЦОС-приложение. Причем это приложение не будет являться лабораторным экспериментом, а будет реальной «боевой» разработкой с достаточным для практического применения уровнем оптимальности исполнения.

Наличие операционной среды, кроме удобства разработки, дае еще один плюс – все компоненты работы с аппаратным обеспечением уже отлажены и работают – не надо тратить время на их отладку.

Еще одной важной особенностью DSP/BIOS является наличие визуальных средств контроля и протоколирования порядка и последовательности исполнения нитей, а также средств отслеживания критических мест, что позволяет легко оценить ход исполнения приложения и быстро отследить и устранить критические участки.

Компоненты DSP/BIOS

Основные компоненты DSP/BIOS в среде разработчика Code Composer Studio, показанные на рисунке 3.1.

На хост-компьютере пишутся исходные файлы проекта (на С, С++ или ассемблере), использующие интерфейсы DSP/BIOS API. Затем с помощью утилиты конфигурирования определяются те компоненты, которые будут использоваться в проекте, и их параметры. Исходные тексты, библиотека DSP/BIOS и файлы конфигурации образуют проект, из которого средствами генерации кода получается исполняемый файл, загружаемый в ЦСП. Встроенные средства анализа Code Composer Studio позволяют производить мониторинг исполнения программы.

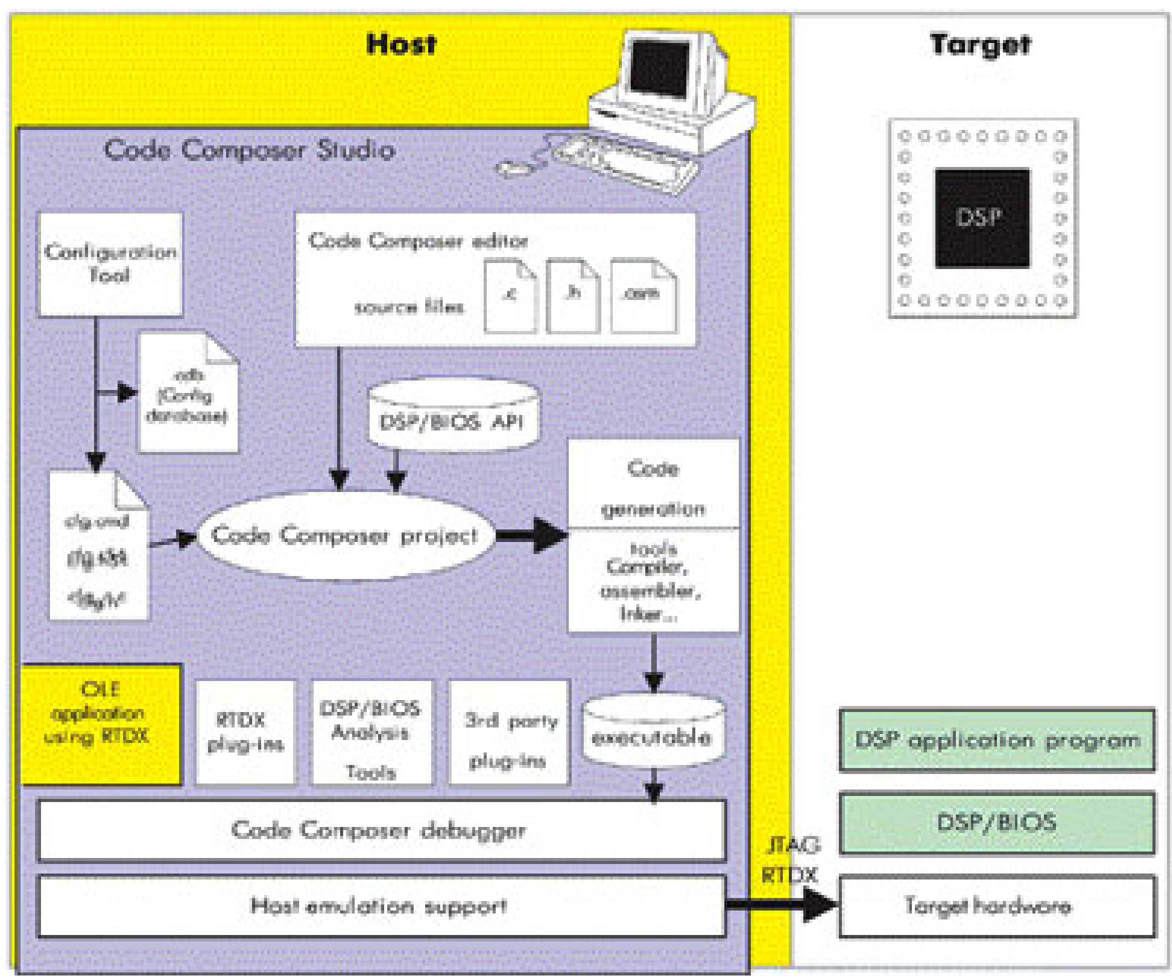


Рис. 3.1 Компоненты DSP/BIOS

DSP/BIOS – достаточно сложная статически конфигурируемая система, для задания конфигурации которой используется специальная графическая утилита, интегрированная в Code Composer Studio (рис. 3.2). Утилита конфигурации – это специализированный визуальный редактор, который позволяет выбирать, какие модули DSP/BIOS будут включены в систему, а какие нет, а также задавать их параметры. Все параметры задаются статически до компиляции. При этом утилита конфигурации позволяет оценить объем требуемой под служебные нужды памяти, а также проверить соответствие заданных параметров, что позволяет избежать ошибок уже на начальном этапе конфигурации системы и сэкономить время на старте. Еще одной функцией утилиты конфигурации является привязка проекта к конкретной аппаратной платформе. Именно в утилите конфигурации задаются параметры карты памяти, распределения прерываний и привязки тактовой частоты процессора к системным часам реального времени. При этом для различных ЦСП существуют уже готовые начальные схемы конфигурации DSP/BIOS.

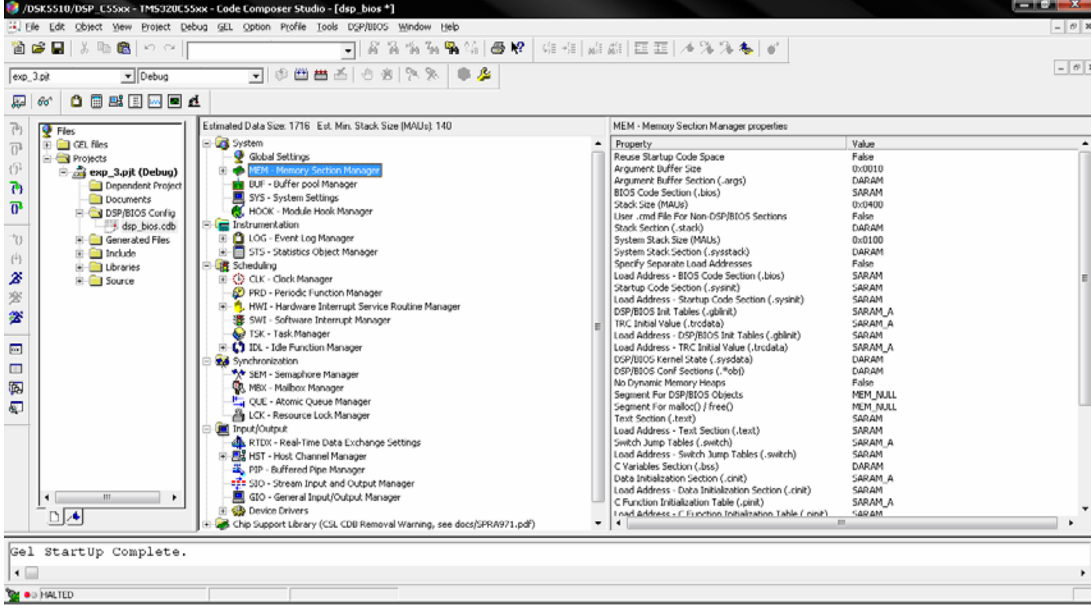


Рис. 3.2 Утилита конфигурации

Все модули ядра реального времени DSP/BIOS могут быть разбиты на 6 групп, каждая их которых предоставляет свой интерфейс (API) пользовательским приложениям:

1) группа функций анализа реального времени и обмена данными;
2) функции аппаратной абстракции;
3) функции ввода/вывода, независимые от аппаратных устройств;
4) функции управления выполнением нитей;
5) функции взаимодействия и синхронизации нитей;
6) прочие функции.

Графический пользовательский интерфейс (Graphical User Interface – GUI) упрощает разработку системы:

1) автоматически подключает необходимые библиотеки;
2) автоматически обрабатывает вектора прерываний и перезапуск;
3) определяет конфигурацию системной памяти (создает CMD-файл);
4) когда CDB-файл сохранен, инструмент конфигурирования создает 5 дополнительных файлов.

Когда CDB-файл добавляется в проект, ИСР CCS автоматически добавляет сгенерированные файлы в директорию проекта (табл. 3.1).

Таблица 3.1

Имя_файла.cdb	Конфигурационная база данных
Имя_файлаcfg_c.c	Си-код, создаваемый инструментом конфигурирования
Имя_файлаcfg.s55	ASM-код, создаваемый инструментом конфигурирования
Имя_файлаcfg.cmd	Командный файл компоновщика
Имя_файлаcfg.h	Заголовочный файл для *cfg_c.c
Имя_файлаcfg.h55	Заголовочный файл для *cfg.s28

Практическая реализация проекта с помощью DSP/BIOS

Необходимо запустить ИСР CCS в режиме симуляции. Создать новый проект под именем «exp_3» в папке «laba_2».

Теперь нужно создать и добавить к проекту исходный код программы, который находится в файле «main.c» (листинг 3.1). Суть исходного кода: плата DSK5510 конфигурируется таким образом, чтобы воспроизвести звуковой сигнал на частотах 1, 2, 3, 4, 5 КГц, который генерирует функция «sinewave()». Для этого в программе настраивается кодек AIC23 – на плате он отвечает за АЦП сигналов, которые поступают на линейный вход, и ЦАП сигнала подаваемого на линейные выходы. На плате кодек AIC23 представляет собой небольшую микросхему (AIC23BI), к которой подключены линейные входы/выходы, в память которой записаны все функции для работы. Чтобы можно было работать с этими функциями, необходимо подключить заголовочный файл «dsk5510_aic23.h». Но прежде чем начинать работу с кодеком нужно произвести инициализацию плата, для этого подключается заголовочный файл «dsk5510.h», в котором объявлена функция инициализации платы: «DSK5510_init()».

Листинг 3.1.

```
#include <math.h>
#include "dsk5510.h"
#include "dsk5510_aic23.h"
#include "dsp_bioscfg.h"
#define two_pi 6.28
/* Codec configuration settings */
DSK5510_AIC23_Config config = {
    0x0017,  //  0  DSK5510_AIC23_LEFTINVOL   Left  line  input
channel volume
    0x0017,  //  1  DSK5510_AIC23_RIGHTINVOL  Right  line  input
channel volume
    0x00d8,  //  2  DSK5510_AIC23_LEFTHPVOL   Left  channel  head-
phone volume
    0x00d8,  //  3  DSK5510_AIC23_RIGHTHPVOL  Right  channel  head-
phone volume
    0x0011,  //  4  DSK5510_AIC23_ANAPATH     Analog  audio  path
control
    0x0000,  //  5  DSK5510_AIC23_DIGPATH     Digital  audio  path
control
    0x0000, // 6 DSK5510_AIC23_POWERDOWN  Power down control
    0x0043, // 7 DSK5510_AIC23_DIGIF      Digital audio inter-
face format
    0x0081, // 8 DSK5510_AIC23_SAMPLERATE Sample rate control
    0x0001   //  9  DSK5510_AIC23_DIGACT      Digital  interface
activation
};
```

```
/*a prototype of function is a generating sound*/
short sinewave(float n);

void main()
{
 DSK5510_AIC23_CodecHandle hCodec;
 short sec, msec, sample;
 float x;
 short frq=1;
 int gen_wav;
 /* Initialize the board support library, must be called first
*/
 DSK5510_init();

 /* Start the codec */
 hCodec = DSK5510_AIC23_openCodec(0, &config);

 /* Generate sine wave for 5 seconds */
 for (sec = 0; sec < 5; sec++)
     {
     x=0;
     for(msec = 0; msec < 1000; msec++)
          {
         for (sample = 0; sample < 48; sample++)
          {
             gen_wav=sinewave(frq*x);

                /* Send a sample to the left channel */
             while (!DSK5510_AIC23_write16(hCodec, gen_wav));

             /* Send a sample to the right channel */
             while (!DSK5510_AIC23_write16(hCodec, gen_wav));

           /* Step of function */
           x+=two_pi/48;
               }
          }
     frq++;
    }
 /* Close the codec */
 DSK5510_AIC23_closeCodec(hCodec);
}

short sinewave(float n)
{
 return((short)(sin(n)*16484));
}
```

Теперь из папки «C:\CCStudio_v3.1\C5500\dsk5510\include\» в папку с проектом копируется «dsk5510_aic23.h» и «dsk5510.h».

Следующим шагом необходимо создать файл конфигурации DSP/BIOS, вследствие того, что объем настроек достаточно большой, а на данном этапе стоит задача ознакомления с основами работы DSP/BIOS, то основа конфигурации DSP/BIOS будет взята из готового конфигурационного файла примера, который поставляется с пакетом ИСР CCS. Итак, нужно из папки «C:\CCStudio_v3.1\examples\dsk5510\bsl\tone\» открыть файл конфигурации (File → Open…) «tone.cdb», после чего сохранить его в папке с проектом под именем «dsp_bios.cdb». После этого его нужно добавить к проекту.

Последним шагом в формировании пакета программы станет подключение библиотеки «dsk5510bslx.lib», которая находится: «c:\CCStudio_v3.1\C5500\dsk5510\lib\». Данная библиотека работает с моделью памяти: «Large», поэтому необходимо настроить опции компиляции.

Для настройки опций компиляции проекта необходимо:

1. Выбрать раздел Project → Build Options… и на вкладке «Compiler» выбрать раздел «Advanced», далее необходимо установить модель памяти «Large», рис. 3.3.

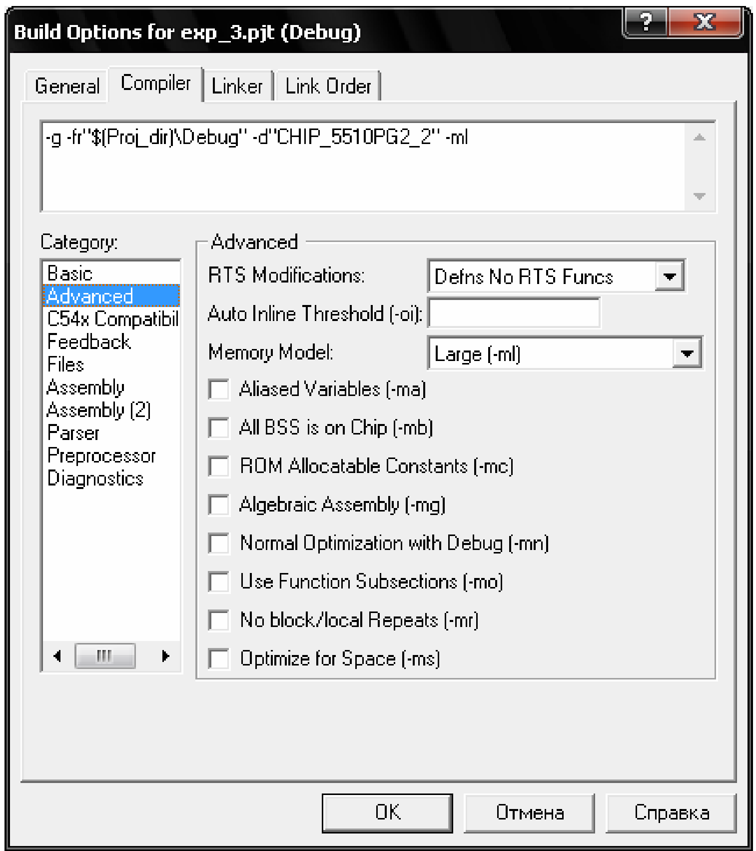


Рис. 3.3 Опции компиляции

2. Указать компилятору тип ЦСП, для этого в разделе (категории) «Preprocessor» в поле «Pre-Define Symbol» указать «CHIP_5510PG2_2», рис. 3.4.

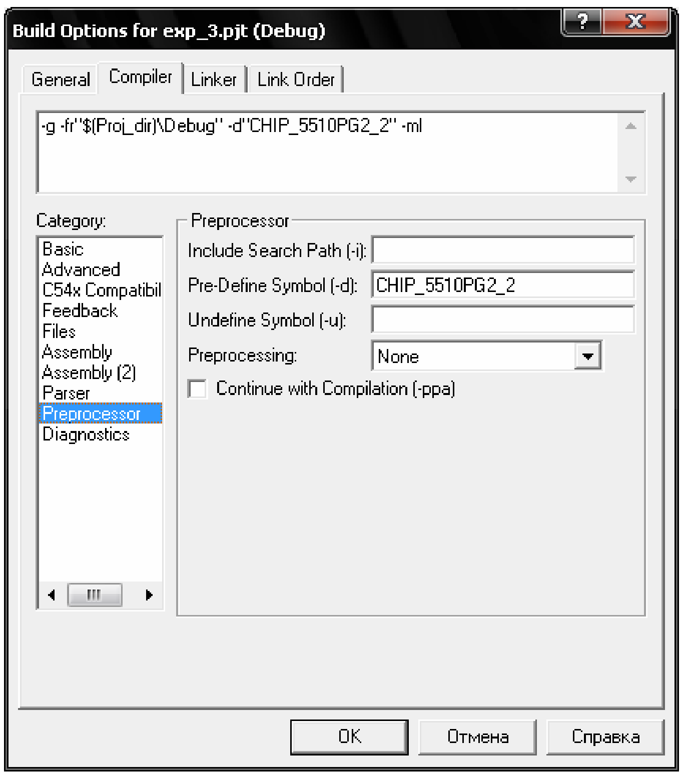


Рис. 3.4 Опции компиляции для процессора

По окончании всех действий проект можно запускать на компиляцию и сборку. Однако ИСР CCS выдаст ошибку при попытке загрузить выходной файл в симулятор, рисунок 3.5.

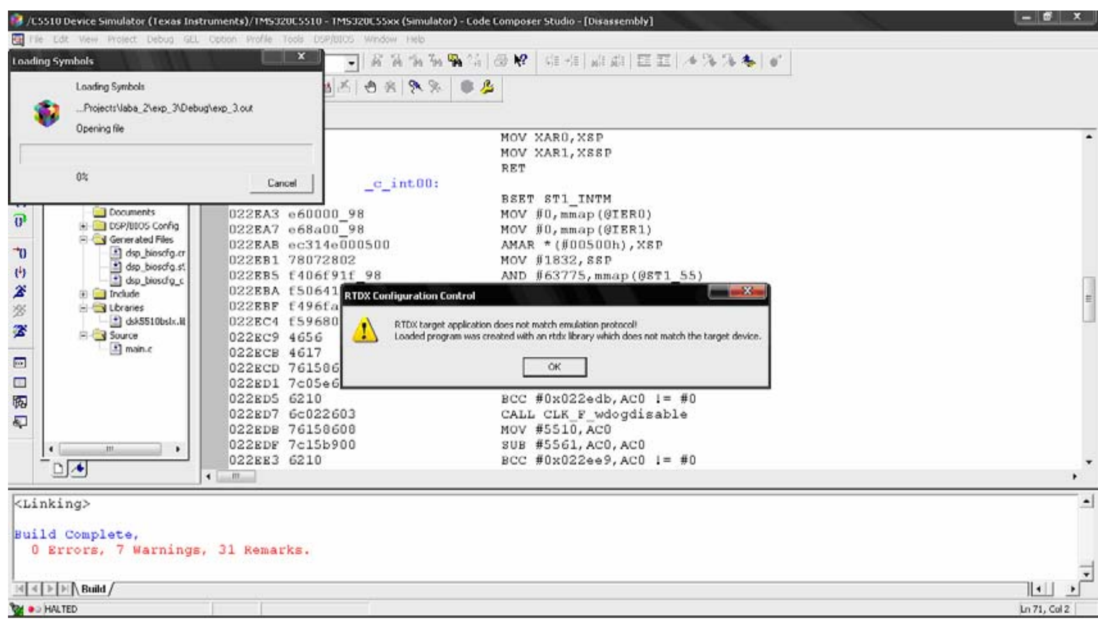


Рис. 3.5 Ошибка загрузки проекта в симулятор

Дело в том, что симулятор не поддерживает тех библиотек, с которыми собран проект, поэтому необходимо выполнять тестирования данной программы на аппаратном уровне, т. е. непосредственно на плате DSK5510.

Однако ИСР CCS не выявило ошибок, поэтому для аппаратной реализации достаточно перенастроить ИСР CCS, снова открыть проект и заново выполнить компиляцию.

Перед тем как запустить программу на выполнение на плате DSK5510 необходимо подключить к линейному выходу наушники или колонки. Чтобы определить, куда именно вставлять разъем следует прочитать надписи на плате возле линейных входов/выходов.

Следует отметить, что звук, который воспроизводит DSK5510, не совпадает с теми частотами, которые задаются программно, также присутствует характерный шум. Это связано с тем, что время работы функции «sinewave()» превышает временной интервал между отсчетами на частоте дискретизации в 48 КГц (это частота задается по умолчанию). Поэтому функция «DSK5510_AIC23_write16» вынуждена ожидать, пока функция «sinewave()» рассчитает текущий отсчет. Именно это и вносит искажение в выходной сигнал. Однако, этот момент не принципиален, т. к. предложенный генератор звуковой частоты находит свое применение при тестировании цифровых фильтров.

Список функции кодека AIC23

1. ***DSK5510_AIC23_rset****(DSK5510_AIC23_CodecHandle hCodec, Uint16 regnum, Uint16 regval)* – процедура при помощи которой можно установить значение необходимого регистра, задав его название и значение целым шестнадцати битным числом;

2. Функция - *Uint16* ***DSK5510_AIC23_rget****(DSK5510_AIC23_CodecHandle hCodec, Uint16 regnum)* возвращает значение регистра по его названию;

3. Функция - ***DSK5510_AIC23_openCodec****(****int*** *id, DSK5510_AIC23_Config *Config)* типа *DSK5510_AIC23_CodecHandle* обозначает открытие кодека с номером и возвращает указатель открытия;

4. ***void*** ***DSK5510_AIC23_closeCodec****(DSK5510_AIC23_CodecHandle hCodec)* - процедура закрытия кодека по указателю;

5. ***void*** ***DSK5510_AIC23_config****(DSK5510_AIC23_CodecHandle hCodec, DSK5510_AIC23_Config *Config)* – процедура применения настроек регистров обязательных для корректной работы кодека;

6. *CSLBool* ***DSK5510_AIC23_write16****(DSK5510_AIC23_CodecHandle hCodec, Int16 val)* – функция записи в регистр вывода шестнадцати битной величины, возвращает булевское значение (true - false);

7. *CSLBool* ***DSK5510_AIC23_write32****(DSK5510_AIC23_CodecHandle hCodec, Int32 val)* – функция записи в регистр вывода тридцати двух битной величины, возвращает булевское значение (true - false);

8. *CSLBool* ***DSK5510_AIC23_read16****(DSK5510_AIC23_CodecHandle hCodec, Int16 *val)*- читает из регистра ввода информацию в шестнадцати разрядную величину , возвращает булевское значение (true - false);

9. *CSLBool* ***DSK5510_AIC23_read32****(DSK5510_AIC23_CodecHandle hCodec, Int32 *val)*- читает из регистра ввода информацию в тридцати двух разрядную величину , возвращает булевское значение (true - false);

10. ***void*** ***DSK5510_AIC23_outGain****(DSK5510_AIC23_CodecHandle hCodec, Uint16 outGain)* – процедура устанавливающая увеличение значение выходных величин (усиление);

11. ***void*** ***DSK5510_AIC23_loopback****(DSK5510_AIC23_CodecHandle hCodec, CSLBool mode)* – устанавливает состояние перемычки;

12. ***void*** ***DSK5510_AIC23_mute****(DSK5510_AIC23_CodecHandle hCodec, CSLBool mode)*- отключает или включает звук;

13. ***void*** ***DSK5510_AIC23_powerDown****(DSK5510_AIC23_CodecHandle hCodec, Uint16 sect)* – включает или отключает аттенюатор аналого-цифрового и цифро-аналогового преобразователя;

14. ***void*** ***DSK5510_AIC23_setFreq****(DSK5510_AIC23_CodecHandle hCodec, Uint32 freq)*- устанавливает величину частоты дискретизации.

Индивидуальные задания студентам выдаются во время лабораторной работы преподавателем.

Лабораторная работа 3

ЦИФРОВАЯ ФИЛЬТРАЦИЯ. РЕАЛИЗАЦИЯ ФИЛЬТРА С КОНЕЧНОЙ ИМПУЛЬСНОЙ ХАРАКТЕРИСТИКОЙ (КИХ)

1. Цифровая фильтрация

Цифровая фильтрация является одним из наиболее мощных инструментальных средств ЦОС. Кроме очевидных преимуществ устранения ошибок в фильтре, связанных с флуктуациями параметров пассивных компонентов во времени и по температуре, дрейфом ОУ (в активных фильтрах) и т. д., цифровые фильтры способны удовлетворять таким техническим требованиям по своим параметрам, которых, в лучшем случае, было бы чрезвычайно трудно или даже невозможно достичь в аналоговом исполнении. Кроме того, характеристики цифрового фильтра могут быть легко изменены программно. Поэтому они широко используются в телекоммуникациях, в приложениях адаптивной фильтрации, таких как подавление эха в модемах, подавление шума и распознавание речи.

Процесс проектирования цифровых фильтров состоит из тех же этапов, что и процесс проектирования аналоговых фильтров. Сначала формулируются требования к желаемым характеристикам фильтра, по которым затем рассчитываются параметры фильтра. Амплитудная и фазовая характеристики формируются аналогично аналоговым фильтрам. Ключевое различие между аналоговым и цифровым фильтрами заключается в том, что, вместо вычисления величин сопротивлений, емкостей и индуктивностей для аналогового фильтра, рассчитываются значения коэффициентов для

цифрового фильтра. Иными словами, в цифровом фильтре числа заменяют физические сопротивления и емкости аналогового фильтра. Эти числа являются коэффициентами фильтра, они постоянно находятся в памяти и используются для обработки (фильтрации) дискретных данных, поступающих от АЦП.

Цифровой фильтр, работающий в реальном масштабе времени, оперирует с дискретными по времени данными в противоположность непрерывному сигналу, обрабатываемому аналоговым фильтром. При этом очередной отсчет, соответствующий отклику фильтра, формируется по окончании каждого периода дискретизации. Вследствие дискретной природы обрабатываемого сигнала, на отсчеты данных зачастую ссылаются по их номерам, например, отсчет 1, отсчет 2, отсчет 3 и т. д. На рисунке 1.1 представлен низкочастотный сигнал, содержащий высокочастотный шум, который должен быть отфильтрован. Вначале сигнал должен быть оцифрован с помощью АЦП для получения выборки x(n). Далее эта выборка поступает на цифровой фильтр, который в данном случае является НЧ фильтром. Отсчеты выходных данных y(n) используются для восстановления аналогового сигнала с использованием ЦАП с низким уровнем ложного сигнала.

Тем не менее, цифровые фильтры не могут являться решением всех возможных задач фильтрации, возникающих при обработке сигналов. Для работы в реальном масштабе времени, DSP-процессор должен быть рассчитан на выполнение всех шагов в программе фильтрации в пределах промежутка времени, соответствующего одному такту дискретизации, т. е. 1/fs. Высоко производительный универсальный DSP процессор с фиксированной точкой типа ADSP-2189M, обладающий быстродействием 75MIPS, способен выполнить операцию умножения с накоплением при реализации одного каскада фильтра за 13,3 нс. DSP процессор ADSP-2189M затрачивает N+5 инструкций при реализации фильтра с количеством каскадов N. Для 100-каскадного фильтра полное время вычисления составляет приблизительно 1,4 мкс. Это соответствует максимально возможной частоте дискретизации 714 кГц, ограничивая, таким образом, ширину полосы частот обрабатываемого сигнала несколькими сотнями килогерц.

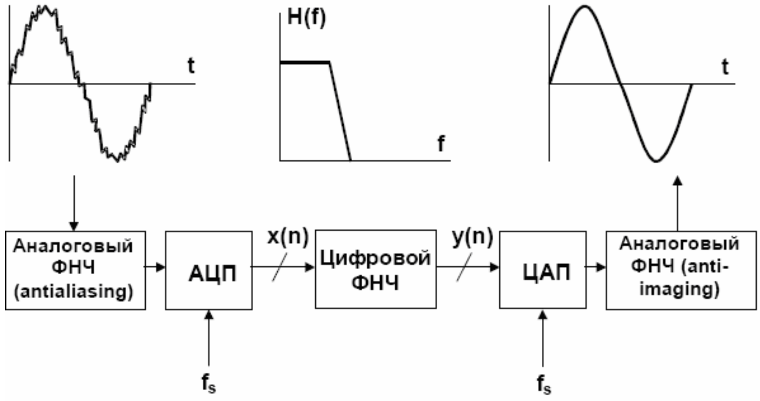


Рис. 1.1 Структура цифрового фильтра

Можно заменить универсальный DSP процессор специализированным аппаратным цифровым фильтром, способным работать на частотах дискретизации, соответствующих видеосигналу. В других случаях ограничения по быстродействию могут быть преодолены сохранением выборки данных, поступающих с большой скоростью от АЦП, в буферной памяти. Затем буферная память читается со скоростью, совместимой с быстродействием цифрового фильтра, основанного на DSP. Используя данный метод, может осуществляться обработка сигнала в псевдореальном масштабе времени в таких системах как радар, где обычно обрабатываются пакеты данных, накапливаемые после каждого излучаемого импульса.

Другой подход заключается в использовании специализированных микросхем цифровых фильтров, подобных фильтрам PulseDSP ™ компании Systolix. 16 разрядный сигма-дельта-АЦП AD7725 имеет на своем кристалле фильтр PulseDSP, который может выполнять за секунду 125 миллионов операций умножения с накоплением.

В дискретных системах, даже с высокой степенью избыточной дискретизации, требуется наличие аналоговых ФНЧ перед АЦП и после ЦАП для устранения эффекта наложения спектра. Более того, с ростом частоты, сигналы выходят за рамки возможностей доступных АЦП, и цифровая фильтрация становится невозможной. Но на крайне высоких частотах и активная аналоговая фильтрация тоже невозможна из-за ограничений, связанных с полосой пропускания и искажениями ОУ, и в этих случаях требования фильтрации удовлетворяются пассивными элементами. Дальнейшее обсуждение будет сфокусировано, в первую очередь, на фильтрах, которые могут работать в реальном масштабе времени и могут быть программно реализованы с использованием DSP.

В качестве примера сравним аналоговый и цифровой фильтры, показанные на рис. 1.2. Частота среза обоих фильтров равна 1 кГц. Аналоговый фильтр реализован в виде фильтра Чебышева первого рода 6 порядка (характеризуется неравномерностью коэффициента передачи в полосе пропускания и равномерностью коэффициента передачи вне полосы пропускания). На практике этот фильтр может быть собран на трех фильтрах второго порядка, каждый из которых состоит из операционного усилителя и нескольких резисторов и конденсаторов. Проектирование фильтра 6 порядка является непростой задачей, а удовлетворение техническим требованиям по неравномерности характеристики в 0,5 дБ требует точного подбора компонентов.

С другой стороны, представленный цифровой фильтр с конечной импульсной характеристикой (КИХ) имеет неравномерность характеристики всего 0,002 дБ в полосе пропускания, линейную фазовую характеристику и значительно более крутой спад частотной характеристики. Таких показателей невозможно достичь аналоговыми методами. На практике су-

ществует много других факторов, учитываемых при сравнительной оценке аналоговых и цифровых фильтров. В большинстве современных систем обработки сигналов используются комбинации аналоговых и цифровых методов для реализации желаемых функций и используются преимущества всех методов, как аналоговых, так и цифровых.

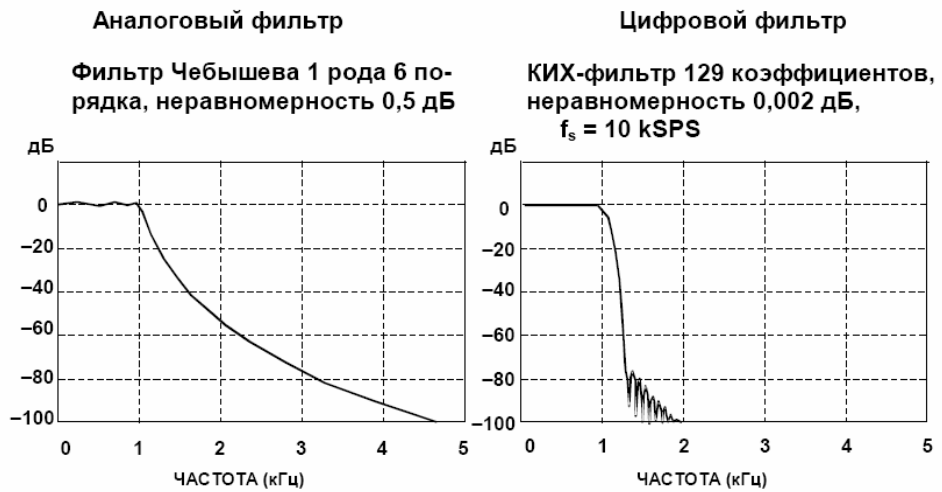


Рис. 1.2 Сравнение АЧХ цифрового и аналогового фильтров

Таблица 1.1 – Сравнение характеристик цифровых и аналоговых фильтров.

Цифровые фильтры	Аналоговые фильтры
Высокая точность	Низкая точность из-за допуска на элементы
Линейная фаза (КИХ-фильтр)	Нелинейная фаза
Нет дрейфа вследствие изменения параметров компонентов	Дрейф вследствие изменения параметров компонентов
Гибкость, возможна адаптивная фильтрация	Реализация адаптивных фильтров затруднена
Легки в моделировании и проектировании	Сложны в моделировании и проектировании
Ограничения при работе в реальном масштабе времени – вычисление должно быть завершено в течение интервала дискретизации	Аналоговые фильтры требуются на высоких частотах и для устранения эффекта наложения спектра

Существует много приложений, в которых цифровые фильтры должны работать в реальном масштабе времени. В них накладываются определенные требования на процессор DSP в зависимости от частоты дискретизации и сложности фильтра. Ключевым моментом является то, что процессор DSP должен проводить все вычисления в течение интервала дискретизации, чтобы быть готовым к обработке следующего отсчета данных. Пусть ширина полосы частот обрабатываемого сигнала равна fa. Тогда частота дискретизации АЦП fs должна быть, по крайней мере, в два раза больше, т. е. 2fa. Интервал дискретизации равен 1/fs. Все вычисления, свя-

занные с реализацией фильтра (включая все дополнительные операции), должны быть закончены в течение этого интервала. Время вычислений зависит от числа звеньев фильтра и быстродействия и эффективности процессора DSP. Каждое звено при реализации фильтра требует одной операции умножения и одной операции сложения (умножения с накоплением). Процессор DSP оптимизируется для быстрого выполнения операций умножения с накоплением. Кроме того, многие процессоры DSP имеют дополнительные особенности, такие как реализация циклической адресации и организация программных циклов с автоматической проверкой условия продолжения цикла, минимизирующие количество дополнительных инструкций, которые в противном случае были бы необходимы.

Требования к цифровой фильтрации для работы в реальном масштабе времени:

1) полоса сигнала = fa;
2) частота дискретизации fs > 2fa;
3) период дискретизации =1/fs;
4) время вычисления фильтра + доп. операции < период дискр;
5) зависит от числа коэффициентов фильтра;
6) скорости операций умножения с накоплением (MAC);
7) эффективности ЦОС;
8) поддержка циклических буферов;
9) отсутствие дополнительных операций.

Фильтры с конечной импульсной характеристикой (КИХ)

Существует два основных типа цифровых фильтров: фильтры с конечной импульсной характеристикой (КИХ) и фильтры с бесконечной импульсной характеристикой (БИХ). Как следует из терминологии, эта классификация относится к импульсным характеристикам фильтров. Изменяя веса коэффициентов и число звеньев КИХ-фильтра, можно реализовать практически любую частотную характеристику. КИХ-фильтры могут иметь такие свойства, которые невозможно достичь методами аналоговой фильтрации (в частности, совершенную линейную фазовую характеристику). Но высокоэффективные КИХ-фильтры строятся с большим числом операций умножения с накоплением и поэтому требуют использования быстрых и эффективных процессоров DSP. С другой стороны, БИХ-фильтры имеют тенденцию имитировать принцип действия традиционных аналоговых фильтров с обратной связью. Поэтому их импульсная характеристика имеет бесконечную длительность. Благодаря использованию обратной связи, БИХ-фильтры могут быть реализованы с меньшим количеством коэффициентов, чем КИХ-фильтры. Другим способом реализации

КИХ-или БИХ-фильтрации являются решетчатые фильтры, которые часто используются в задачах обработки речи. Цифровые фильтры применяются в приложениях адаптивной фильтрации, благодаря своему быстродействию и простоте изменения характеристик воздействием на его коэффициенты.

Типы цифровых фильтров:

1. Фильтр скользящего среднего.
2. Фильтр с конечной импульсной характеристикой (КИХ):
   a) линейная фаза;
   b) легкость проектирования;
   c) значительные вычислительные затраты.
3. Фильтр с бесконечной импульсной характеристикой:
   a) основаны на классических аналоговых фильтрах;
   b) высокая вычислительная эффективность.
4. Решетчатые фильтры (могут быть КИХ или БИХ).
5. Адаптивные фильтры.

Элементарной формой КИХ-фильтра является фильтр скользящего среднего (moving average), показанный на рис. 1.3. Фильтры скользящего среднего популярны для сглаживания данных, например, для анализа стоимости акций и т. д. Входные отсчеты x(n) пропускаются через ряд регистров памяти (помеченных z^{-1} в соответствии с представлением элемента задержки при z-преобразовании). В приведенном примере имеется четыре каскада, соответствующих 4-точечному фильтру скользящего среднего. Каждый отсчет умножается на 0,25, и результаты умножения суммируются для получения значения скользящего среднего, которое подается на выход y(n). На рисунке также представлено общее уравнение фильтра скользящего среднего на N точек. N относится к числу точек при вычислении фильтра, а не к разрешающей способности АЦП или ЦАП.

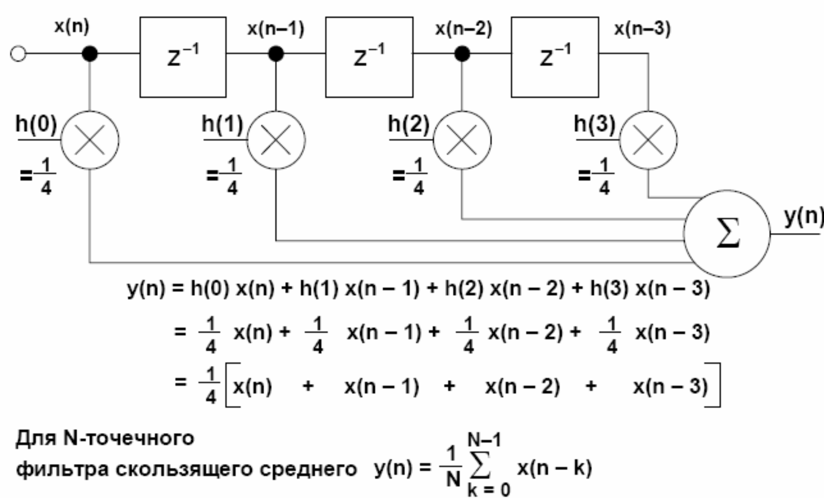


Рис. 1.3 4-х точечный фильтр скользящего среднего

С учетом равенства коэффициентов, наиболее простой путь исполнения фильтра скользящего среднего представлен на рис. 1.4. Обратите внимание, что первым шагом является запоминание первых четырех отсчетов x(0), x(1), x(2), x(3) в регистрах. Эти величины суммируются и затем умножаются на 0,25 для получения первого выхода y(3). Обратите внимание, что начальные значения выходов y(0), y(1) и y(2) некорректны, потому что, пока отсчет x(3) не получен, не все регистры заполнены.

$$y(3) = 0.25\left[x(3) + x(2) + x(1) + x(0)\right]$$

$$y(4) = 0.25\left[x(4) + x(3) + x(2) + x(1)\right]$$

$$y(5) = 0.25\left[x(5) + x(4) + x(3) + x(2)\right]$$

$$y(6) = 0.25\left[x(6) + x(5) + x(4) + x(3)\right]$$

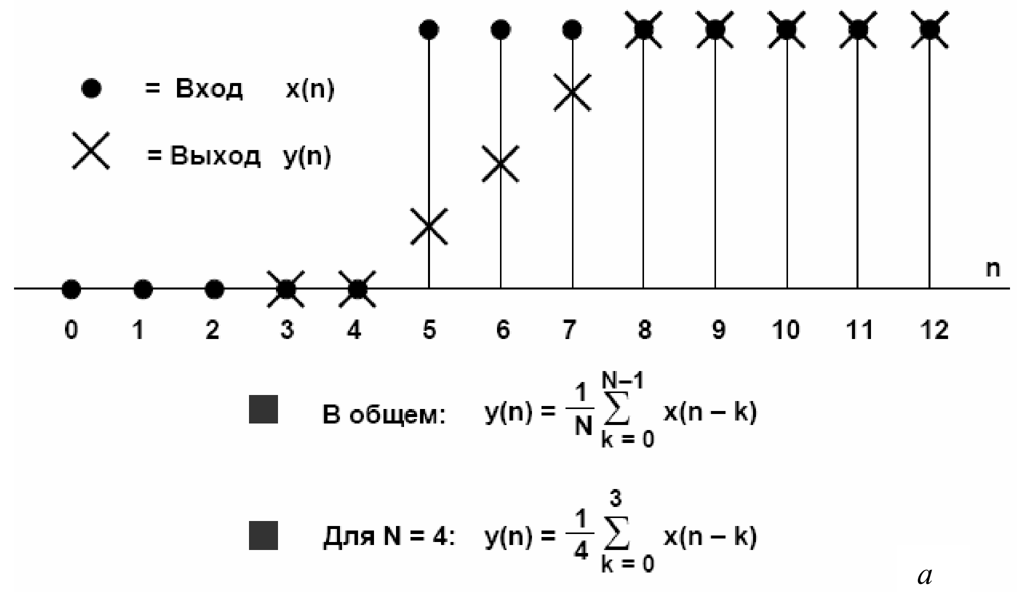
$$y(7) = 0.25\left[x(7) + x(6) + x(5) + x(4)\right]$$

-
- **Вычисление каждого выходного значения требует 1 умножения, 1 сложения и 1 вычитания**
-

Рис. 1.4 Вычисление выходного сигнала фильтра скользящего среднего

Когда получен отсчет x(4), он суммируется с результатом, а отсчет x(0) вычитается из результата. Затем новый результат должен быть умножен на 0,25. Поэтому вычисления, требуемые для получения нового значения на выходе, состоят из одного суммирования, одного вычитания и одного умножения, независимо от длины фильтра скользящего среднего.

Реакция 4-точечного фильтра скользящего среднего на ступенчатое воздействие представлена на рис. 1.5 (а). Реакция фильтра скользящего среднего на воздействие в виде смеси шума и ступенчатого сигнала представлена на рис. 1.5 (б). Фильтр скользящего среднего не имеет выброса по фронту входного сигнала. Это делает его полезным в приложениях обработки сигналов, где требуется фильтрация случайного белого шума при сохранении характера входного импульса. Из всех возможных линейных фильтров, фильтр скользящего среднего дает самый низкий уровень шума при заданной крутизне фронта импульса. Это показано на рис.1.6, где уровень шума понижается по мере увеличения числа точек. Существенно, что время реакции фильтра на ступенчатое воздействие от 0 % до 100 % равно произведению общего количества точек фильтра на период дискретизации.



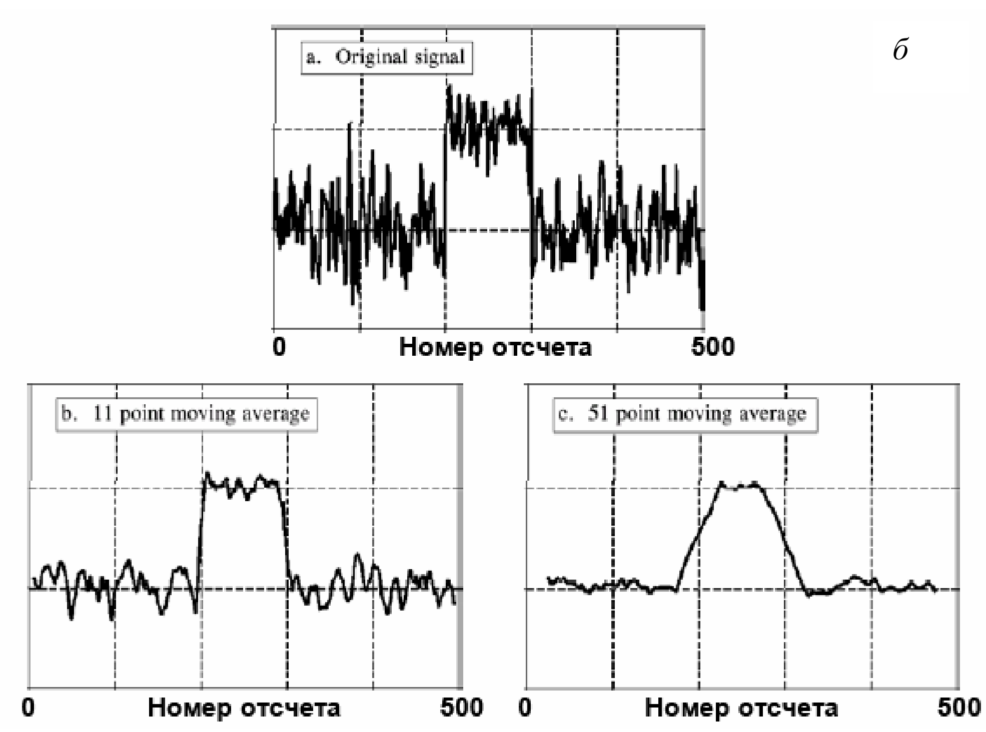


Рис. 1.5. *а* – реакция фильтра скользящего среднего на ступенчатое воздействие;
б – реакция фильтра скользящего среднего на воздействие
в виде смеси шума и ступенчатого сигнала

Частотная характеристика простого фильтра скользящего среднего выражается функцией sin(x)/x. Она представлена в линейном масштабе на рис. 1.6. Увеличение числа точек при реализации фильтра сужает основной лепесток, но существенно не уменьшает амплитуду боковых лепестков частотной характеристики, которая равна приблизительно -14 дБ для

фильтра с 11 и с 31 отводами (длиной буфера). Естественно, эти фильтры не подходят в том случае, где требуется большое ослабление в полосе задержания.

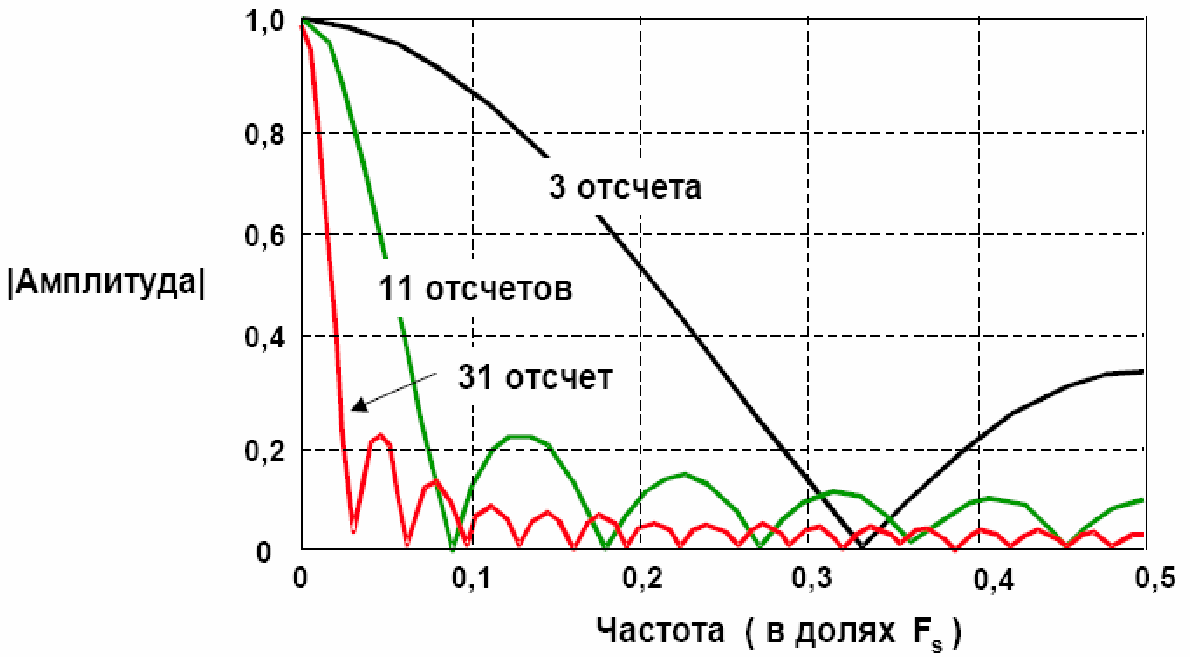


Рис. 1.6 АЧХ фильтра скользящего среднего

Можно существенно улучшить эффективность простого КИХ-фильтра скользящего среднего, выбирая разные веса или значения коэффициентов вместо равных значений. Крутизна спада может быть увеличена добавлением большего количества звеньев в фильтр, а характеристики полосы затухания улучшаются выбором надлежащих коэффициентов фильтра. Обратите внимание, что, в отличие от фильтра скользящего среднего, для реализации каждой ступени обобщенного КИХ-фильтра требуется цикл умножения с накоплением. Сущность проектирования КИХ-фильтра сводится к выбору соответствующих коэффициентов и необходимого числа звеньев при формировании желаемой частотной характеристики фильтра H(f). Для включения необходимой частотной характеристики H(f) в набор КИХ-коэффициентов имеются различные алгоритмы и программные пакеты. Большинство этого программного обеспечения разработано для персональных компьютеров и доступно на рынке. Ключевой теоремой проектирования КИХ-фильтра является утверждение, что коэффициенты h(n) КИХ-фильтра являются просто квантованными значениями импульсной характеристики этого фильтра. Соответственно, импульсная характеристика является дискретным преобразованием Фурье от H(f).

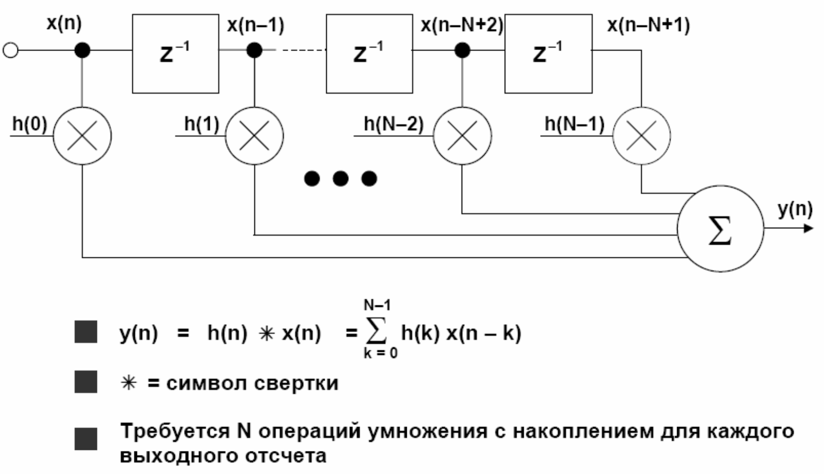


Рис. 1.7 Фильтр с конечной импульсной характеристикой порядка N

Обобщенная форма КИХ-фильтра с числом звеньев N представлена на рис. 1.7. Как было сказано, КИХ-фильтр должен работать в соответствии с уравнением, задающим свертку:

$$\mathrm{Y\ (n) = h\ (k)\ {*}x\ (n)} = \sum_{k=0}^{N-1} h(k)x(n-k),$$

где h(k) – массив коэффициентов фильтра и x(n-k) – входной массив данных фильтра. Число N в уравнении представляет собой число звеньев и определяет эффективность фильтра, как было сказано выше. КИХ-фильтр с числом звеньев N требует N циклов (операций) умножения с накоплением.

Согласно рис. 1.8, диаграммы КИХ-фильтров часто изображаются в упрощенном виде. Операции суммирования представляются стрелками, указывающими в точки, а операции умножения обозначают, помещая коэффициенты h(k) рядом со стрелками на линиях. Элемент задержки z^{-1} показывают, помещая его обозначение выше или рядом с соответствующей линией.

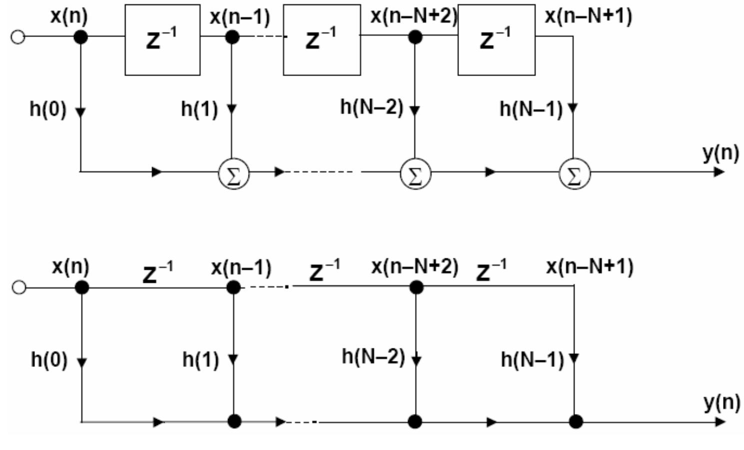


Рис. 1.8 Упрощенная схема фильтра

Реализация КИХ-фильтра на процессоре DSP с использованием циклических буферов

В рядах, задаваемых уравнениями КИХ-фильтров, предполагается последовательное обращение к N коэффициентам от h(0) до h(N-1). Соответствующие точки данных циркулируют в памяти. При этом добавляются новые отсчеты данных, заменяя самые старые, и каждый раз производится вычисление выходного значения фильтра. Для реализации циклического буфера может использоваться фиксированный объем оперативной памяти, как показано на рис. 1.9 для КИХ-фильтра с 4 звеньями. Самый старый отсчет данных заменяется новым после каждой операции вычисления свертки. Выборка из четырех последних отсчетов данных всегда сохраняется в оперативной памяти.

Чтобы упростить адресацию, чтение из памяти старых значений начинается с адреса, который следует непосредственно за адресом только что записанного нового элемента выборки. Например, если значение x(4) только что записано в ячейку памяти 0, то значения данных читаются из ячеек 1, 2, 3 и 0. Этот пример может быть расширен применительно к любому числу звеньев фильтра. Используя адресацию ячеек памяти таким способом, адресный генератор должен лишь вычислять последовательные адреса, независимо от того, является ли данная операция чтением памяти или записью. Такой буфер в памяти данных называется циклическим, потому что, когда достигается его последняя ячейка, указатель автоматически позиционируется на начало буфера.

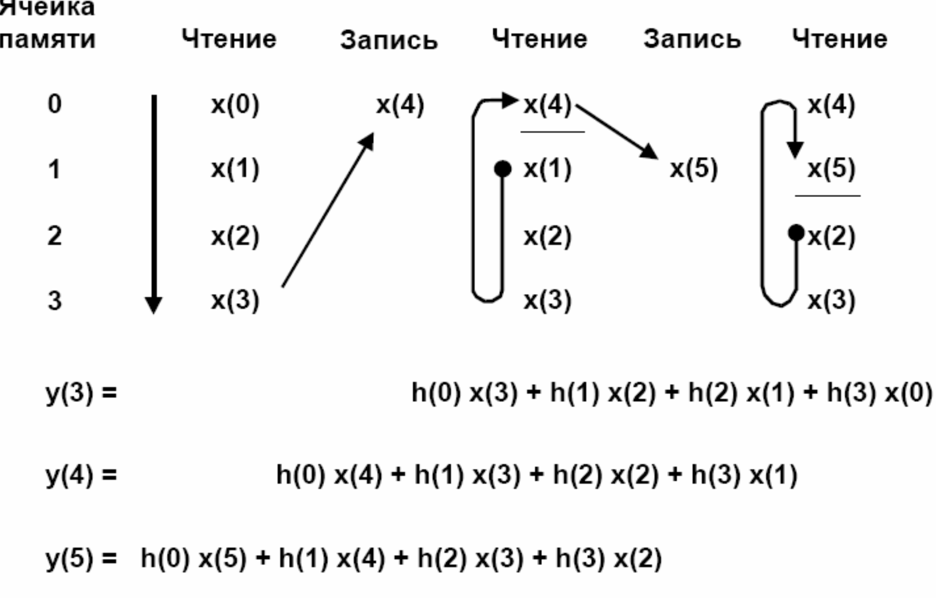


Рис. 1.9 Вычисление выходного сигнала КИХ-фильтра 4-ого порядка
с использованием циклического буфера

Выборка коэффициентов из памяти осуществляется одновременно с выборкой данных. В соответствии с описанной схемой адресации, самый старый отсчет данных выбирается первым. Поэтому сначала должна осуществляться выборка из памяти последнего коэффициента. При использовании адресного генератора, поддерживающего инкрементную адресацию, коэффициенты могут быть сохранены в памяти в обратном порядке: h(N-1) помещается в первую ячейку, а h(0) – в последнюю. И наоборот, коэффициенты могут быть сохранены в порядке возрастания их номеров, если использовать адресный генератор, поддерживающий декрементную адресацию. В примере, показанном на рис. 1.9, коэффициенты сохранены в обратном порядке.

Простая итоговая блок-схема для этих операций представлена на рисунке 1.10. Все операции, выполняемые за один цикл фильтра, производятся за один командный цикл процессора, благодаря чему существенно увеличивается эффективность вычислений. Данное преимущество известно как реализация циклов без дополнительных операций.

Ограничение на число звеньев фильтра, реализующего подпрограммы КИХ-фильтрации в реальном масштабе времени, определяется, прежде всего, длительностью процессорного цикла, частотой дискретизации и требуемым объемом других вычислений.

1. Получение отсчета от АЦП (обычно по прерыванию)
2. Помещение отсчета в циклический буфер входного сигнала
3. Обновление указателя циклического буфера входного сигнала
4. Обнуление аккумулятора
5. Осуществление фильтрации (цикл по всем коэффициентам)
6. Выборка коэффициента из циклического буфера коэффициентов
7. Обновление указателя циклического буфера коэффициентов
8. Выборка отсчета из циклического буфера входного сигнала
9. Обновление указателя циклического буфера входного сигнала
10. Умножение коэффициента на отсчет
11. Добавление нового слагаемого к промежуточному результату
12. Выдача отфильтрованного отсчета на ЦАП

Рис. 1.10 Псевдокод программы фильтра, использующей DSP процессор с циклической буферизацией

2. Реализация КИХ-фильтра на симуляторе DSK5510

Программная реализация

1. Создать проект под названием «exp_1», в предварительно созданной папке «laba_3».

2. Создать и добавить к проекту файл «filter.h», в соответствии с листингом 2.1. Исходный код разработан в соответствии с алгоритмом, рассмотренным в разделе 1, и снабжен построчным комментарием, поэтому рассмотрен не будет.

Листинг 2.1 – Листинг файла «filter.h»

```
#include <stdio.h>

#define LENBUF    120
#define LENFILTER  16

//входной буфер
int inp_buf[LENBUF];
//коэффициенты фильтра умноженные на число 512
int coeffBuff[]={0,0,2,8,23,50,84,108,108,84,50,23,8,2,0,0};
//выходной буфер
int out_buf[LENBUF];
//буфер линии задержки
int SimplBuff[LENFILTER];
//вспомогательная переменная
int coeff;
//счетчики
int count;
int count_run;
int count_con;
//функция запуска фильтрации
int run_filter()
{
 for(count_run=0;count_run<LENBUF;count_run++)
            {
            //чтение входного отсчета
            coeff=inp_buf[count_run];
            //запись текущего отсчета в линию задержки
            SimplBuff[0]=coeff;
            //запуск функции свертки сигнала
            coeff=convolution();
            //запись в выходной буфер отфильтрованного сигнала
            out_buf[count_run]=coeff;
            }
 return 0;
}
//функция осуществляющая свертку сигнала
int convolution()
{
 //вспомогательные переменные
 long int coeff_mx;
 long int summ;
 long int tmp1,tmp2;

 summ=0;
 for(count_con=0;count_con<LENFILTER;count_con++)
            {
            //чтение текущего отсчета из линии задержки
            tmp1=SimplBuff[count_con];
            //чтение коэффициента фильтра
            tmp2=coeffBuff[count_con];
            //перемножение отсчета с коэффициентом фильтра
            coeff_mx=tmp1*tmp2;
```

```
            //накопление результата
            summ +=coeff_mx;
            }
 //цикл сдвига линии задержки на единицу вправо
 for(count_con=LENFILTER-1;count_con>0;count_con--)
            {
            SimplBuff[count_con]=SimplBuff[count_con-1];
            }
 //деление накопленного результат на число 512, т. к. коэффициенты
 //филтра были умножены на число 512
 summ >>=9;
 return summ;
}
//функция очистки линии задержки
void ClearSimpleBuff()
{
 for(count=0;count<LENFILTER;count++)
            SimplBuff[count]=0;
}
```

Коэффициента фильтра, рассчитанные для нормированной частоты среза 0.0773, приведены в табл. 2.1.

Таблица 2.1

Значение коэффициентов КИХ-фильтра

№	Дробное значение	Значение, умноженное на число 512	Округление
B_1	0	0	0
B_2	0.0005	0.256	0
B_3	0.0034	1.7408	2
B_4	0.0149	7.6288	8
B_5	0.0446	22.8352	23
B_6	0.0982	50.2784	50
B_7	0.1637	83.8144	84
B_8	0.2104	107.7248	108
B_9	0.2104	107.7248	108
B_{10}	0.1637	83.8144	84
B_{11}	0.0982	50.2784	50
B_{12}	0.0446	22.8352	23
B_{13}	0.0149	7.6288	8
B_{14}	0.0034	1.7408	2
B_{15}	0.0005	0.256	0
B_{16}	0	0	0

Умножение коэффициентов фильтра на число 512 и округление, вызвано тем, что операции с дробными числами (с плавающей точкой) производятся с очень большими затратами времени. Увеличение времени вычисления приводит к ухудшению качества отфильтрованного потока звука.

Константа 512 выбрана вследствие того, что это число можно представить как 29, а это означает, что накопленный результат умножения с накоплением достаточно побитно сдвинуть вправо на 9 разрядов оператором «>>=», и это будет соответствовать делению суммы на константу 512.

В файле «filter.h» реализованы следующие функции:

Таблица 2.2

Функции реализованные в файле «filter.h»

Функции	Описание
int run_filter()	Заносит входной отсчет в линию задержки и запускает функцию свертки.
int convolution()	Осуществляет непосредственно процедуру свертки.
void ClearSimpleBuff()	

3. Создать и добавить к проекту файл «main.c», в соответствии с листингом 2.2.

Листинг 2.2 – Листинг файла «main.c»

```
#include "filter.h"
//вспомогательная переменная
int len;
//указатели на входной и выходной файлы
FILE *inFile;
FILE *outFile;
//заголовок wav файла
char wavHD[44];
int main (void)
{
 //очистка линии задержки
 ClearSimpleBuff();
 //открытие входного и выходного файла
 inFile = fopen("..\\data\\coldplay_in.wav", "rb");
 outFile = fopen("..\\data\\coldplay_out.wav", "wb");
 //проверка корректности открытия выходного файла
 if(outFile == NULL)
           {
           printf("ERROR, OUT FILE\n");
           return 1;
           }
 //чтение заголовка входного файла
 fread(wavHD, sizeof(char), 44, inFile);
 //запись заголовка в выходной файл
 fwrite(wavHD, sizeof(char), 44, outFile);
  //Создание единичного импульса
 for(len=0;len<LENBUF;len++)
           inp_buf[len]=0;
 inp_buf[0]=512;
 run_filter();
/* len=LENBUF;
  while(len == LENBUF)
           {
           //считать данные из файла
           len = fread(inp_buf,1, LENBUF, inFile);
           //вызов функции обработки (фильтрации) входного буфера
           run_filter();
                  //запись данных в файл
           fwrite(out_buf, 1, len, outFile);
           } */
 return 0;
}
```

Здесь, закомментирован цикл считывания данных из входного файла «**coldplay_in.wav**», для того, чтобы построить импульсную характеристику фильтра. Для построения импульсной характеристики, входной буфер сначала обнуляется в цикле, а затем первому отсчету (нулевому) присваивается значение 512, т. к. коэффициенты умножены на число 512.

4. Все остальные файлы проекта создаются по аналогии с предыдущими работами. В таблице 2.3, приведен список файлов проекта.

Таблица 2.3

Файлы проекта

Файлы	Описание
main.c	Главный связывающий файл
lnk.cmd	Файл компоновщика
filter.h	Заголовочный файл реализующий фильтрацию
exp_1.pjt	Файл проекта
coldplay_in.wav	Входной звуковой файл, располагается в папке проекта в каталоге «**data**»
coldplay_out.wav	Выходной звуковой файл, располагается в папке проекта в каталоге «**data**»

5. Необходимо скомпилировать проект.

Тестирование КИХ-фильтра

1. Необходимо открыть окно визуализации данных для входного и выходного буфера, с настройками, которые показаны на рисунке 2.1, и минимизировать оба окна визуализации, сняв флажок с опции «Float In Main Window» в контекстном меню окна. Эти действия необходимы для построения графика входного сигнала (единичного импульса) и графика реакции КИХ-фильтра на импульсное воздействие.

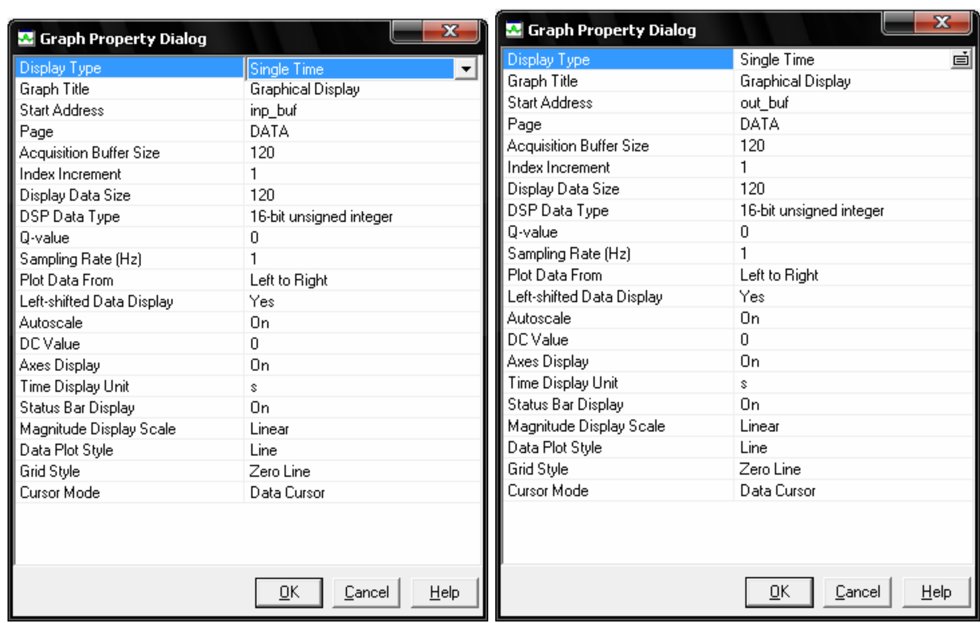


Рис. 2.1 Настройки окон визуализации

2. Следующим шагом следует запустить проект на исполнение, после завершения работы программы вид ИСР CCS станет таким, как показано на рис. 2.2.

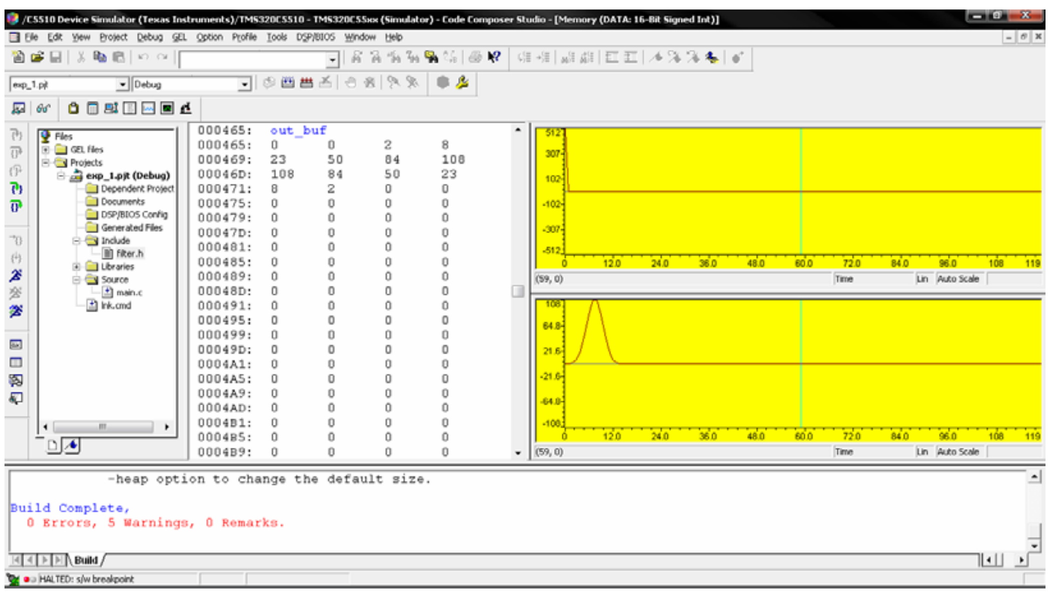


Рис. 2.2 Построение импульсной характеристики фильтра

Для просмотра содержимого памяти выделенной под выходной буфер, необходимо открыть окно просмотра содержимого памяти, и в настройках указать начальный адрес просмотра «out_buf», рис. 2.3.

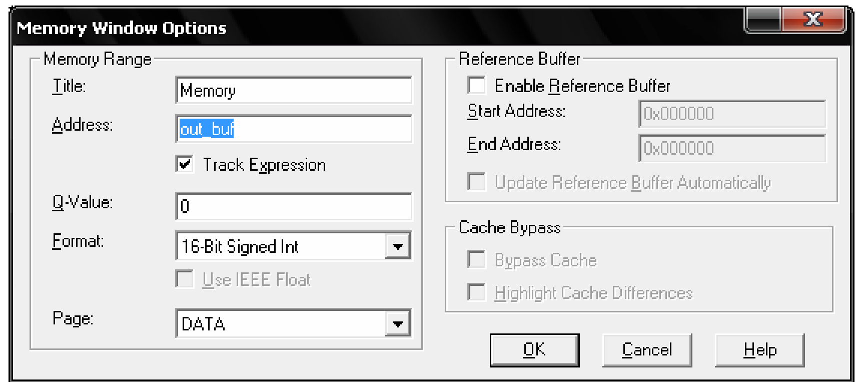


Рис. 2.3 Настройка окна просмотра содержимого памяти

Показателем корректной работы фильтра является то, что в выходном буфере содержаться коэффициенты фильтра.

3. Не закрывая окон визуализации необходимо закомментировать участок кода в файле «main.c», который отвечает за формирование единичного импульса, и раскомментировать цикл чтения данных из входного файла, фильтрацию и сохранение результата.

4. Установите точку прерывания в строке запуска функции фильтрации, файл «main.c».

5. Перекомпилируйте проект и запустите на выполнение, окно ИСР CCS примет вид как показано на рис. 2.4.

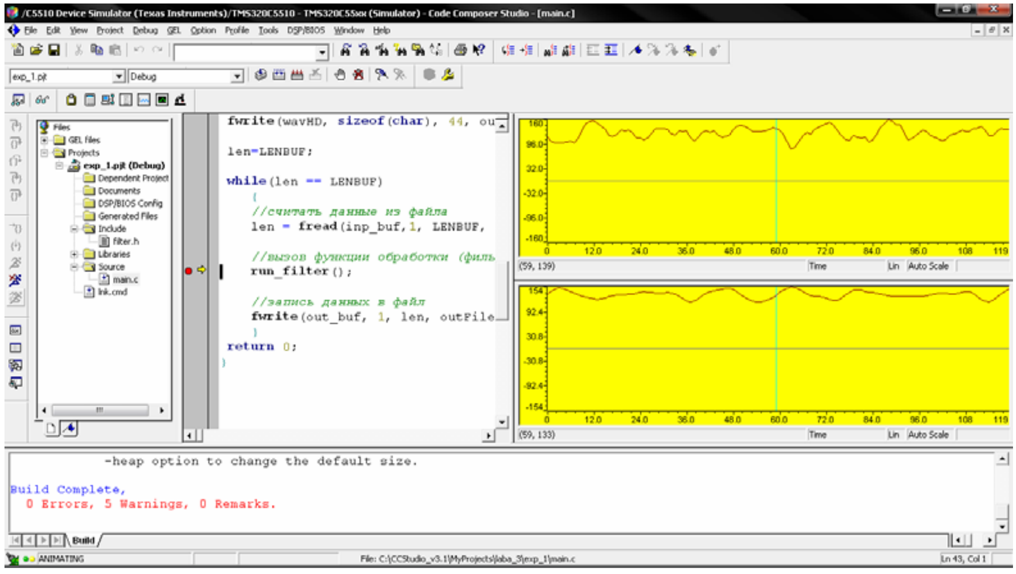


Рис. 2.4 Построение осциллограммы входного и выходного файла

Признаком успешной работы фильтра является более гладкая кривая на осциллограмме выходного сигнала, нежели входного сигнала, на рис. 2.4 выходная осциллограмма располагается снизу. Так как частота дискретизации входного файла соответствует 8000 Гц, то частота среза фильтра будет рассчитываться следующим образом: 8000*0.0773≈618.2 Гц.

6. В процессе работы программы убрать точку прерывания и дать программе закончить вычисления, на это потребуется некоторое время. В процессе работы программы размер выходного файла будет постоянно увеличиваться, а по завершении станет равным размеру входного.

7. Запустить приложение «Общая громкость (Регулятор громкости)», которое располагается: «Пуск→Стандартные→Развлечения». Выбрать раздел «Параметры→Свойства», рис. 2.5.

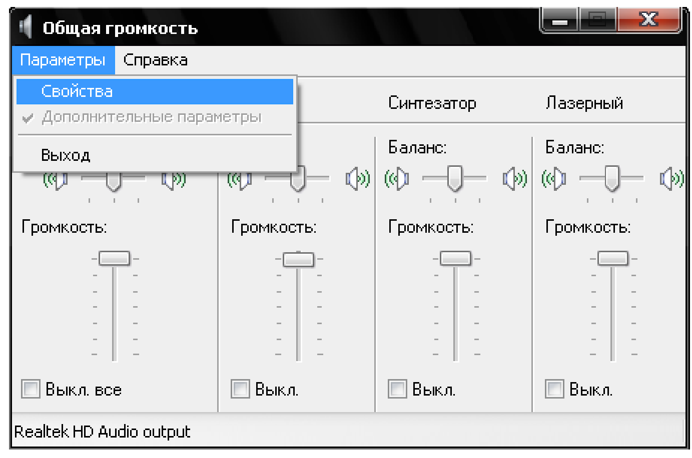


Рис. 2.5 Регулятор громкости звука системы Windows

8. В появившемся окне в разделе «Микшер» нужно выбрать устройство ввода звука (на рис. 2.6 это устройство «Realtek HD Audio Input») и нажать кнопку «ОК».

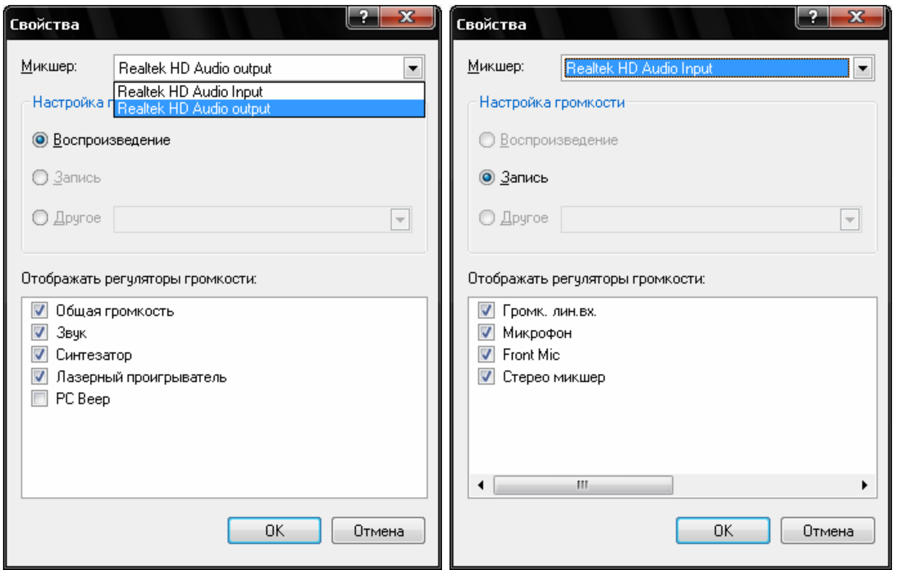


Рис. 2.6 Свойства регулятора громкости

9. Окно регулятора громкости станет таким как показано на рис. 2.7.

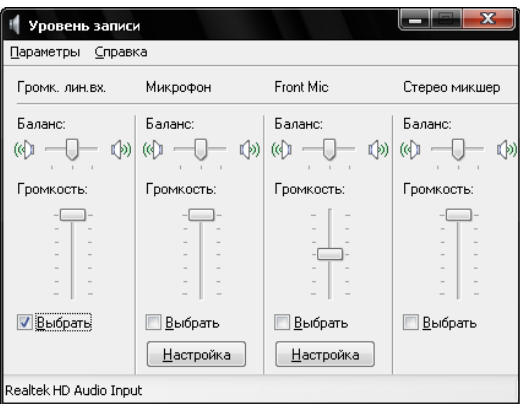


Рис. 2.7 Регулятор громкости для входного звукового сигнала

Теперь нужно выбрать «**громкость линейного входа**», установив галочку. После этого система Windows будет воспринимать входной звуковой сигнал с линейного входа звуковой карты.

10. В линейный вход звуковой карты вставить разветвитель, аналогичный тому, что показан на рис. 2.8. Как правило, линейный вход на звуковой карте голубого цвета.

Рис. 2.8 Звуковой разветвитель

11. Звуковым кабелем «папа-папа» соединить одно гнездо разветвителя с линейным выходом, а в оставшееся свободное гнездо разветвителя вставить колонки (наушники).

12. Запустить программу «**TES2.EXE**», с помощью которой будет строиться спектрограмма, рис. 2.9. Эта программа анализирует входной звуковой сигнал, в данном случае сигнал с линейного входа.

Рис. 2.9 Внешний вид программы TES2

13. Теперь нужно нажать кнопку и воспроизвести любым проигрывателем входной звуковой фай «**coldplay_in.wav**». Вид программы TES2 будет аналогичен тому, что показан на рисунке 2.10.

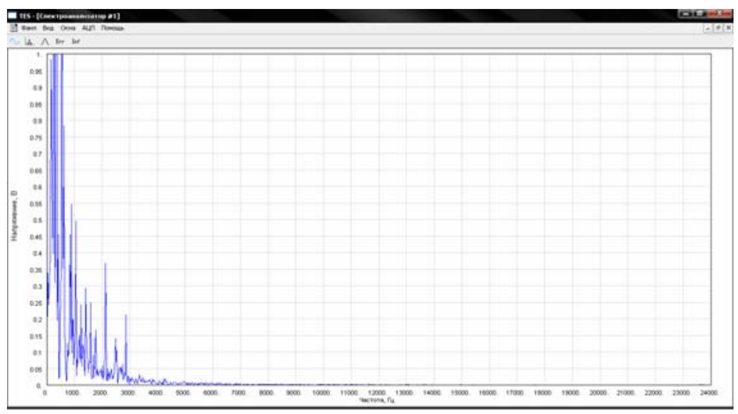


Рис. 2.10 Спектр входного звукового файла

14. Тем же проигрывателем воспроизвести выходной файл, рис. 2.11.

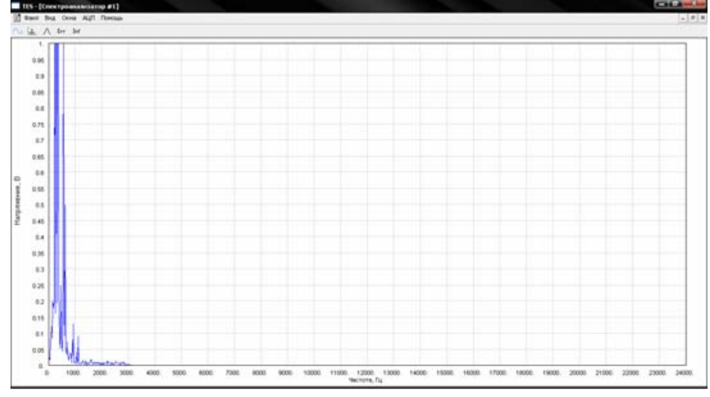


Рис. 2.11 Спектр выходного звукового файла

Из рисунка отчетливо видна частота среза, которая соответствует заданным характеристикам КИХ-фильтра.

3. Реализация КИХ-фильтра на DSK5510 для фильтрации звукового сигнала в реальном времени

1. Сконфигурировать ИСР CCS для работы с платой DSK5510.
2. Создать проект «exp_2» аналогичный проекту «exp_3» из работы №2.
3. В файл «main.c» внести изменения в соответствии с листингом 3.1.

Листинг 3.1 – Листинг файла «main.c»

```
#include "bios_filtrcfg.h"
#include "filter.h"

#include "dsk5510.h"
#include "dsk5510_aic23.h"

/* Codec configuration settings */
DSK5510_AIC23_Config config = {
    0x0017, // 0 DSK5510_AIC23_LEFTINVOL  Left line input channel volume
    0x0017, // 1 DSK5510_AIC23_RIGHTINVOL Right line input channel volume
    0x00d8, // 2 DSK5510_AIC23_LEFTHPVOL  Left channel headphone volume
    0x00d8, // 3 DSK5510_AIC23_RIGHTHPVOL Right channel headphone volume
    0x0011, // 4 DSK5510_AIC23_ANAPATH    Analog audio path control
    0x0000, // 5 DSK5510_AIC23_DIGPATH    Digital audio path control
    0x0000, // 6 DSK5510_AIC23_POWERDOWN  Power down control
    0x0043, // 7 DSK5510_AIC23_DIGIF      Digital audio interface format
    0x0081, // 8 DSK5510_AIC23_SAMPLERATE Sample rate control
    0x0001  // 9 DSK5510_AIC23_DIGACT     Digital interface activation
};

void main()
{
 DSK5510_AIC23_CodecHandle hCodec;
 Int16 *x;

 DSK5510_init();
 hCodec = DSK5510_AIC23_openCodec(0, &config);

 DSK5510_AIC23_setFreq(hCodec,DSK5510_AIC23_FREQ_44KHZ);

 ClearSimpleBuff();

 while(1)
            {
            while (!DSK5510_AIC23_read16(hCodec, x));
            inp_buf[0]=*x;

            run_filter();

            while (!DSK5510_AIC23_write16(hCodec, inp_buf[0])); //вывод
исходного сигнала в левый канал
            while (!DSK5510_AIC23_write16(hCodec, out_buf[0])); //вывод
отфильтрованного сиганала в правый канал
            }
}
```

4. В директорию с проектом скопировать файл «filter.h» из проекта рассмотренного во втором разделе.

5. В файле «filter.h» изменить значение константы «LENBUF» со 120 на 1. Таким образом, размер входного и выходного буфера станет равным единицы.

6. Линейный выход звуковой карты соединить с линейным входом платы DSK5510.

7. В разветвитель, в линейном входе звуковой карты, вставить колонки (наушники).

8. Свободное гнездо разветвителя соединить с линейным выходом платы DSK5510.

9. Запустить программу «TES2» и открыть окно спектроанализатора.

10. Скомпилировать проект в среде ИСР CCS, и загрузить в плату.

11. Поставить на воспроизведение любой звуковой файл.

12. Запустить программу на выполнение, после этого в окне спектроанализатора программы «TES2» появится спектр сигнала, рис. 3.1.

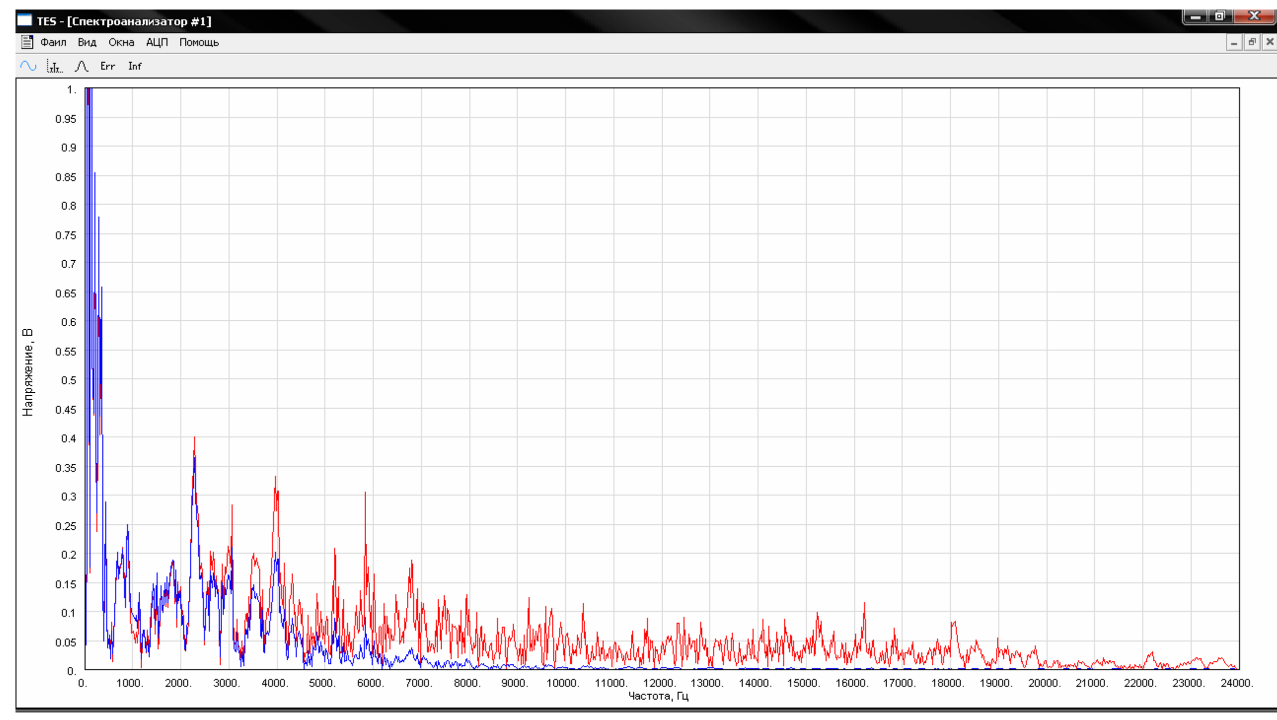


Рис. 3.1 Спектр исходного и отфильтрованного сигнала

Здесь красным цветом обозначен исходный сигнал, а синим отфильтрованный сигнал. Программа «TES2» анализирует левый и правый канал стерео сигнала, разделяя их на канал A и канал B.

Частота среза вычисляется следующим образом: 44000 Гц (частота дискретизации) * 0.0773 ≈ 3400 Гц.

Индивидуальные задания студентам выдаются во время лабораторной работы преподавателем.

Лабораторная работа 4

ЦИФРОВАЯ ФИЛЬТРАЦИЯ. РЕАЛИЗАЦИЯ ФИЛЬТРА С БЕСКОНЕЧНОЙ ИМПУЛЬСНОЙ ХАРАКТЕРИСТИКОЙ (БИХ)

1. Фильтры с бесконечной импульсной характеристикой – БИХ

КИХ-фильтры не имеют реальных аналоговых эквивалентов. Самой близкой аналогией является фильтр скользящего среднего с взвешиванием. Кроме того, частотные характеристики КИХ-фильтров имеют только нули и не имеют полюсов. С другой стороны, БИХ-фильтры имеют традиционные аналоговые эквиваленты (фильтры Баттерворта, Чебышева, эллиптический и Бесселя) и могут быть проанализированы и синтезированы с использованием традиционных методов проектирования фильтров.

БИХ-фильтры получили такое название, потому что их импульсные характеристики растянуты на бесконечном временном интервале. Это объясняется тем, что данные фильтры являются рекурсивными, т. е. используют обратную связь. Хотя БИХ-фильтры могут быть реализованы с меньшим, чем КИХ-фильтры, количеством вычислений, БИХ-фильтры не могут иметь таких характеристик, которыми обладают КИХ-фильтры. Более того, БИХ-фильтр не имеет линейной фазовой характеристики. Но вычислительные преимущества БИХ-фильтра теряются, когда выходной сигнал фильтра подвергается децимации, поскольку в этом случае всякий раз приходится вычислять заново значение выходной величины.

БИХ-фильтры обычно реализуются с помощью звеньев второго порядка, которые называются биквадратными фильтрами, потому что описываются биквадратными уравнениями в z-области. Фильтры высокого порядка проектируют, используя каскадирование биквадратных звеньев. Например, фильтр шестого порядка требует трех биквадратных звеньев.

Особенности фильтра с бесконечной импульсной характеристикой:

1) имеют обратную связь (рекурсия);
2) импульсная характеристика имеет бесконечную длительность;
3) потенциально нестабильны;
4) нелинейная фазочастотная характеристика;
5) более эффективны, чем КИХ-фильтры;
6) нет вычислительных преимуществ при децимации по выходу;
7) обычно проектируется по характеристике аналогового фильтра;
8) обычно реализуется каскадным соединением звеньев второго порядка (биквадратные фильтры).

Структура биквадратного БИХ-фильтра представлена на рис. 1.1. Нули формируются коэффициентами прямой связи b_0, b_1 и b_2; а полюса (порядок) определяются коэффициентами обратной связи a_1 и a_2.

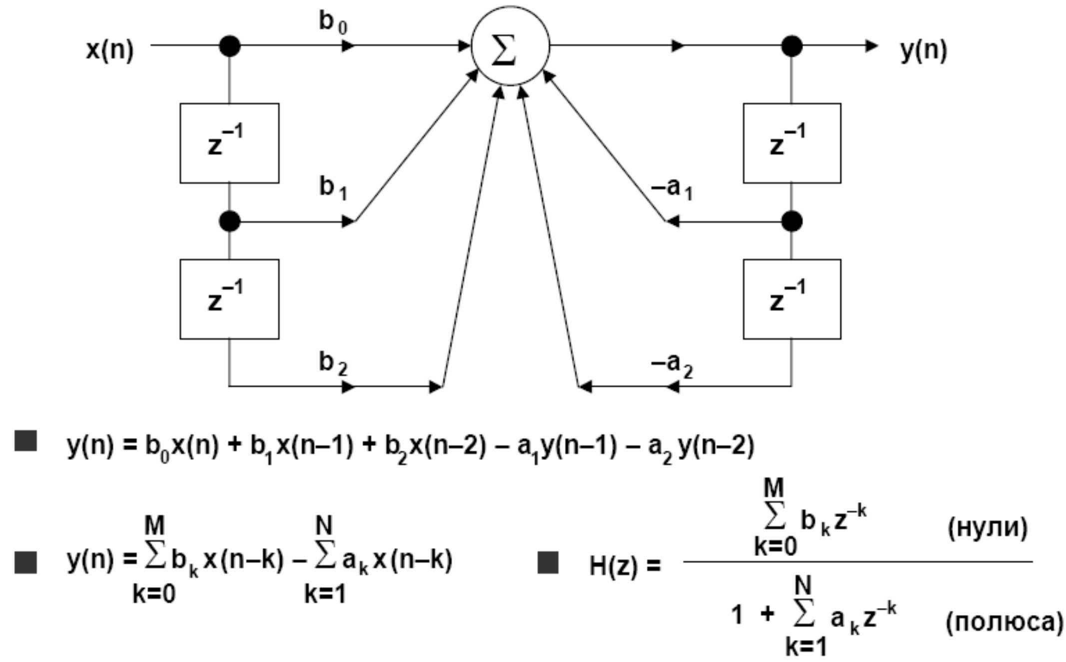


Рис. 1.1 Аппаратная реализация БИХ-фильтра второго порядка (биквадратного)

Общее уравнение цифрового фильтра, представленное на рис. 1.1, описывает обобщенную передаточную функцию H(z), которая содержит полиномы и в числителе, и в знаменателе. Корни знаменателя определяют расположение полюсов фильтра, а корни числителя характеризуют расположение нулей. Хотя существует возможность создания непосредственно по этому уравнению БИХ-фильтра более высокого порядка (так называемая прямая реализация), накапливающиеся ошибки квантования (из-за арифметики с фиксированной точкой и конечной длины слова) могут вызывать неустойчивость работы фильтра и большие ошибки. По этой причине правильнее расположить каскадно несколько биквадратных звеньев с соответствующими коэффициентами, чем использовать прямую форму реализации. Данные при вычислении биквадратных фильтров могут масштабироваться раздельно, а затем биквадратные звенья каскадируются для минимизации ошибок квантования коэффициентов и накапливающихся ошибок рекурсивного накопления. Каскадные биквадратные фильтры работают более медленно, чем их эквиваленты прямой формы реализации, но они более устойчивы и в них минимизируются эффекты, связанные с арифметическими ошибками конечной разрядности данных.

Первая прямая форма биквадратного звена, представленная на рис. 1.1, требует использования четырех регистров. Эта конфигурация может быть заменена эквивалентной схемой, представленной на рис. 1.2, которая называется второй прямой формой реализации и требует использования только двух регистров. Можно показать, что уравнения, описывающие биквадрат-

ный БИХ-фильтр второй прямой формы реализации, такие же, как и уравнения первой прямой формы реализации. Как и в случае КИХ-фильтра, система обозначений при изображении БИХ-фильтра часто упрощается, как показано на рис. 1.3.

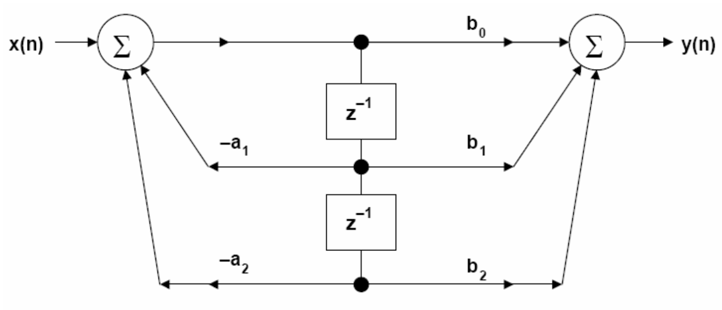


Рис. 1.2 Биквадратный БИХ-фильтр форма 2

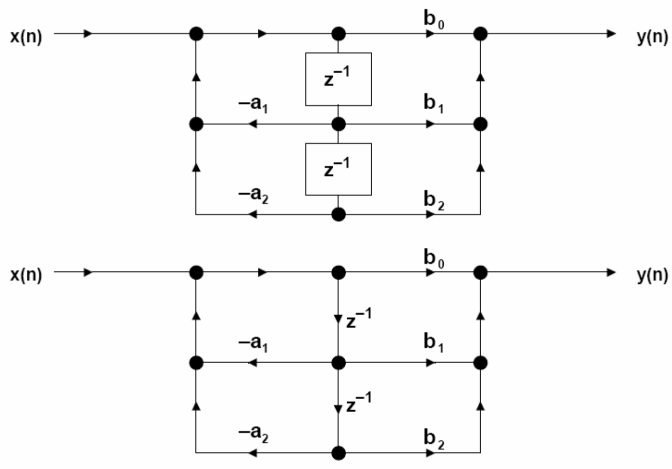


Рис. 1.3 Упрощенная схема биквадратного БИХ-фильтра

Методы проектирования БИХ-фильтров

Популярный метод проектирования БИХ-фильтра сводится к тому, что сначала проектируется эквивалентный аналоговый фильтр, а затем функция передачи H(s) преобразуется математически в z-область, H(z). Проектирование фильтров более высоких порядков выполняется каскадированием биквадратных звеньев. Наиболее популярными аналоговыми фильтрами являются фильтры Баттерворта, Чебышева, эллиптические и Бесселя. Существует множество программ САПР, способных генерировать функцию передачи фильтра, заданную с помощью преобразования Лапласа.

Фильтр Баттерворта, не имеющий нулей частотной характеристики, (также называемый фильтром с максимально плоской характеристикой), не

создает пульсаций (неравномерности) в полосе пропускания и в полосе задержки, т. е. обладает монотонной характеристикой в обеих полосах. Фильтр Чебышева 1-го рода имеет более быстрый спад частотной характеристики, чем фильтр Баттерворта (при равном порядке), и создает пульсации (неравномерность) в полосе пропускания. Реже используются фильтры Чебышева 2-го рода, имеющие пульсации (неравномерность) в полосе задержки, а не в полосе пропускания.

Эллиптический фильтр (фильтр Кауэра) имеет полюса и нули частотной характеристики и создает пульсации (неравномерность) и в полосе пропускания, и в полосе задержки. Этот фильтр имеет более быстрый спад частотной характеристики, чем фильтр Чебышева при том же числе полюсов (порядке). Эллиптический фильтр часто используется там, где допускается несколько худшая фазовая характеристика.

Наконец, фильтр Бесселя (Томпсона), который не имеет нулей частотной характеристики, обладает оптимальной импульсной характеристикой и линейной фазовой характеристикой, но имеет худший спад частотной характеристики из всех типов обсуждавшихся фильтров при том же числе полюсов (порядке).

Популярные аналоговые фильтры:

1. Баттерворта:

a) нет нулей частотной характеристики, нет пульсаций в полосе пропускания и задержки;

b) максимально плоская характеристика (быстрый спад без пульсаций).

2. Чебышева 1-го рода:

a) нет нулей частотной характеристики, пульсации в полосе пропускания, нет пульсаций в полосе задержки;

b) более короткая область перехода, чем у фильтра Баттерворта для данного порядка;

c) фильтр 2-го рода имеет пульсации в полосе задержки, нет пульсаций в полосе пропускания.

3. Эллиптический (Кауэра):

a) имеет полюса и нули, пульсации и в полосе пропускания, и в полосе задержки;

b) более короткая область перехода, чем у фильтра Чебышева для данного порядка;

c) фазовая характеристика хуже.

4. Бесселя (Томпсона):

a) нет нулей частотной характеристики, нет пульсаций в полосе пропускания и задержки;

b) оптимизирован по линейной фазовой и импульсной характеристикам;

c) самая длинная переходная характеристика из всех фильтров данного порядка.

Все вышеперечисленные типы аналоговых фильтров описаны в литературе, их преобразования по Лапласу H(s) доступны либо из таблиц, либо могут быть получены с помощью средств САПР. Существует три метода преобразования изображения по Лапласу в z-изображение: метод инвариантности импульсной характеристики, билинейное преобразование и согласованное z-преобразование. Результирующее z-изображение может быть преобразовано в коэффициенты биквадратного БИХ-фильтра.

Подход САПР при проектировании БИХ-фильтра подобен программе Паркса-Макклиллана, используемой для КИХ-фильтров. Эта методика использует алгоритм Флетчера-Пауэла (Fletcher-Powell).

При вычислении производительности специального процессора DSP, предназначенного для реализации БИХ-фильтров, необходимо исследовать эталонные требования эффективности вычислений для биквадратного звена фильтра.

Методы проектирования БИХ-фильтров:

1. Метод инвариантности импульсной характеристики:

a) начинается с определения H(s) для аналогового фильтра;

b) взятие обратного преобразования Лапласа для получения импульсной характеристики;

c) получение z-преобразования H(z) из дискретной импульсной характеристики;

d) z-преобразование выдает коэффициенты фильтра;

e) должен быть учтен эффект наложения спектров.

2. Метод билинейного преобразования:

a) другой метод для преобразования H(s) в H(z);

b) характеристики определяются дифференциальным уравнением, описывающим аналоговую систему;

c) не важен эффект наложения спектра.

3. Метод согласованного z-преобразования

a) отображает H(s) в H(z) для фильтров и с полюсами, и с нулями.

4. Методы САПР:

a) алгоритм Флетчера-Пауэла;

b) осуществляются каскадированием биквадратных звеньев.

Скорость обработки данных при реализации БИХ-фильтров:

1) определение количества биквадратных звеньев, требуемых для реализации желаемой частотной характеристики;

2) умножение этого количества на время выполнения одного биквадратного звена для DSP процессор;

3) результат (плюс дополнительные операции) является минимально допустимым периодом дискретизации (1/fs) для работы в реальном масштабе времени.

Сравнение КИХ- и БИХ-фильтров

При выборе между КИХ- и БИХ-фильтрами следует руководствоваться несколькими основными принципами. Как правило, БИХ-фильтры более эффективны, чем КИХ-фильтры, потому что они требуют меньшего количества памяти и меньшего количества операций умножения с накоплением. БИХ-фильтры могут быть разработаны, основываясь на предыдущем опыте проектирования аналоговых фильтров. БИХ-фильтры могут приносить проблемы неустойчивости, но это происходит реже, если проектируемые фильтры высокого порядка реализуются как системы, состоящие из каскадов второго порядка.

С другой стороны, КИХ-фильтры требуют большего количества звеньев и, соответственно, операций умножения с накоплением для реализации частотной характеристики с заданной частотой среза, но при этом имеют линейную фазовую характеристику. КИХ-фильтры работают на конечном временном интервале данных, поэтому, если часть данных испорчена (например, в результате сбоев в работе АЦП), КИХ-фильтр будет «звенеть» только на временном интервале, соответствующем N-1 отсчетам. БИХ-фильтр, из-за наличия обратной связи, будет «звенеть» значительно более длительный период времени.

Таблица 1.1

Сравнение КИХ-и БИХ-фильтров

БИХ-фильтры	КИХ-фильтры
Более эффективны	Менее эффективны
Есть аналоговый эквивалент	Нет аналогового эквивалента
Могут быть нестабильными	Всегда стабильны
Нелинейная фазовая характеристика	Линейная фазовая характеристика
Больше «звон» при наличии сложных сигналов	Меньше «звон» при наличии сложных сигналов
Доступны средства САПР	Доступны средства САПР
Децимация не влияет на эффективность	Децимация увеличивает эффективность

Если необходимы фильтры с крутым спадом и испытывается дефицит во времени, отведенном для обработки, хорошим выбором являются эллиптические БИХ-фильтры. Если число операций умножения с накопле-

нием не является чрезмерным и требуется линейная фаза, то должен быть выбран КИХ-фильтр.

Расчет коэффициентов БИХ-фильтра в математическом пакете MATLAB 7.0

Для расчета коэффициентов БИХ-фильтра в MATLAB 7.0 предусмотрено множество функций, в том числе и специальная утилита Filter Design & Analysis Tool (FDA Tool), однако, функция yulewalk(Nc, f, m) позволяет это сделать наиболее просто.

Входные параметры:

1) Nc – Количество коэффициентов плюс 1;

2) f – Вектор точек частоты, изменяется от 0 до 1, где 1 это половина частоты дискретизации (0.5 нормализованной частоты);

3) m – Величина амплитуды в точках, указанных в векторе f, исходная амплитуда принимается равной единицы. Количество элементов m и f равны;

4) выходные параметры – коэффициенты B и A.

Для построения импульсной характеристики используется функция **impz(B, A, N)**, где B и A – коэффициенты БИХ-фильтра, а N – количество точек для построения.

Для построения АЧХ и ФЧХ фильтра используется функция **freqz(B, A, N, Fd)**, где B и A – коэффициенты БИХ-фильтра, N – количество точек для построения, Fd – частота дискретизации.

На рисунке 1.4 показан пример расчет ФНЧ.

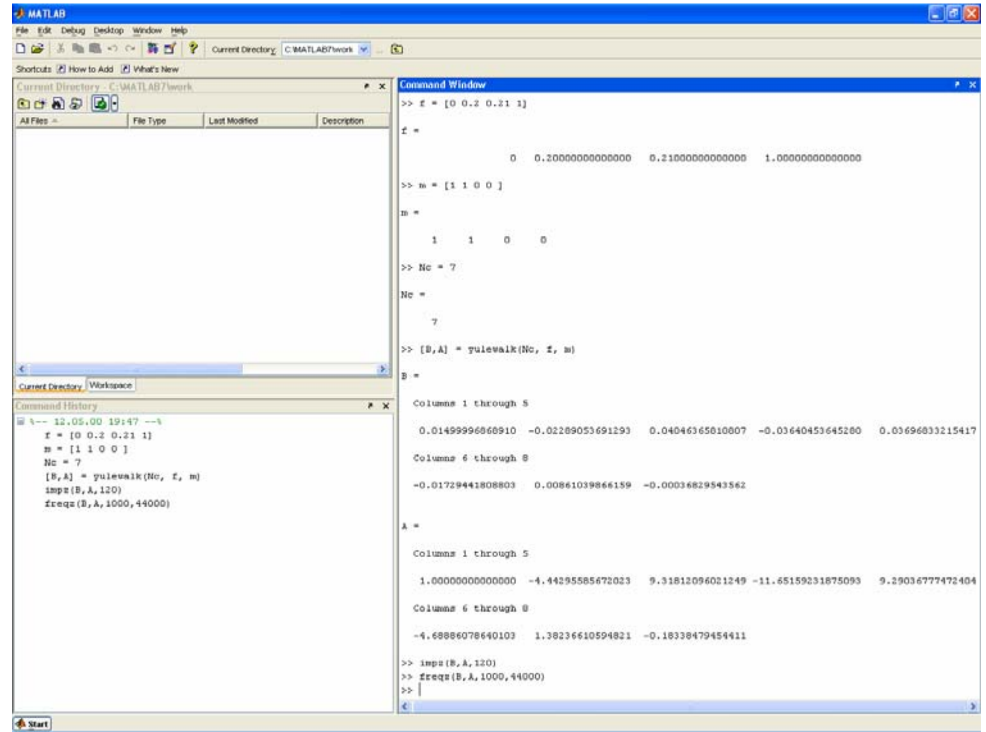


Рис. 1.4 Пример расчета ФНЧ фильтра в среде MATLAB

1) Фильтр низкой частоты с нормальной частотой среза 0.1 (табл. 1.2)

Таблица 1.2

Коэффициенты ФНЧ

f = [0 0.2 0.21 1], m = [1 1 0 0]				
A	Умноженное на 512	Округление до целого	Погрешность	Погрешность, %
1	512.0000000000	512	0.0000000000	0.0000000000
-4.442955857	-2274.7933986408	-2275	0.0004035183	0.0403518280
9.31812096	4770.8779316288	4771	0.0002384148	0.0238414788
-11.65159232	-5965.6152672005	-5966	0.0007514312	0.0751431249
9.290367775	4756.6683006587	4757	0.0006478503	0.0647850276
-4.688860786	-2400.6967226373	-2401	0.0005923386	0.0592338599
1.382366106	707.7714462455	708	0.0004463941	0.0446394052
-0.183384795	-93.8930148066	-94	0.0002089555	0.0208955456
				0.3288902699
B	Умноженное на 512	Округление до целого	Погрешность	Погрешность, %
0.014999969	7.6799839688	8	0.0006250313	0.0625031311
-0.022890537	-11.7199548994	-12	0.0005469631	0.0546963087
0.040463658	20.7173929513	21	0.0005519669	0.0551966892
-0.036404536	-18.6391226638	-19	0.0007048385	0.0704838547
0.036968332	18.9277860629	19	0.0001410428	0.0141042846
-0.017294418	-8.8547420611	-9	0.0002837069	0.0283706912
0.008610399	4.4085241147	4	0.0007978987	0.0797898662
-0.000368295	-0.1885672630	0	0.0003682954	0.0368295436
				0.4019743692
Разность суммы погрешностей (B – A)				0.0730840993

2) Фильтр низкой частоты с нормальной частотой среза 0.2 (табл. 1.3)

Таблица 1.3

Коэффициенты ФНЧ

f = [0 0.4 0.41 1], m = [1 1 0 0]				
A	Умноженное на 512	Округление до целого	Погрешность	Погрешность, %
1	512.0000000000	512	0.0000000000	0.0000000000
-1.536825788	-786.8548032301	-787	0.0002835874	0.0283587441
2.56905269	1315.3549772448	1315	0.0006933149	0.0693314931
-2.18828026	-1120.3994929432	-1120	0.0007802597	0.0780259655
1.669214529	854.6378389899	855	0.0007073457	0.0707345723
-0.767060026	-392.7347335594	-393	0.0005180985	0.0518098517
0.254588948	130.3495414251	130	0.0006826981	0.0682698096
-0.037346468	-19.1213916878	-19	0.0002370931	0.0237093140
				0.3902397503
B	Умноженное на 512	Округление до целого	Погрешность	Погрешность, %
0.055419464	28.3747655520	28	0.0007319640	0.0731963969
0.119305357	61.0843427943	61	0.0001647320	0.0164732020
0.190984545	97.7840867969	98	0.0004217055	0.0421705475
0.227660285	116.5620658969	117	0.0008553400	0.0855340045

0.185596674	95.0254971016	95	0.0000497990	0.0049799027
0.121731981	62.3267743792	62	0.0006382312	0.0638231209
0.049738973	25.4663540577	25	0.0009108478	0.0910847769
0.0131243	6.7196417292	7	0.0005475747	0.0547574748
				0.4320194261
Разность суммы погрешностей (В – А)				0.0417796759

3) Полосовой фильтр с нормальной частотой пропускания 0.165 – 0.33

Таблица 1.4

Коэффициенты полосового фильтра

f [0 0.32 0.33 0.66 0.67 1] m = [0 0 1 1 0 0]				
А	Умноженное на 512	Округление до целого	Погрешность	Погрешность, %
1	512.0000000000	512	0.0000000000	0.0000000000
0.017922482	9.1763109225	9	0.0003443573	0.0344357270
0.940927029	481.7546389739	482	0.0004792208	0.0479220754
0.010435385	5.3429172093	5	0.0006697602	0.0669760174
0.660088668	337.9653980685	338	0.0000675819	0.0067581897
0.0341823	17.5013374301	18	0.0009739503	0.0973950332
0.112911141	57.8105044096	58	0.0003701086	0.0370108575
0.005846488	2.9934020710	3	0.0000128866	0.0012886580
				0.2917865583
В	Умноженное на 512	Округление до целого	Погрешность	Погрешность, %
0.119059995	60.9587173885	61	0.0000806301	0.0080630101
0.012257812	6.2759996612	6	0.0005390618	0.0539061838
-0.181295087	-92.8230843331	-93	0.0003455384	0.0345538412
-0.025091566	-12.8468817547	-13	0.0002990591	0.0299059073
0.18151005	92.9331456229	93	0.0001305750	0.0130574955
0.030708454	15.7227282019	16	0.0005415465	0.0541546481
-0.119353206	-61.1088417217	-61	0.0002125815	0.0212581488
-0.017831151	-9.1295491884	-9	0.0002530258	0.0253025759
				0.2402018106
Разность суммы погрешностей (В - А)				0.0515847477

2. Реализация БИХ-фильтра на симуляторе DSK5510

Программная реализация БИХ-фильтра аналогична реализации КИХ-фильтра, однако, добавляется дополнительная линия задержки и дополнительный цикл накопления с умножением для обработки коэффициентов А.

1. Необходимо создать папку «laba_4» и в ней папку «exp_1».

2. Скопировать в папку с проектом все файлы из папки «exp_1» работы № 3.

3. Внести изменения в файл «filter.h» в соответствии с листингом 2.1.

Листинг 2.1 – Листинг файла «filter.h»

```
#include <stdio.h>

#define LENBUF            120
#define LENFILTER_B       8
#define LENFILTER_A       8

//коэффициенты фильтра умноженные на число 512
 int coeffBuff_b[]={8,-12,21,-19,19,-9,4,0};//(0.2)/2
 //8,-12,21,-19,19,-9,4,0};//(0.2)/2
 //28,61,98,116,95,62,26,7};//(0.4)/2
 //61,6,-93,-13,93,16,-61,-9};//полосовой

 int coeffBuff_a[]={512,-2275,4770,-5966,4757,-2401,708,-94};//(0.2)/2
  //512,-2275,4770,-5966,4757,-2401,708,-94};//(0.2)/2
 //512,-787,1315,-1120,854,-393,130,-19};//(0.4)/2
 //512,9,482,5,338,18,58,3};//полосовой

//входной буфер
 int inp_buf[LENBUF];
//выходной буфер
 int out_buf[LENBUF];
//буфер линии задержки
 int SimplBuff_b[LENFILTER_B];
 int SimplBuff_a[LENFILTER_A];
//вспомогательная переменная
 int coeff;

//счетчики
int count;
int count_run;
int count_con;
```

```
//функция запуска фильтрации
int run_filter()
{
 for(count_run=0;count_run<LENBUF;count_run++)
           {
           //чтение входного отсчета
           coeff=inp_buf[count_run];
           //запись текущего отсчета в линию задержки
           SimplBuff_b[0]=coeff;
           //запуск функции свертки сигнала
           coeff=convolution();
           //запись выходного отсчета в линию задержки
           SimplBuff_a[0]=coeff;
           //запись в выходной буфер отфильтрованного сигнала
           out_buf[count_run]=coeff;//+150;
           }
 return 0;
}
//функция осуществляющая свертку сигнала
int convolution()
{
 //вспомогательные переменные
 long int coeff_mxb,coeff_mxa;
 long int summ_b,summ_a;
 long int tmp1,tmp2,tmp3,tmp4;
//---------------------------------------------------------------------------
----
 for(summ_b=0,count_con=0;count_con<LENFILTER_B;count_con++)
```

```
            {
            //чтение текущего отсчета из линии задержки
            tmp1=SimplBuff_b[count_con];
            //чтение коэффициента фильтра
            tmp2=coeffBuff_b[count_con];
            //перемножение отсчета с коэффициентом фильтра
            coeff_mxb=tmp1*tmp2;
            //накопление результата
            summ_b +=coeff_mxb;
            }
 //цикл сдвига линии задержки на единицу вправо
 for(count_con=LENFILTER_B-1;count_con>0;count_con--)
            {
            SimplBuff_b[count_con]=SimplBuff_b[count_con-1];
            }
//----------------------------------------------------------------------------
----
 //цикл сдвига линии задержки на единицу вправо
  for(count_con=LENFILTER_A-1;count_con>0;count_con--)
            {
            SimplBuff_a[count_con]=SimplBuff_a[count_con-1];
            }

 for(summ_a=0,count_con=1;count_con<LENFILTER_A;count_con++)
            {
            //чтение текущего отсчета из линии задержки
            tmp3=SimplBuff_a[count_con];
            //чтение коэффициента фильтра
            tmp4=coeffBuff_a[count_con];
            //перемножение отсчета с коэффициентом фильтра
            coeff_mxa=tmp3*tmp4;
            //накопление результата
            summ_a +=coeff_mxa;
            }
 //нахождения разности сумм и деление на 512
 return (summ_b-summ_a)>>9;
}

//функция очистки линии задержки
void ClearSimpleBuff()
{
 for(count=0;count<LENFILTER_B;count++)
            SimplBuff_b[count]=0;

 for(count=0;count<LENFILTER_A;count++)
            SimplBuff_a[count]=0;
}
```

4. Файл «main.c» остается без изменений.
5. Скомпилировать и загрузить проект в симулятор.

Тестирование БИХ-фильтра

А. Фильтр низкой частоты с нормальной частотой среза 0.1

Импульсная характеристика, построенная по средствам MATLAB 7.0, приведена на рис. 2.1, АЧХ-и ФЧХ-фильтра на рис. 2.2.

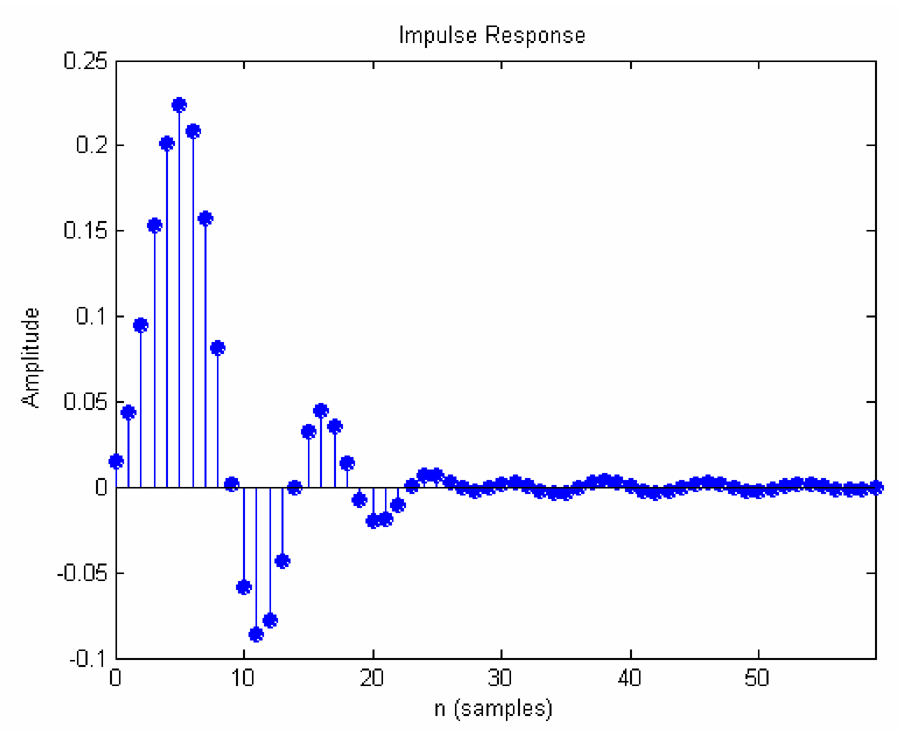


Рис. 2.1 Импульсная характеристика

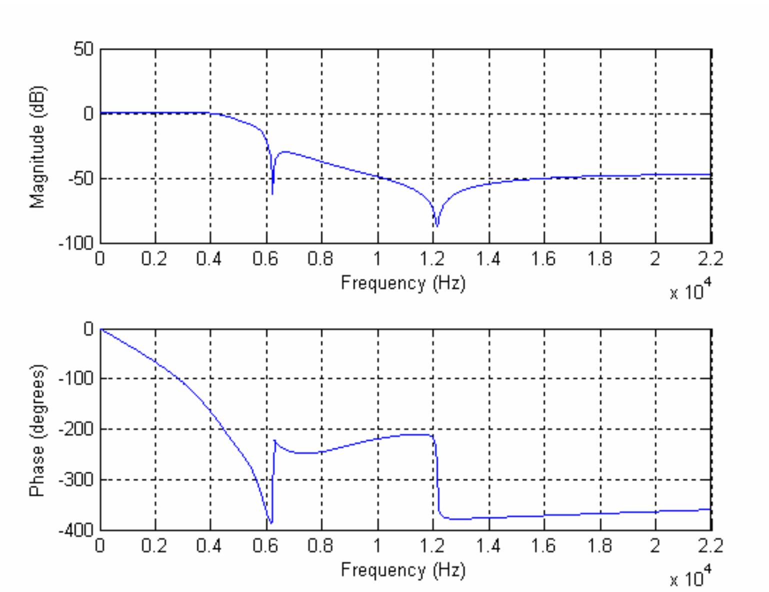


Рис. 2.2 АЧХ и ФЧХ для частоты дискретизации 44 кГц

В файле «filter.h» пара коэффициентов для этого фильтра закомментирована, поэтому необходимо скопировать в буфер обмена и вставить эти коэффициенты в соответствующие массивы coeffBuff_b и coeffBuff_a. Для построения импульсной характеристики необходимо закомментировать участок кода в файле «main.c» отвечающий за считывание данных из файла «**coldplay_in.wav**», и раскомментировать участок кода, который отвечает за формирование единичного импульса.

После этого нужно перекомпилировать проект, запустить его на выполнение и построить график по данным, содержащимся в выходном буфере (out_buf). Все действия аналогичны тем, что были в работе № 3, при тестирование КИХ-фильтра. Импульсная характеристика приведена на рис. 2.3.

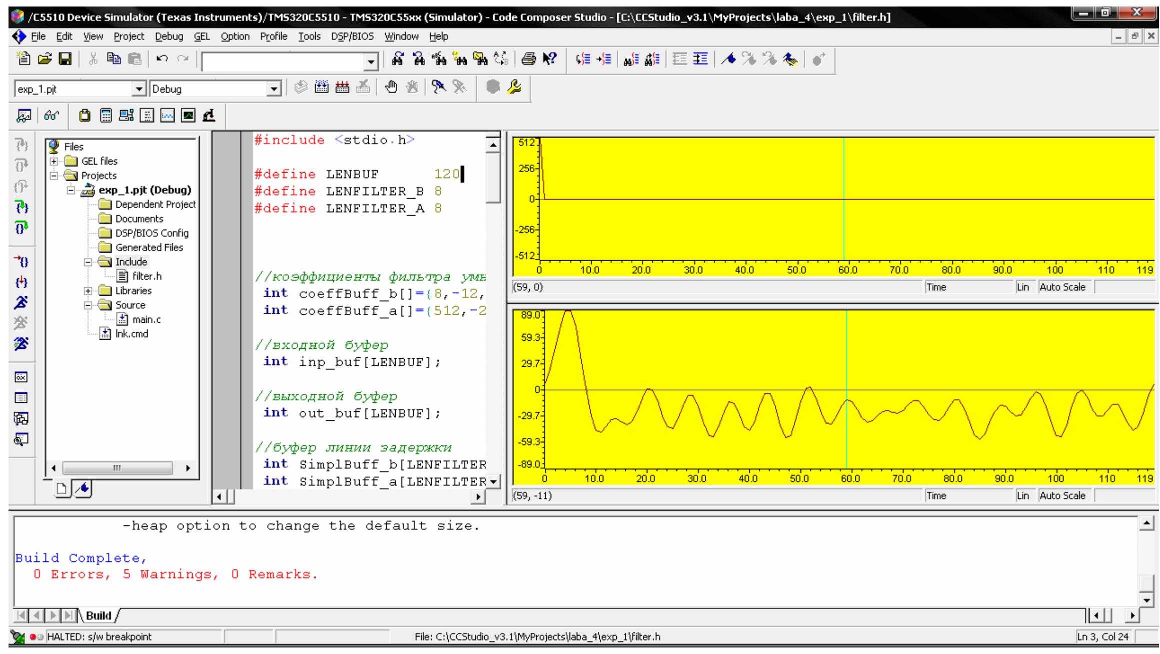


Рис. 2.3 Импульсная характеристика

После построения импульсной характеристики необходимо внести изменения в исходный код, для того, чтобы произвести фильтрацию звукового файла «**coldplay_in.wav**». Осциллограмма фильтрованного и исходного сигнала представлена на рисунке 2.4.

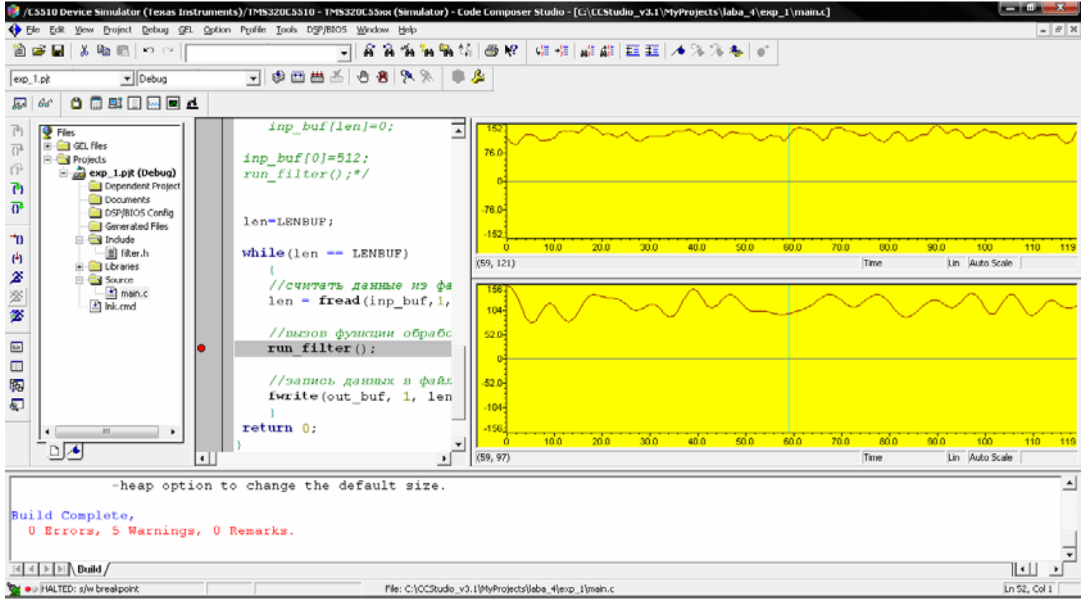


Рис. 2.4 Построение осциллограммы входного и выходного файла

По завершении работы программы необходимо воспроизвести выходной звуковой файл «**coldplay_out.wav**» и при помощи программы «TES2» просмотреть спектр звукового сигнал, предварительно соединив надлежащим образом линейные входы и выходы компьютера между собой. Спектрограмма исходного сигнала представлена на рис. 2.5, а результат фильтрации на рис. 2.6.

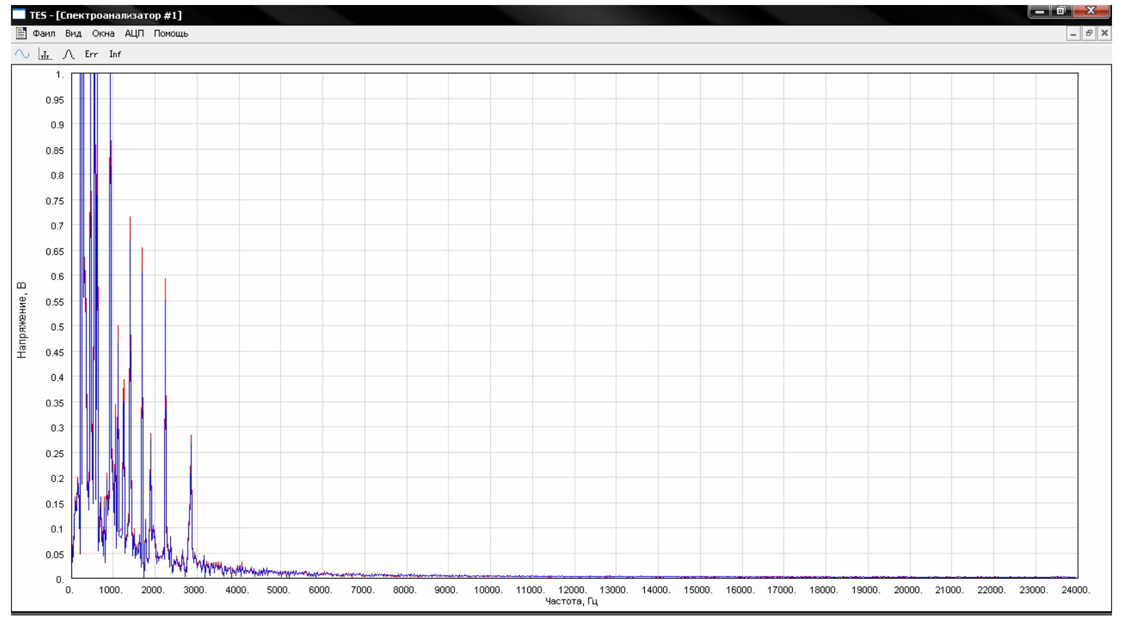


Рис. 2.5 Спектр исходного сигнала

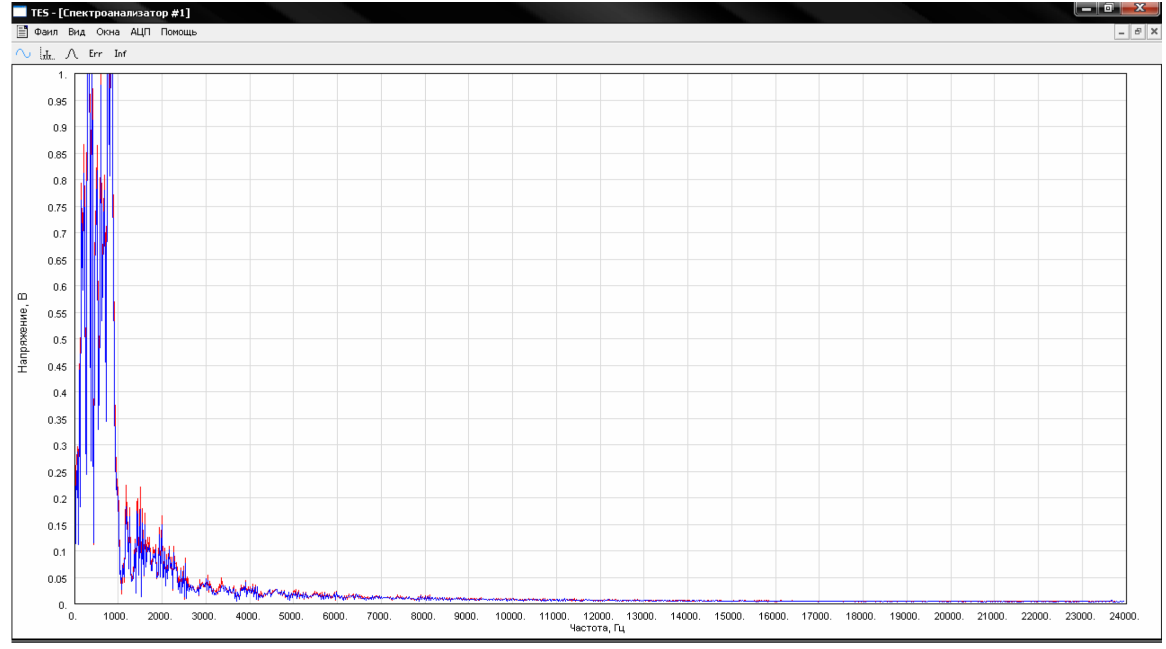


Рис. 2.6 Спектр отфильтрованного сигнала

При воспроизведении выходного файла, отчетливо слышен посторонний звук, а в конце файла «писк». Из спектрограммы, очевидно, что на расчетной частоте среза нет очевидного спада сигнала. Если построить

спектрограмму концовки файла «**coldplay_out.wav**», то она будет аналогична представленной на рис. 2.7.

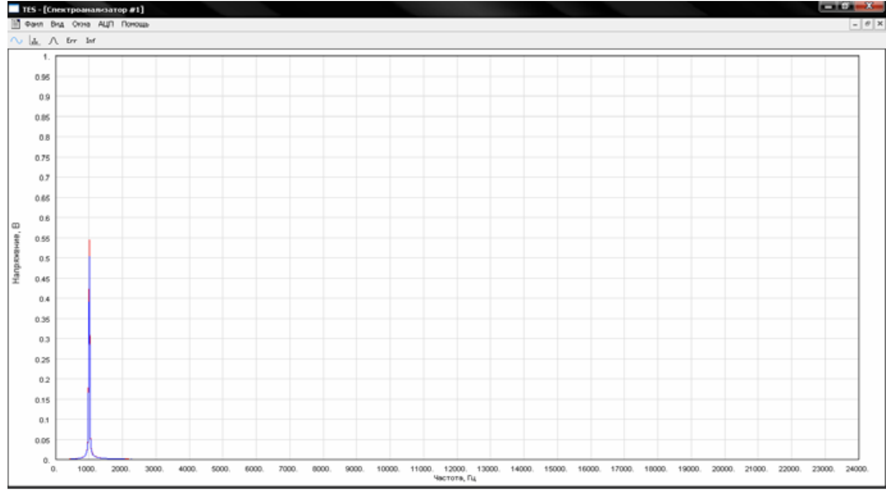


Рис. 2.6 Шум после фильтрации

Появление шума на частоте 1 кГц при частоте дискретизации 8 кГц, вызвано нестабильной работой фильтра, в таком случае говорят, что фильтр неустойчив. Аналогичная картина будет наблюдаться и при фильтрации в реальном времени рис. 3.1, однако, частота шума станет равной 22 кГц, т. к. частота дискретизации равна 44 кГц.

Причиной неустойчивой работы фильтра явилась квантование коэффициентов, т. к. теоретическая импульсная характеристика (рис. 2.1) является правильной, а следовательно, и рассчитанные коэффициенты правильны. Неустойчивую работу фильтра можно наблюдать на рис. 2.3, т. к. эта импульсная характеристика очень сильно отличается от идеальной. Если построить импульсную характеристику аналогичную рис. 2.3 только для 1000 отсчетов, то она будет такой как показано на рис. 2.7.

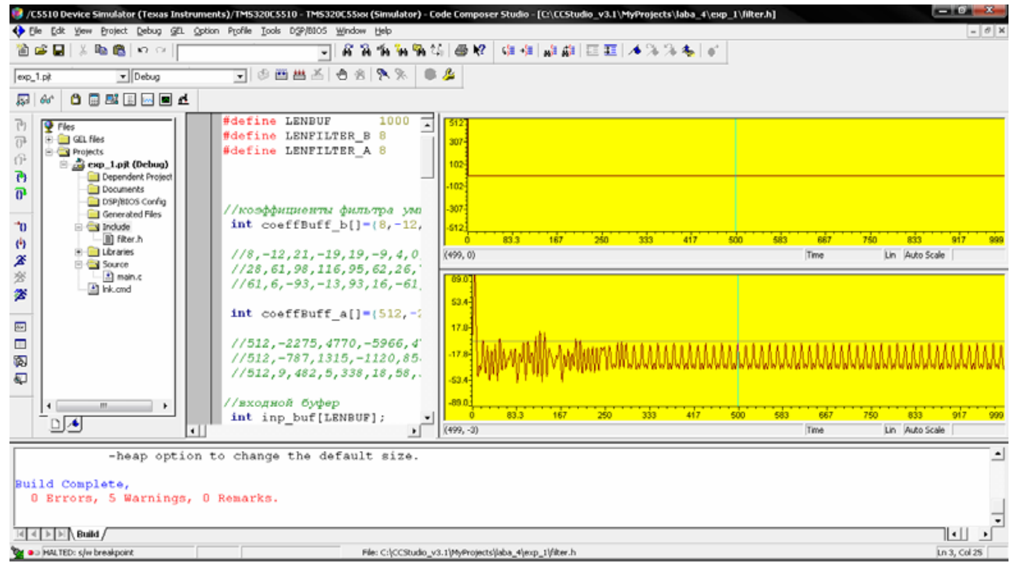


Рис. 2.7 Импульсная характеристика неустойчивого фильтра

Видно, что спустя определенное время после появления единичного импульса, фильтр начинает самопроизвольно «колебаться».

Избежать подобных проблем можно подбором большего коэффициента квантования (число на которое домножаются коэффициенты, и на которое делится конечная сумма после операций умножения с накоплением), это приведет к уменьшению потерь при вычислениях. Кроме того не следует квантовать какой-либо один тип коэффициентов, например тип B, а другой не квантовать, это также приведет к неустойчивой работе фильтра, не зависимо от величины коэффициента.

B. Фильтр низкой частоты с нормальной частотой среза 0.2

Импульсная характеристика, построенная по средствам MATLAB 7.0, приведена на рис. 2.8, АЧХ-и ФЧХ-фильтра на рис. 2.9.

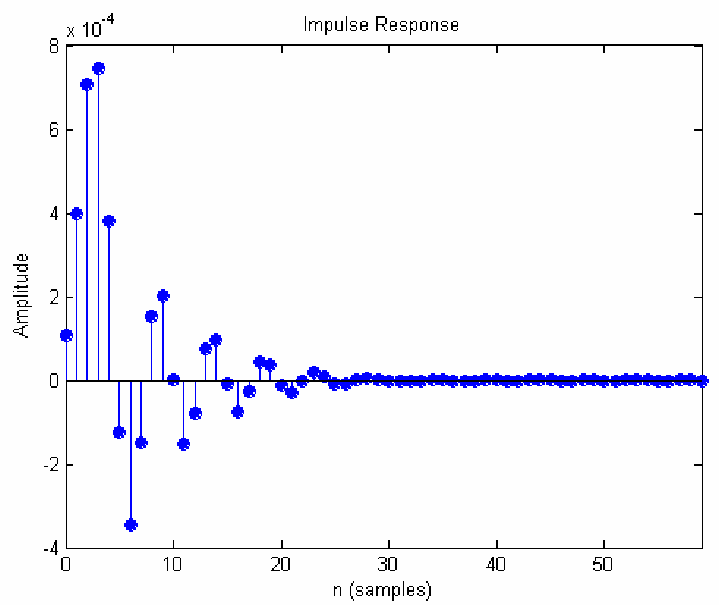


Рис. 2.8 Импульсная характеристика

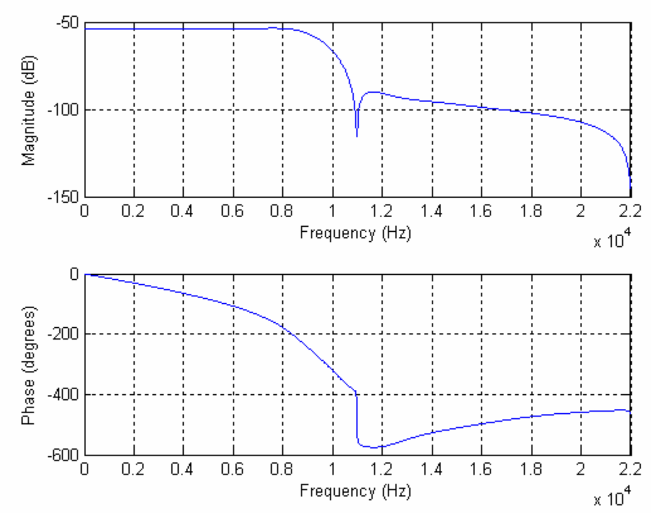


Рис. 2.9 АЧХ-и ФЧХ для частоты дискретизации 44 кГц

Импульсная характеристика, построенная в ИСР CCS, приведена на рис. 2.10.

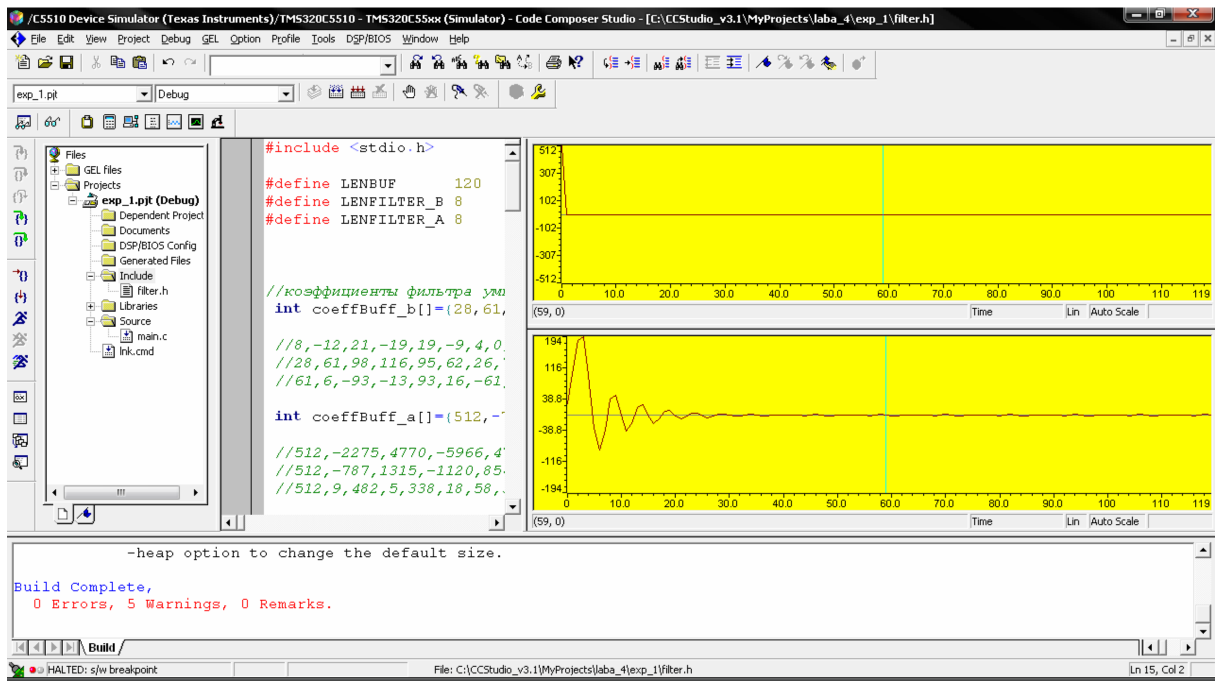


Рис. 2.10 Импульсная характеристика

Очевидно, что эта импульсная характеристика в отличие от импульсной характеристики предыдущего фильтра устойчива. Осциллограмма фильтрации входного звукового файла представлена на рис. 2.11, а спектрограмма выходного файла – на рис. 2.12. В отличие от предыдущего фильтра, отчетливо видна частота среза.

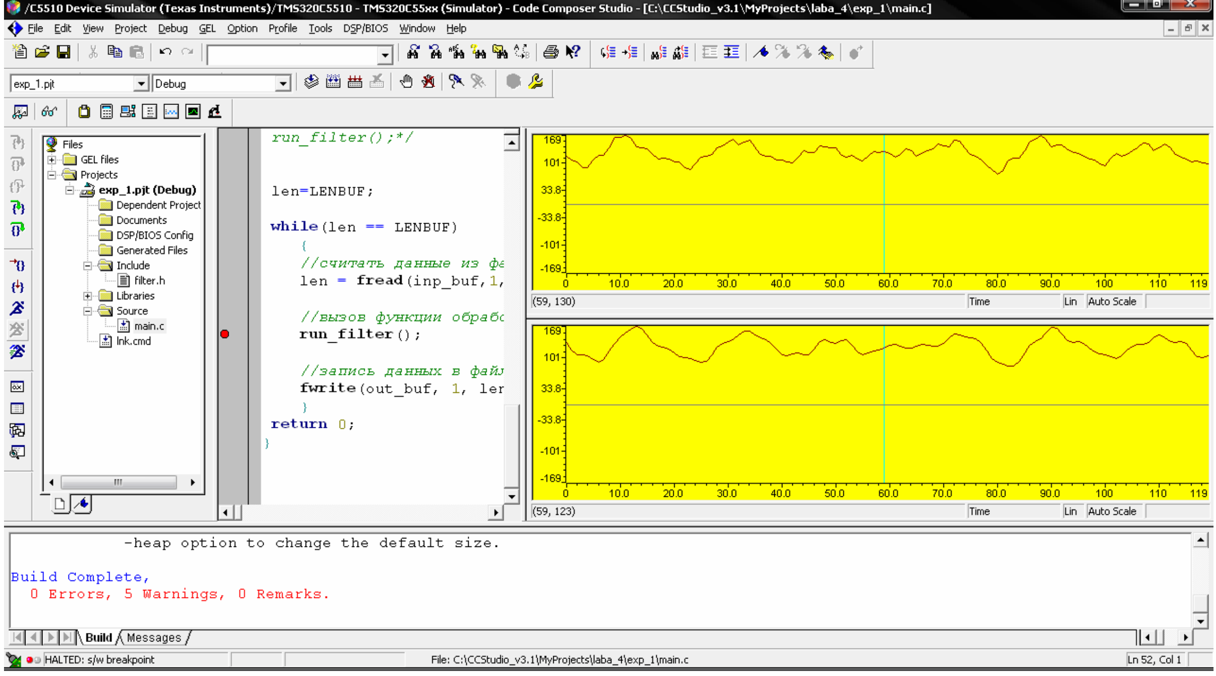


Рис. 2.11 Построение осциллограммы входного и выходного файла

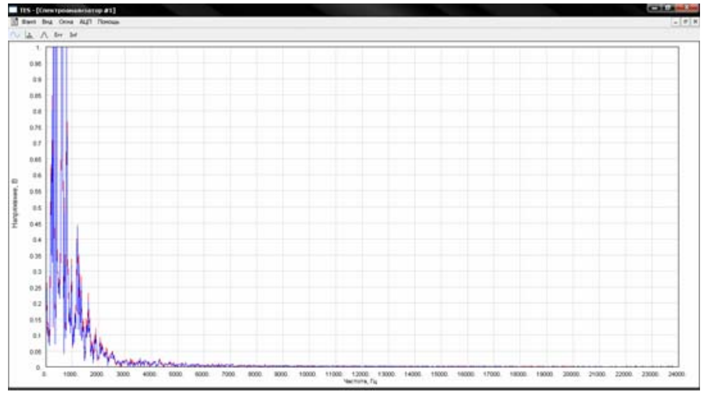


Рис. 2.11 Спектр отфильтрованного сигнала

C. Полосовой фильтр с нормальной частотой среза 0.165 – 0.33

Импульсная характеристика, построенная по средствам MATLAB 7.0, приведена на рисунке 2.12, АЧХ-и ФЧХ-фильтра на рис. 2.13.

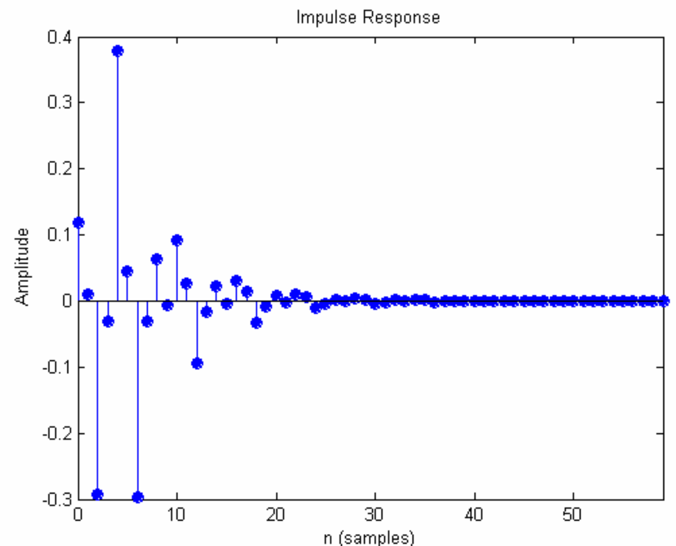


Рис. 2.12 Импульсная характеристика

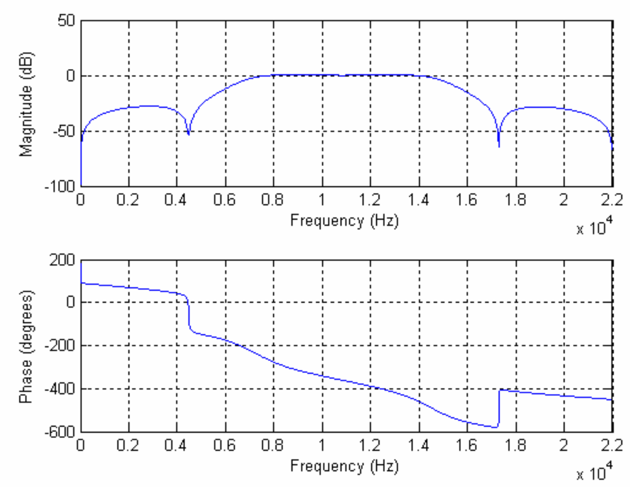


Рис. 2.13 АЧХ и ФЧХ для частоты дискретизации 44 кГц

Импульсная характеристика, построенная в ИСР CCS, приведена на рис. 2.14.

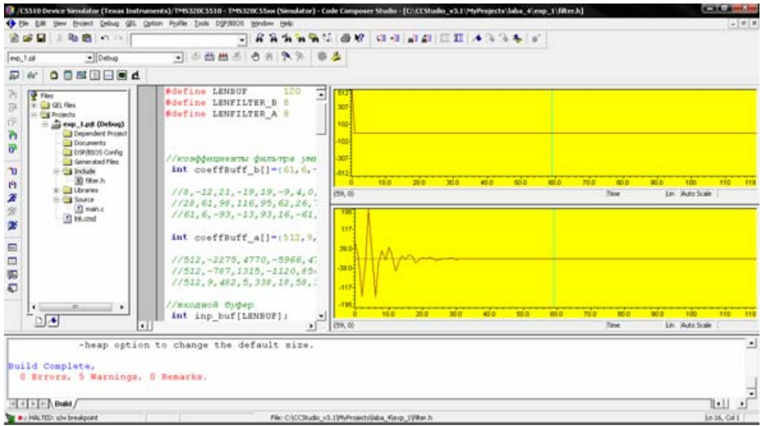


Рис. 2.14 Импульсная характеристика

Если теперь попытаться применить этот полосовой фильтр для фильтрации входного звукового файла, то в выходном файле будет сплошной шум. Это связано с тем, что отсчеты в файлах типа *.wav, хранятся в виде числа от 0 до 255, т. е. положительные. Из осциллограммы представленной на рис. 2.15 видно, что некоторые отсчеты получаются отрицательными, а отрицательные отсчеты проигрывателем будут восприниматься как число 255. За счет этого произойдет искажение. Появление же отрицательных отсчетов объясняется тем, что основная энергия сигнала сосредоточена в области низких частот, а т. к. низкая частота в этом фильтре попадает в полосу задерживания, то сигнал теряет большую часть своей энергии.

Чтобы фильтрация была корректной, необходимо перед помещением отфильтрованного отсчета в выходной буфер, прибавить к нему любое число. Для этого в файле «**filter.h**» в теле функции **run_filter()** сделать следующие изменения: строку «**out_buf[count_run]=coeff;//+150;**» изменить на «**out_buf[count_run]=coeff+150;**». Это приведет к тому, что осциллограмма выходных отсчетов поднимется и станет такой как представлено на рис. 2.16.

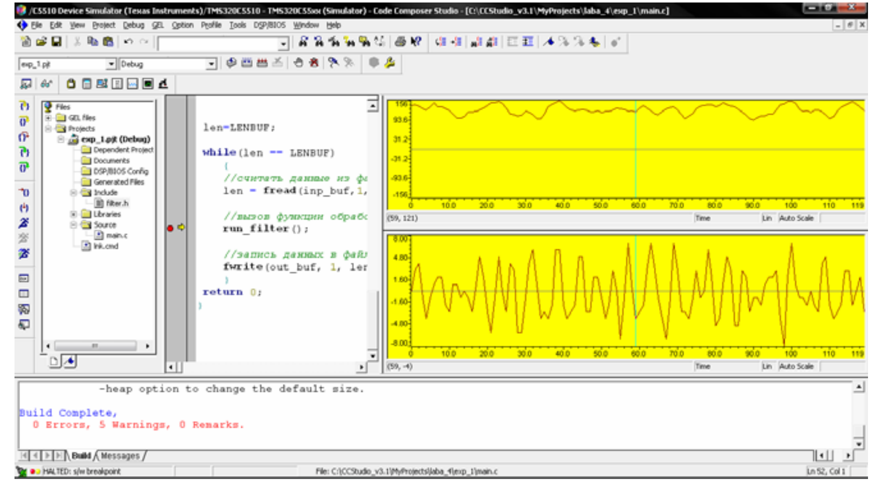


Рис. 2.15 Построение осциллограммы входного и выходного файла

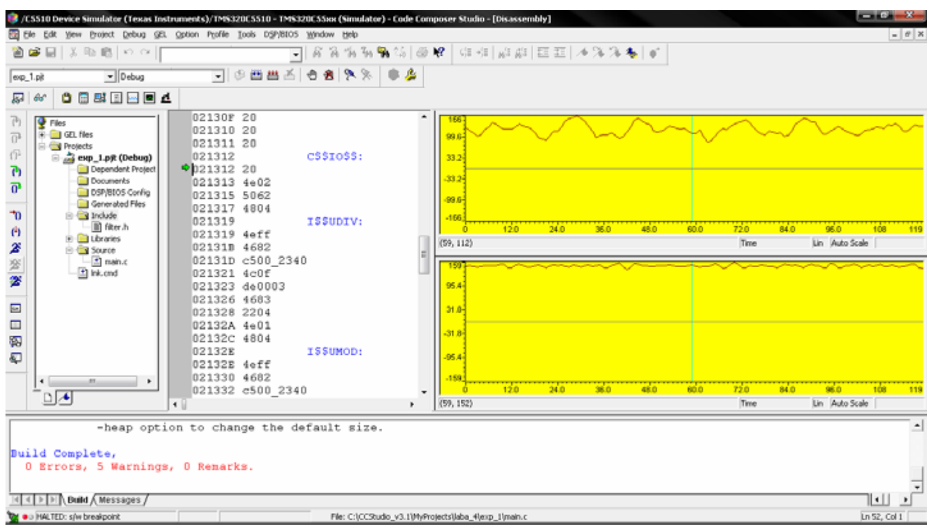


Рис. 2.16 Построение осциллограммы входного и выходного файла, с увеличением уровня сигнала

Спектрограмма выходного файла представлена на рис. 2.17.

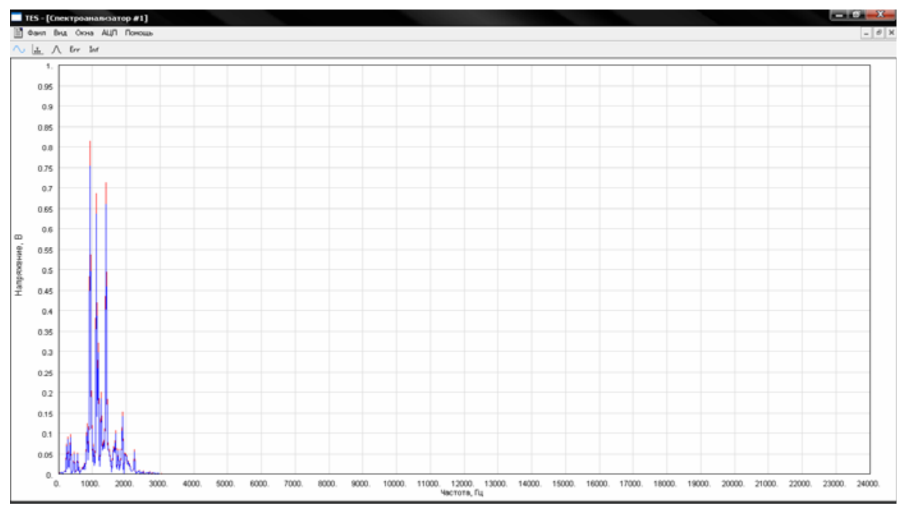


Рис. 2.17 Спектр отфильтрованного сигнала

3. Реализация БИХ-фильтра на DSK5510 для фильтрации звукового сигнала в реальном времени

1) Сконфигурировать ИСР CCS для работы с платой DSK5510.
2) Создать проект «exp_2» аналогичный проекту «exp_2» из работы №3.
3) Заменить файл «filter.h» из проекта «exp_2» работы №3 на файл, разработанный в этой работе.

А. Фильтр низкой частоты с нормальной частотой среза 0.1

1. Установить соответствующие коэффициенты для фильтрации.
2. Соединить линейные входы и выходы платы DSK5510 и компьютера, подобно тому, как это было сделано в работе № 3 и соответствующем разделе.
3. Скомпилироваться и запустить проект на исполнение.

4. Поставить на воспроизведение любой звуковой файл.

5. Запустить программу «TES2» и построить спектрограмму, аналогичную рис. 3.1.

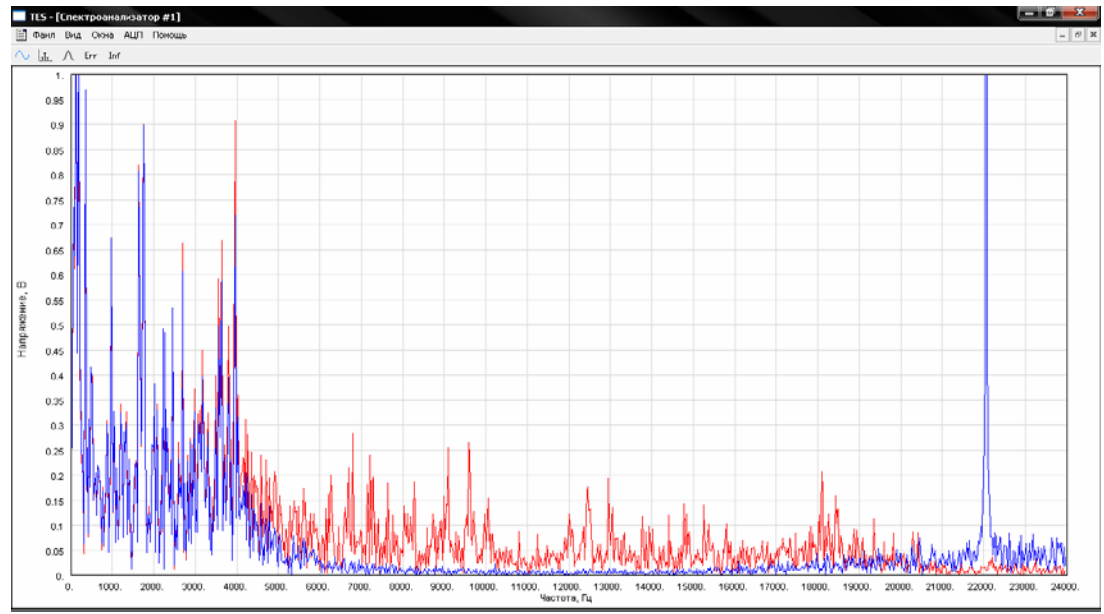


Рис. 3.1 Спектр исходного и отфильтрованного сигнала

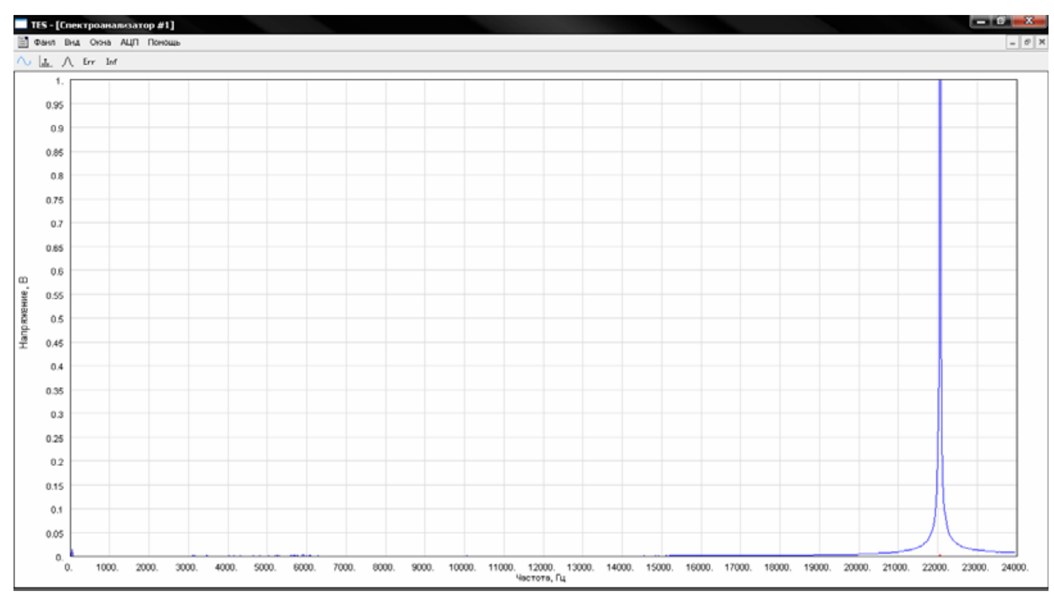


Рис. 3.2 Шум квантования на частоте 44 кГц

B. Фильтр низкой частоты с нормальной частотой среза 0.2

1. Установить соответствующие коэффициенты для фильтрации.
2. Скомпилироваться и запустить проект на исполнение.
3. Поставить на воспроизведение любой звуковой файл.
4. Запустить программу «TES2» и построить спектрограмму, аналогичную рис. 3.3.

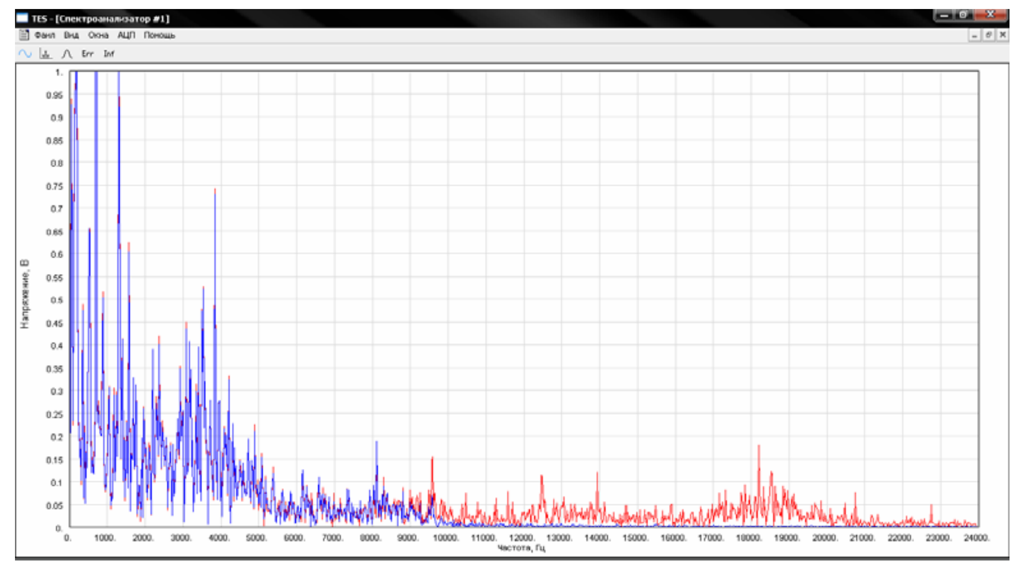


Рис. 3.3. Спектр исходного и отфильтрованного сигнала.

С. Полосовой фильтр с нормальной частотой среза 0.165 – 0.33

1. Установить соответствующие коэффициенты для фильтрации.
2. Скомпилироваться и запустить проект на исполнение.
3. Поставить на воспроизведение любой звуковой файл.
4. Запустить программу «TES2» и построить спектрограмму, аналогичную рис. 3.4.

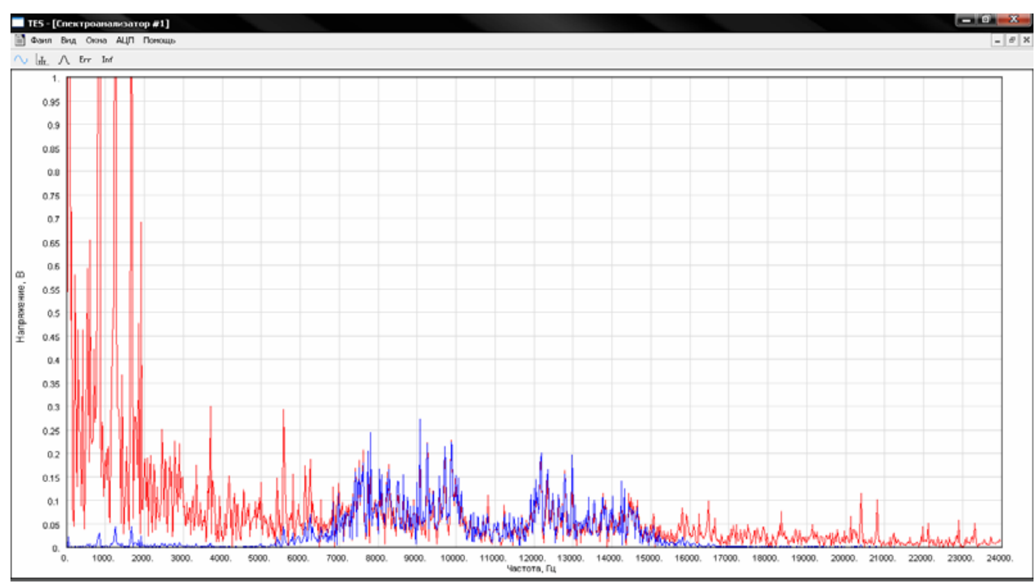


Рис. 3.4. Спектр исходного и отфильтрованного сигнала

При реализации полосового фильтра в реальном времени, нет необходимости прибавлять к выходному сигналу постоянной составляющей, т. к. АЦП DSK5510 выдает как отрицательные, так и положительные значения.

Индивидуальные задания студентам выдаются во время лабораторной работы преподавателем.

ОЦЕНКА РАБОТЫ СТУДЕНТОВ. РЕЙТИНГОВАЯ СИСТЕМА

1. Общие положения. Рейтинговая система оценки знаний студентов по дисциплине (далее – рейтинговая система) представляет собой комплекс организационных, учебных и контрольных мероприятий, базирующийся на учебно-методическом обеспечении всех видов деятельности по данному предмету.

Рейтинговая система включает непрерывный мониторинг учебной деятельности студентов, дифференциацию оценки успеваемости по различным видам деятельности в рамках конкретной дисциплины, график контрольных мероприятий, рейтинговую оценку знаний по дисциплине.

Основными целями введения рейтинговой системы являются:

1) стимулирование повседневной систематической работы студентов;

2) снижение роли случайных факторов при сдаче экзаменов и/или зачетов;

3) равномерное распределение учебной нагрузки студентов и преподавателей в течение семестра.

2. Организация рейтингового контроля успеваемости студентов дневной формы обучения. В рамках рейтинговой системы успеваемость студентов по дисциплине оценивается в ходе текущего и итогового контроля на экзамене или зачете и оценки выполнения лабораторных работ.

Текущий контроль осуществляется в течение семестра три раза. Первые два раза в дни проведения аттестации по университету и последний раз в день последнего занятия перед началом сессии. Текущий контроль будет осуществляться в виде двух аттестационных и одной итоговой письменных работ. За каждую из трех письменных работ будет выставлена оценка по 10-балльной шкале. Таким образом, максимальный суммарный балл за текущий контроль составляет 30 баллов (т. е. вклад текущего контроля в результирующую оценку 30%).

В одном семестре предусмотрены 4 лабораторные работы с максимальной оценкой 10 баллов за одну лабораторную работу. 10 баллов выставляется, если лабораторная работа выполнена в срок и защищена без ошибок не позднее следующего лабораторного занятия. Защита подразумевает под собой демонстрацию выполненной лабораторной работы и ответы на вопросы преподавателя. Преподаватель задает пять вопросов. За каждый правильный ответ выставляется 2 балла. Если преподаватель считает ответ неполным, он имеет право вместо 2 баллов выставить промежуточную оценку за ответ: 0.5, 1.0 или 1.5. Таким образом, максимальный суммарный балл за лабораторные работы составляет 40 баллов (т. е. вклад лабораторных работ в результирующую оценку 40%). В случае защиты лабораторной работы позже срока оценка снижается в два раза.

Итоговый контроль представляет собой зачет или экзамен в сессионный период по дисциплине в целом. Зачет или экзамен проводится по стандартным билетам, утвержденным заведующим кафедрой. Оценка за

данную работу выставляется следующим образом: в билете содержится 3 вопроса, ответ на каждый из них оценивается по 10-бальной шкале. Таким образом, максимальный суммарный балл за текущий контроль составляет 30 баллов (т. е. вклад итогового контроля в результирующую оценку 30%).

Результирующая оценка по дисциплине является интегральным показателем, формируемым на основе оценки знаний студента в ходе текущего и итогового контроля.

График контрольных мероприятий по дисциплинам учебного плана, согласуется с заведующим кафедрой в начале семестра и размещается на стенде кафедры. График предполагает равномерное распределение контрольных мероприятий в семестре.

Для студентов, пропустивших контрольное мероприятие по уважительной причине, подтвержденной документально, кафедра, отвечающая за преподавание данной дисциплины, устанавливает дополнительные сроки отчетности.

Для студентов, пропустивших контрольное мероприятие по неуважительной причине, вопрос о возможности выполнения данного вида работ решается заведующим кафедрой, отвечающей за преподавание данной дисциплины.

Студент, получивший неудовлетворительную оценку (ниже 4 баллов) по одному из видов текущего контроля, по решению кафедры может быть допущен (не более одного раза) к его повторному выполнению. Студент допускается к последующим контрольным мероприятия независимо от результатов предыдущих.

В случае получения средней оценки по текущему контролю без округления ниже 4-х баллов студент не допускается к экзамену (зачету).

3. Выставление оценок по рейтинговой системе. По результатам текущего контроля преподаватели заполняют ведомость текущей успеваемости по дисциплине. Ведомость может иметь следующий вид (табл. 1):

Таблица 1

Вариант оценки текущей успеваемости

№	Ф.И.О.	1 лаб. раб.	2 лаб. раб.	3 лаб. раб.	4 лаб. раб.	1 атт. раб.	2 атт. раб.	Итог. раб.	Оценка
1	Иванов	8	8	6	7	5	6	7	6,7
2	Петров	4	4	5	5	4	3	6	4,4
3	Сидоров	9	9	8	7	9	8	9	8,4
4	…								

Заполнение ведомости текущей успеваемости по дисциплине и ее подписание заведующим кафедрой производится не позднее, чем за день до начала экзаменационной сессии.

Ведомость текущей успеваемости по дисциплине хранится в делах кафедры, отвечающей за преподавание данной дисциплины.

Оценка за итоговую работу представляет собой среднее арифметическое трех оценок за каждый вопрос. Результат итогового контроля может иметь следующий вид (табл. 2).

Таблица 2

Вариант оценки итогового контроля

№	Ф.И.О.	1 вопрос	2 вопрос	3 вопрос	Оценка
1	Иванов	7	8	8	7,7
2	Петров	4	6	4	4,7
3	Сидоров	9	8	9	8,7
4	…				

Заполнение экзаменационной ведомости (внесение результирующей оценки за экзамен) производит преподаватель, принимающий экзамен по дисциплине путем выведения среднего арифметического из оценок текущей успеваемости и итогового контроля. В рассмотренном примере это будут следующие оценки (табл. 3).

Таблица 3

Выставление результирующей оценки

№	Ф.И.О.	Текущий контроль	Итоговый контроль	Оценка
1	Иванов	6,7	7,7	7,2=7
2	Петров	4,4	4,7	4,55=5
3	Сидоров	8,4	8,7	8,55=9
4	…			

Округление результирующей оценки до целого числа происходит по правилам математического округления. Если дисциплина в семестре заканчивается зачетом, то вместо 10-балльной оценки в зачетную ведомость записывается «зачтено» в случае если получена положительная оценка без округления.

В зачетку выставляется результирующая оценка при условии успешной сдачи экзамена или зачета.

4. Организация рейтингового контроля успеваемости студентов заочной формы обучения. Для студентов заочной формы обучения оценка успеваемости производится следующим образом:

1) выставляется оценка по 10-бальной шкале за защиту контрольной работы;

2) выставляется оценка по 10-бальной шкале за каждый вопрос итоговой работы (зачет или экзамен);

3) выводится среднее арифметическое между всеми четырьмя оценками, которое и будет итоговой оценкой;

4) если дисциплина в семестре заканчивается зачетом, то вместо 10-балльной оценки в зачетную ведомость записывается «зачтено» в случае, если получена положительная оценка без округления.

ЛИТЕРАТУРА

Основная литература

1. Таненбаум, Э. Архитектура компьютера / Э. Таненбаум. – 4-е изд. – СПб.: Питер, 2003.

2. Гук, М. Процессоры Intel: от 8086 до Pentium IV / М. Гук. – СПб: Питер, 2003.

3. Киселев, А. А. Современные микропроцессоры / А. А. Киселев, Б. Б. Корнеев. – СПб.: BHV, 2003.

4. Брам, П. Микропроцессор 80386 и его программирование / П. Брамм, Д. Брамм. – М.: Мир, 1990.

5. Хамахер, К. Организация ЭВМ / К. Хамахер, З. Вранешич, С. Заки. – 5-е изд. – СПб.: Питер, 2003.

6. Морисита, И. Аппаратные средства микроЭВМ: пер. с япон. / И. Морисита. – М.: Мир, 1988.

7. Щелкунов, Н. Н. Микропроцессорные средства и системы / Н. Н. Щелкунов, А. П. Дианов. – М.: Радио и связь, 1989.

8. Морс, С. П. Архитектура микропроцессора 80286: пер. с англ. / С. П. Морс, Д. Д. Алберт. – М.: Радио и связь, 1990.

9. Фрир, Дж. Построение вычислительных систем на базе перспективных микропроцессоров: пер. с англ. / Дж. Фрир. – М.: Мир, 1990.

10. Гук, М. Процессоры Pentium II, Pentium Pro и просто Pentium / М. Гук. – СПб.: ПитерКом, 1999.

Дополнительная литература

1. Лю, Ю-Чжен Микропроцессоры семейства 8086/8088. Архитектура, программирование и проектирование микрокомпьютерных систем: пер. с англ. / Ю-Чжен Лю, Г. Гибсон. – М.: Радио и связь, 1987.

2. Казаринов, Ю. М. Микропроцессорный комплект К1810: Структура, программирование, применение: Справочная книга / Ю. М. Казаринова. – М.: Высшая школа, 1990.

3. Бердышев, Е. Технология MMX. Новые возможности процессоров P5 и P6 / Е. Бердышев. – М.: ДИАЛОГ-МИФИ, 1998.

4. Коршуна, И. В. Современные микроконтроллеры: Архитектура, средства проектирования, примеры применения, ресурсы сети Интернет/© «Телесистемы» / И. В. Коршуна. – М.: Аким, 1998.

5. Ланнэ. А. А. Цифровой процессор обработки сигналов TMS320C10 и его применение / А. А. Ланнэ. – Л.: ВАС, 1990.

Учебное издание

ПАВЛОВЕЦ Павел Викторович

МИКРОПРОЦЕССОРНЫЕ СРЕДСТВА И СИСТЕМЫ

Учебно-методический комплекс
для студентов специальности 1-40 02 01
«Вычислительные машины, системы и сети»

Редактор *Т. Н. Лупенько*

Дизайн обложки *В. А. Виноградовой*

Подписано в печать 13.04.11. Формат 60x84 1/16. Бумага офсетная. Ризография.
Усл. печ. л. 21,11. Уч.-изд. л. 17,52. Тираж 50 экз. Заказ 277.

Издатель и полиграфическое исполнение:
учреждение образования «Полоцкий государственный университет».

ЛИ № 02330/0548568 от 26.06.2009 ЛП № 02330/0494256 от 27.05.2009

Ул. Блохина, 29, 211440, г. Новополоцк.