

Inside the Machine

Иллюстрированное введение в

Микропроцессоры и компьютерная архитектура

Джон Стоукс

ars technica library



ВНУТРИ МАШИНЫ

Inside the Machine

An Illustrated Introduction to
Microprocessors and Computer Architecture

Jon Stokes



NO STARCH
PRESS

Сан-Франциско

ВНУТРИ МАШИНЫ. Авторские права © 2007, Джон Стоукс.

Все права защищены. никакая часть этой работы не может быть воспроизведена или передана в любой форме или любыми средствами, электронными или механическими, включая фотокопирование, запись или любую систему хранения или поиска информации, без предварительного письменного разрешения владельца авторских прав и издателя.

Напечатано в Канаде

10 09 08 07 2 3 4 5 6 7 8 9

ISBN-10: 1-59327-104-2

ISBN-13: 978-1-59327-104-6

Издатель: Уильям Поллок

Монтажер-постановщик: Элизабет Кэмпбелл

Дизайн обложки: Octopod Studios

Редактор по развитию: Уильям Поллок

Редакторы: Сара Лемэр, Меган Дунчак

Композитор: Райли Хоффман

Корректор: Стефани Провинс

Индексатор: Нэнси Гюнтер

Для получения информации о распространителях книг или переводах обращайтесь напрямую в No Starch Press, Inc.:

Нет крахмала Press, Inc.

555 De Haro Street, Suite 250, Сан-Франциско, Калифорния 94107

телефон: 415.863.9900; факс: 415.863.9950; info@nostarch.com; www.nostarch.com

Данные каталогизации публикаций Библиотеки Конгресса

Стоукс, Джон

Внутри машины: иллюстрированное введение в микропроцессоры и компьютерную архитектуру / Джон Стоукс.

П. см.

Включает индекс.

ISBN-13: 978-1-59327-104-6

ISBN-10: 1-59327-104-2

1. Архитектура компьютера. 2. Микропроцессоры -- Дизайн и конструкция. I. Название.

TK7895.M5C76 2006 г.

621,39'2 -- dc22

2005037262

No Starch Press и логотип No Starch Press являются зарегистрированными товарными знаками No Starch Press, Inc. Другие названия продуктов и компаний, упомянутые здесь, могут быть товарными знаками соответствующих владельцев. Вместо того, чтобы использовать символ товарного знака при каждом появлении названия товарного знака, мы используем названия только в редакционных целях и в интересах владельца товарного знака без намерения нарушить права на товарный знак.

Информация в этой книге распространяется на условиях «как есть» без каких-либо гарантий. Несмотря на то, что при подготовке этой работы были приняты все меры предосторожности, ни автор, ни No Starch Press, Inc. не несут никакой ответственности перед каким-либо физическим или юридическим лицом в отношении любых убытков или ущерба, причиненных или предположительно вызванных прямо или косвенно содержащейся в нем информацией.

На фотографии в центре обложки показана небольшая часть кристалла микропроцессора Intel 80486DX2 при 200-кратном оптическом увеличении. Большинство видимых элементов — это верхние металлические соединительные слои, которые соединяют вместе большинство встроенных компонентов.

Фото на обложке Мэтта Бритта и Мэтта Гиббса.

Моим родителям, которые привили мне любовь к учебе и образованию, а
также бабушке и дедушке, которые оплатили счет.

КРАТКОЕ СОДЕРЖАНИЕ

Предисловие	XVIII
Благодарности	XVII
Введение	XIX
Глава 1: Основные концепции вычислений	1
Глава 2: Механизм выполнения программы	19
Глава 3: Конвейерное выполнение	35
Глава 4: Суперскалярное выполнение	61
Глава 5: Intel Pentium и Pentium Pro	79
Глава 6. Процессоры PowerPC: серии 600, серии 700 и 7400.....	111
Глава 7. Процессоры Intel Pentium 4 и Motorola G4e: подходы и философия проектирования	137
Глава 8. Процессоры Intel Pentium 4 и Motorola G4e: серверная часть	161
Глава 9: 64-битные вычисления и x86-64	179
Глава 10: G5: IBM PowerPC 970	193
Глава 11: Понимание кэширования и производительности	215
Глава 12. Процессоры Intel Pentium M, Core Duo и Core 2 Duo	235
Библиография и рекомендуемая литература	271
Указатель	275

СОДЕРЖАНИЕ ПОДРОБНО

ПРЕДИСЛОВИЕ	xv
БЛАГОДАРНОСТИ	xvii
ВВЕДЕНИЕ	xix
1	
ОСНОВНЫЕ ВЫЧИСЛИТЕЛЬНЫЕ ПОНЯТИЯ	1
Калькуляторная модель вычислений	2
Компьютерная модель файл-клерка	3
Компьютер с хранимой программой	4
Уточнение модели файл-клерка	6
Регистрационный файл	7
Оперативная память: когда одними регистрами обойтись	8
Модель файл-клерка пересмотрена и расширена	9
Пример: сложение двух чисел	10
Пристальный взгляд на поток кода: программа	11
Общие типы инструкций	11
Базовая архитектура DLW-1 и формат арифметических инструкций	12
Более пристальный взгляд на доступ к памяти: регистровый и немедленный	14
Непосредственные значения	14
Регистровая адресация	16
2	
МЕХАНИКА ВЫПОЛНЕНИЯ ПРОГРАММЫ	19
Коды операций и машинный язык	19
Машинный язык на DLW-1	20
Двоичное кодирование арифметических инструкций	21
Двоичное кодирование инструкций доступа к памяти	23
Преобразование программы-примера на машинный язык	25
Модель программирования и ISA	26
Модель программирования	26
Регистр команд и счетчик команд	26
Выборка инструкций: загрузка регистра инструкций	28
Запуск простой программы: цикл выборки-выполнения	28
Часы	29
Инструкции по отделениям	30
Безусловный переход	30
Условная ветвь	30
Экскурс: загрузка	34

3**ИСПОЛНЕНИЕ ТРУБОПРОВОДА****35**

Жизненный цикл инструкции	36
Основная последовательность инструкций	
38 Объяснение конвейерной обработки	40
Применение аналогии	43
Неконвейерный процессор	
Конвейерный процессор	43
Ускорение за счет конвейерной обработки	45
Время выполнения программы и скорость выполнения	48
Взаимосвязь между скоростью выполнения и временем выполнения программы	51
Пропускная способность инструкций и конвейер S высокие	52
Задержка выполнения инструкций и зависание конвейера	53
Ограничения конвейерной обработки	58

4**СУПЕРСКАЛЬНОЕ ИСПОЛНЕНИЕ****61**

Суперскалярные вычисления и IPC	64
возможностей суперскалярной обработки с помощью исполнительных модулей	
Основные числовые форматы и компьютерная арифметика	65
устройства	66
памяти	67
Модули доступа к ISA	69
Краткая история ISA	
ISA	71
на программный	73
проектирования	74
Перенос сложности с аппаратного на данных	
опасности	74
файл	76
Структурные опасности	
Регистрационный файл	77
Опасности при управлении	

5**INTEL PENTIUM И PENTIUM PRO****79**

Оригинальный Pentium	80
Кэши	81
Pentium	82
Модуль ветвления и предсказание ветвления	
82	85
Pentium сзади Конец	87
x86 накладных расходов на Pentium	91
Резюме: Pentium в историческом контексте	92
Микроархитектура Intel P6: Pentium Pro	
Отделение передней части от задней	93
94 Конвейер P6	
100 Предсказание ветвей на P6	102
Серверная часть P6	
102 CISC, RISC и трансляция набора команд	
103 Блок декодирования инструкций микроархитектуры P6	106
107 Резюме: устаревших версий архитектуры x86 на P6	
107 Заключение	107

6

ПРОЦЕССОРЫ POWERPC: СЕРИИ 600, СЕРИИ 700 И 7400 Краткая история	111
PowerPC	112
PowerPC 601	112
Конвейер и интерфейс модели 601	113
Бэкэнд 601 115 Еще раз о задержке и	
пропускной способности 117 Резюме: 601 в	
историческом контексте 118 PowerPC 603 и	
603e 118 Серверная часть модели	
603e 119 Внешний интерфейс модели 603e, окно инструкций и предсказание	
ветвления	

7

INTEL PENTIUM 4 ПРОТИВ. MOTOROLA G4E: ПОДХОДЫ И КОНСТРУКТИВНАЯ ФИЛОСОФИЯ	137
Пристрастие Pentium 4 к скорости	138
Общие подходы и принципы проектирования процессоров Pentium 4 и G4e	141
Обзор архитектуры и конвейера G4e	144
Этапы 1 и 2: выборка инструкций	145
Этап 3: Декодирование/ Отправка	145
Этап 4: Выпуск	146
Выполнение	146
и обратная запись	147
Предсказание ответвлений на G4e и Pentium 4	148
147. Обзор архитектуры Pentium....	148
Расширение окна инструкций	149
Кэш трассировки	149
Обзор линейки процессоров Pentium 4	155
Этапы 1 и 2:	
Трассировка указателя следующей инструкции в кэше	155
Этапы 3 и 4: выборка кэша трассировки	155
Этап 5: Привод	155
Этапы с 6 по 8: Выделение и переименование (ROB)	155
Этап 9: Очередь	156
Этапы с 10 по 12: Расписание	156
Этапы 13 и 14: Выпуск	157

Этапы 15 и 16: Регистрационные файлы	158
Этап 17: Выполнение	158
Этап 18:	
Флаги	158
Этап 19 : Проверка	
ветки	158
Этап 20:	
Привод	158
Этапы 21 и далее:	
завершить и зафиксировать	158
Окно инструкций Pentium 4	159

4

8**INTEL PENTIUM 4 ПРОТИВ. MOTOROLA G4E: ЗАДНЯЯ ЧАСТЬ****161**

Некоторые замечания о форматах операндов	161
Целочисленные исполнительные блоки	163
Внутренние интерфейсы G4e: ускорение работы с обычными	
случаями	163
Внутренние устройства Pentium 4: ускорьте	
общий процесс в два раза	164
Блоки с плавающей запятой	
(FPU)	165
FPU 166	
G4e 166	FPU процессора Pentium
4 167	
Заключительные замечания по	
FPU процессоров G4e и Pentium 4	168
Векторные исполнительные	
устройства	168
Краткий обзор	
вектора Вычисления	168
Еще раз о векторах: набор	
инструкций AltiVec	169
Векторные операции	
AltiVec	170
VU G4e: правильное выполнение	
SIMD	173
MMX от	
Intel	174
SSE и	
SSE2	175
Векторный блок Pentium 4:	
Алфавитный бульон делается быстро	176
Увеличение плавающих -Точечная	
производительность с SSE2	177
Выходы	

9**64-БИТНЫЕ ВЫЧИСЛЕНИЯ И X86-64****179**

Intel IA-64 и AMD x86-64	180
Почему 64	
биты?	181
Что такое 64-битные	
вычисления?	181
Текущие 64-разрядные	
приложения	183
Динамический	
диапазон	183
Преимущества	
расширенного динамического диапазона, или Как существующий рынок 64-битных	
вычислений использует 64-битные целые числа	184
Виртуальное адресное	
пространство по сравнению с физическим адресным пространством	
185	
Преимущества 64-битного адреса	186
64-битная альтернатива: x86-64	187
Расширенные регистры	187
Дополнительные регистры	188
Переключение режимов	189
старое	192
Заключение	192

10

G5: IBM POWERPC 970

193

Обзор: философия дизайна	194 Кэши и внешний
интерфейс	194 Предсказание
ветвления	195 Компромисс: декодирование,
взлом и формирование группы	196 Правила отправки 970-х
годов	198 Предварительное декодирование и
групповая отправка	199 Некоторые
предварительные выводы о схеме групповой отправки 970-х	199 Серверная
часть PowerPC 970	200 Целочисленный блок,
блок регистра условий и блок ответвлений	201 Целочисленные
единицы измерения не полностью симметричны	201
Целочисленные единицы задержки и пропускная способность	202 Серверная
202 CRU	202 Предварительные
выводы о целочисленной производительности модели 970	203 Единицы
загрузки-склада	203 Передняя
шина	204 Модули с плавающей
запятой	205 Векторные вычисления на PowerPC
970	206 Проблема с плавающей запятой
Очереди	209 Целочисленные и очереди
задач загрузки-сохранения	210 Очереди выдачи
BU и CRU	210 Очереди задач
вектора	211 Последствия схемы
групповой диспетчеризации модели 970 для производительности	211
Выводы	213

11

ПОНИМАНИЕ КЭШИРОВАНИЯ И ПРОИЗВОДИТЕЛЬНОСТИ

215

Основы кэширования	215 Кэш 1-го
уровня	217 Кэш 2-го
уровня	218 Пример: краткое путешествие
байта по иерархии памяти	218 Кэш-
промахи	219 Местность
отсчета	220 Пространственная локализация
данных	220 Пространственная
локализация кода	221 Временная
локализация кода и данных	222 Населенный пункт:
выводы	222 Организация кэша: блоки и
блочные фреймы	223 ОЗУ
тегов	224 Полностью ассоциативное
картографирование	224 Прямое
сопоставление	225 Ассоциативное
сопоставление N-Way Set	226 Четырехстороннее
ассоциативное сопоставление наборов	226 Двустороннее
ассоциативное сопоставление наборов	228 Двухстороннее
и прямое сопоставление	229 Двухстороннее
и четырехстороннее	229 Ассоциативность:
Выводы	229

Новый взгляд на временную и пространственную локализацию: политика замены/выселения и Размеры блоков	230
Типы политики замены/выселения	230
Блок	231
Политики записи: сквозная и обратная запись	232
Выводы	233

12

Кодовые названия и торговые марки процессоров INTEL PENTIUM M, CORE DUO И CORE 2	
DUO 235	236 Восхождение к власти-
Эффективные вычисления	237 Удельная
мощность	237 Динамическая
плотность мощности	237 Плотность
статической мощности	238 Pentium
M	239 Фаза
выборки	239 Фаза декодирования:
слияние микроопераций	240 Предсказание
переходов	244 Модуль
исполнения стека	246 Конвейер
и серверная часть	246 Резюме: Pentium
M в историческом контексте	246 Core Duo/
Solo	247 Линия Intel идет вперед
Многоядерный	247 Улучшения
Core Duo	251 Резюме: Core Duo в
историческом контексте	254 Core 2
Duo	254 Фаза
выборки	256 Фаза
декодирования	257 Конвейер
Core	258 Серверная часть
Core	258 Усовершенствования
векторной обработки	262 Устранение
неоднозначности памяти: поток результатов Версия	
Спекулятивное исполнение	264
Резюме: Core 2 Duo в историческом контексте	270
БИБЛИОГРАФИЯ И РЕКОМЕНДУЕМАЯ ЧТЕНИЕ	271
Общий	271 PowerPC ISA и
расширения	271 PowerPC 600 Серийные
процессоры	271 Процессоры PowerPC
серий G3 и G4	272 IBM PowerPC 970 и
POWER	272 x86 ISA и
расширения	273 Pentium и семейство
P6	273 Пентиум
4	274 Pentium, Core и Core
ресурсы	274
ИНДЕКС	275

ПРЕДИСЛОВИЕ

«Целью вычислений является понимание, а не числа».
— Ричард В. Хэмминг (1915–1998).

Когда математик и пионер вычислительной техники Ричард Хэмминг сформулировал это правило в 1962 году, эра цифровых вычислений все еще находилась в зачаточном состоянии. Во всем мире существовало всего около 10 000 компьютеров; каждый был большим и дорогим, и каждый требовал бригады инженеров по техническому обслуживанию и эксплуатации. Чтобы получить результаты от этих гигантских машин, нужно было усердно вводить длинные цепочки чисел, ждать, пока машина выполнит свои вычисления, а затем интерпретировать полученную массу единиц и нулей. Этот утомительный и кропотливый процесс побудил Хэмминга напомнить своим коллегам, что множество чисел, с которыми они работали ежедневно, были лишь средством для достижения гораздо более высокой и часто нечисловой цели: более глубокого понимания окружающего мира.

В сегодняшнюю пост-интернет-эпоху сотни миллионов людей регулярно используют компьютеры не только для получения информации, но и для бронирования авиабилетов, игры в покер, составления фотоальбомов, поиска друзей и выполнения любых других видов человеческой деятельности от мирское к возвышенному. В резком контрасте с

как это было 40 лет назад, опыт использования компьютера для выполнения математических операций с большими наборами чисел довольно непривычен для многих пользователей, которые третят лишь очень небольшую часть своего компьютерного времени на явное выполнение арифметических операций. В популярных операционных системах от Microsoft и Apple маленькое приложение-калькулятор спрятано где-то в папке, и большинство пользователей обращаются к нему лишь изредка, если вообще обращаются. Это маленькое, редко используемое приложение-калькулятор является идеальной метафорой скрытой идентичности современного компьютера как устройства для перетасовки чисел.

Эта книга нацелена на то, чтобы заново представить компьютер как вычислительное устройство, которое слой за слоем выполняет чудесные ловкие движения рук, чтобы скрыть от пользователя быстрый поток чисел внутри машины. Первые несколько глав знакомят с основными понятиями вычислительной техники, а в последующих главах представлен ряд более сложных объяснений, основанных на реальных аппаратных средствах, которые показывают, как инструкции, данные и численные результаты перемещаются через компьютеры, которые люди используют каждый день. В конце концов, книга «Внутри машины» призвана дать читателю промежуточные знания о том, как функционируют различные микропроцессоры и как они взаимодействуют друг с другом с точки зрения дизайна и производительности.

В конце концов, я попытался написать книгу, которую хотел бы прочитать, будучи студентом бакалавриата компьютерной инженерии: книгу, которая собирает воедино части в виде общей картины, но при этом содержит достаточно подробной информации, чтобы дать твердое понимание основных принципов проектирования, лежащих в основе современных микропроцессоров. Я надеюсь, что сочетание экспозиции, истории и архитектурной «сравнительной анатомии» в «Внутри машины» поможет достичь этой цели.

БЛАГОДАРНОСТИ

Эта книга представляет собой квинтэссенцию и адаптацию моих технических статей и новостных репортажей для Ars Technica за более чем восемь лет, и поэтому она отражает идеи и информацию, предоставленные мне многими тысячами читателей, которые не пожалели времени, чтобы связаться со мной со своими отзывами. Журналисты, профессора, студенты, профессионалы отрасли и, во многих случаях, некоторые ученые и инженеры, работавшие над процессорами, описанными в этой книге, внесли свой вклад в текст на этих страницах, и я хочу поблагодарить этих корреспондентов за их исправления, разъяснения и терпеливые объяснения. В частности, я хотел бы поблагодарить сотрудников IBM за их помощь в написании статей, которые послужили материалом для части книги, посвященной PowerPC 970. Я также хотел бы поблагодарить Intel Corp. и Джорджа Альфса в частности, за ответы на мои вопросы о процессорах, описанные в главе 12. (Все ошибки — мои собственные.)

Я хочу поблагодарить Билла Поллока из No Starch Press за согласие опубликовать Внутри машины и за терпеливое руководство моей первой книгой. Среди других сотрудников No Starch Press, за которых следует поблагодарить Элизабет Кэмпбелл (производственный редактор), Сару Лемэр (редактор), Райли Хоффман (композитор), Стефани Провинс (корректор) и Меган Дунчак.

Я хотел бы выразить особую благодарность сотрудникам Ars Technica и участникам форума сайта, многие из которых предоставили мне конструктивную критику, поддержку и образование, без которых эта книга была бы невозможна. Спасибо также моим техническим специалистам по предварительному прочтению, особенно Ли Харрисону и Хольгеру Беттагу, оба из которых предоставили бесценные советы и отзывы по предыдущим черновикам этого текста. Наконец, я хотел бы поблагодарить мою жену Кристину за ее терпение и любящую поддержку, помогавшую мне закончить этот проект.

Джон Стоукс
Чикаго, 2006 г.

ВВЕДЕНИЕ

Inside the Machine — это введение в компьютеры, призванное заполнить пробел, существующий между классическими, но более сложными введениями в компьютерную архитектуру, такими как работы Джона Л. Хенnessи и Дэвида А.

Популярные учебники Паттерсона и растущая масса работ, которые просто слишком просты для мотивированных читателей-неспециалистов. Читатели, имеющие некоторый опыт работы с компьютерами и даже с минимальным опытом написания сценариев или программирования, должны закончить книгу «Внутри машины», имея полное и глубокое понимание высокогоуровневой организации современных компьютеров. Если они того пожелают, такие читатели будут хорошо подготовлены для изучения более сложных произведений, таких как вышеупомянутая классика, либо самостоятельно, либо в рамках формальной учебной программы.

Сравнительный подход книги, описанный ниже, вводит новые конструктивные особенности, сравнивая их с более ранними функциями, предназначенными для решения тех же проблем. Таким образом, начинающим и продвинутым читателям рекомендуется читать главы по порядку, поскольку каждая глава предполагает знакомство с концепциями и конструкциями процессоров, представленными в предыдущих главах.

Более продвинутые читатели, уже знакомые с некоторыми рассмотренными процессорами, обнаружат, что отдельные главы могут быть самостоятельными. Широкое использование заголовков и подзаголовков в книге означает, что ее также можно использовать в качестве общего справочника по описанным процессорам, хотя это не та цель, для которой она была разработана.

Первые четыре главы «Внутри машины» посвящены закладке концептуальной основы для последующих глав, посвященных изучению реальных микропроцессоров. В этих главах используется упрощенный пример процессора, DLW, для иллюстрации базовых и промежуточных понятий, таких как различие между инструкциями и данными, программирование на языке ассемблера, суперскалярное выполнение, конвейерная обработка, модель программирования, машинный язык и т. д.

Средняя часть книги состоит из подробных исследований двух популярных линеек процессоров для настольных ПК: линейка Pentium от Intel и линейка PowerPC от IBM и Motorola. В этих главах читатель знакомится с хронологическим развитием каждой линейки процессоров, описывая эволюцию обсуждаемых микроархитектур и архитектур набора команд. Попутно вводятся и изучаются более продвинутые концепции, такие как спекулятивное выполнение, векторная обработка и трансляция набора команд, посредством обсуждения одного или нескольких реальных процессоров.

На протяжении всей средней части книги общим подходом является то, что можно было бы назвать «сравнительной анатомией», в которой новые функции каждого нового процессора объясняются с точки зрения того, чем они отличаются от аналогичных функций предшественников и/или конкурентов. Кульминацией сравнительной части книги являются главы 7 и 8, которые состоят из подробных сравнений двух совершенно разных и очень важных процессоров: Intel Pentium 4 и Motorola MPC7450 (широко известного как G4e).

После краткого введения в 64-разрядные вычисления и 64-разрядных расширений популярной архитектуры набора команд x86 в главе 9 микроархитектура первого массового 64-разрядного процессора IBM PowerPC 970 рассматривается в главе 10. Это исследование процессора 970, большая часть которого также напрямую применима к процессору IBM для мейнфреймов POWER4, завершает обзор процессоров PowerPC в книге.

В главе 11 рассказывается об организации и функционировании иерархии памяти почти во всех современных компьютерах.

Внутри машины Заключительная глава посвящена подробному изучению процессоров Intel последнего поколения: Pentium M, Core Duo и Core 2 Duo. В этой главе содержится наиболее подробное обсуждение этих процессоров, доступное в Интернете или в печати, а также некоторые новые сведения, которые не публиковались до печати этой книги.

1

ОСНОВНЫЕ ВЫЧИСЛИТЕЛЬНЫЕ ПОНЯТИЯ

Современные компьютеры бывают всех форм и размеров, и они помогают нам в миллионе различных типов задач, начиная от серьезных, таких как управление воздушным движением и исследования рака, до не очень серьезных, таких как компьютерные игры. и ретушь фотографий. Но какими бы разнообразными ни были компьютеры по своим внешним формам и способам их использования, все они поразительно похожи по основным функциям. Все они полагаются на ограниченный набор технологий, которые позволяют им творить бесчисленные чудеса, которых мы привыкли ожидать от них.

Сердцем современного компьютера является микропроцессор . называемый центральным процессором (ЦП) — крошечный квадратный кусочек кремния, на котором выгравирована микроскопическая сеть ворот и каналов, по которым течет электричество. Эта сеть вентилей (транзисторов) и каналов (проводов или линий) представляет собой очень маленькую версию схемы, которую мы все видели, открывая пульт от телевизора или старый радиоприемник. Короче говоря, микропроцессор — это не просто «сердце» современного компьютера — это сам по себе компьютер. Как только вы поймете, как работает этот крошечный компьютер, у вас будет

глубокое понимание фундаментальных концепций, лежащих в основе всех современных вычислений, от вышеупомянутой системы управления воздушным движением до кремниевого мозга, управляющего тормозами роскошного автомобиля.

Эта глава познакомит вас с микропроцессором, и вы начнете чтобы почувствовать, насколько просты компьютеры на самом деле. Вам нужно освоить лишь несколько основных понятий, прежде чем вы приступите к изучению микропроцессорных технологий, подробно описанных в последующих главах этой книги.

С этой целью в этой главе строится общая концептуальная основа, на которую я повешу технические детали, описанные в остальной части книги. И новичкам в изучении компьютерной архитектуры, и более продвинутым читателям рекомендуется прочитать эту главу от начала до конца, потому что содержащиеся в ней абстракции и обобщения представляют собой большие концептуальные «коробки», в которые я позже поместлю особенности конкретных архитектур.

Калькуляторная модель вычислений

Рисунок 1-1 представляет собой абстрактное графическое представление того, что делает компьютер. Короче говоря, компьютер принимает на вход поток инструкций (код) и поток данных, а на выходе выдает поток результатов. Для целей нашего первоначального обсуждения мы можем обобщить, сказав, что кодовый поток состоит из различных типов арифметических операций, а поток данных состоит из данных, над которыми работают эти операции. Таким образом, поток результатов состоит из результатов этих операций. Можно также сказать, что поток результатов начинает течь, когда операторы в кодовом потоке выполняются над операндами в потоке данных.

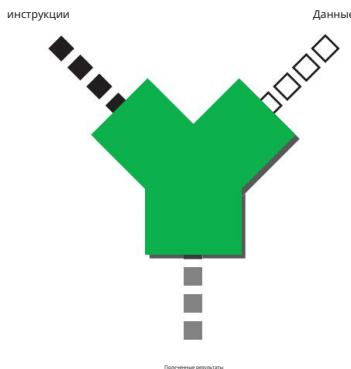


Рисунок 1-1: Простое представление компьютера общего назначения

ПРИМЕЧАНИЕ. Рисунок 1-1 — это моя собственная вариация традиционного способа представления производительности процессора. арифметико-логическое устройство (АЛУ) — часть процессора, выполняющая операции сложения, вычитания и т. д. чисел. Однако вместо того, чтобы показывать два операнда, поступающих в верхние порты, и результат, выходящий из нижнего порта (как это принято в литературе), я изобразил потоки кода и данных, входящие в верхние порты, и поток результатов, выходящий из нижнего порта.

Чтобы проиллюстрировать это, представьте, что один из этих маленьких черных ящиков в потоке кода на рис. Рисунок 1-2.



Рисунок 1-2: Инструкции объединяются с данными для получения результатов

Вы можете думать об этих черно-белых прямоугольниках как о клавишиах калькулятора: белые клавиши — это числа, а черные — операторы, а серые прямоугольники — это результаты, которые появляются на экране калькулятора. Таким образом, два входных потока (кодовый поток и поток данных) представляют собой последовательности нажатий клавиш (клавиши арифметических операций и цифровые клавиши), а выходной поток представляет результирующую последовательность чисел, отображаемых на экране калькулятора.

Описанные выше простые вычисления представляют собой то, что, как мы интуитивно думаем, делают компьютеры: подобно карманному калькулятору, компьютер принимает числа и арифметические операторы (такие как +, -, ÷, × и т. д.) в качестве входных данных, выполняет запрошенную операцию, а затем отображает результаты. Эти результаты могут быть в форме значений пикселей, которые составляют визуализированную сцену в компьютерной игре, или они могут быть выражены в долларах в финансовой электронной таблице.

Компьютерная модель файлового клерка

«Калькуляторная» модель вычислений, хотя и полезная во многих отношениях, не является единственным и даже не лучшим способом осмысления того, что делают компьютеры. В качестве альтернативы рассмотрим следующее определение компьютера:

Компьютер — это устройство, которое перемещает числа с места на место, считывая, записывая, стирая и перезаписывая разные числа в разных местах в соответствии с набором входных данных, фиксированным набором правил для обработки этих входных данных и предшествующей историей всех входных данных, которые компьютер видел с момента последнего сброса, до тех пор, пока не будет выполнен заранее определенный набор критериев, которые заставят компьютер остановиться.

Мы могли бы, вслед за Ричардом Фейнманом, назвать эту идею компьютера как устройства для чтения, записи и модификации чисел моделью вычислений «файлового клерка» (в отличие от вышеупомянутой модели калькулятора). В модели файл-клерка компьютер обращается к большому (теоретически бесконечному) хранилищу последовательно расположенных чисел с целью изменения этого хранилища для достижения желаемого результата. Как только этот желаемый результат достигнут, компьютер останавливается, чтобы люди могли прочитать и интерпретировать теперь измененное хранилище.

Поначалу модель вычислений с файл-клерком может показаться вам не такой уж полезной, но по мере продвижения в этой главе вы начнете понимать, насколько она важна. Такой взгляд на компьютеры силен, потому что он делает акцент на конечном продукте вычислений, а не на самих вычислениях. В конце концов, цель компьютеров не только в абстрактных вычислениях, но и в получении пригодных для использования результатов из заданного набора данных.

ПРИМЕЧАНИЕ Те, кто изучал информатику, узнают в предыдущем описании начало обсуждения машины Тьюринга. Однако машина Тьюринга слишком абстрактна для наших целей, поэтому я не буду ее описывать. Описание, которое я разрабатываю здесь, ближе к классической модели вычислений с уменьшенным набором инструкций (RISC), где компьютер «фиксируется» вместе с хранилищем. Модель Тьюринга о компьютере как о подвижной головке чтения-записи (с таблицей состояний), перемещающейся по линейной «ленте», слишком далека от реальной аппаратной организации, чтобы ее можно было сбить с толку в данном обсуждении.

Другими словами, в вычислениях важно не то, что вы занимались математикой, а то, что вы начали с совокупности чисел, применили к ней последовательность операций и получили совокупность результатов. Эти результаты могут, опять же, представлять значения пикселей для отрендеренной сцены или моментального снимка окружающей среды в моделировании погоды. Действительно, идея о том, что компьютер — это устройство, преобразующее один набор чисел в другой, должна быть интуитивно понятна любому, кто когда-либо пользовался фильтром Photoshop. Как только мы поймем компьютеры не с точки зрения математики, которую они выполняют, а с точки зрения чисел, которые они перемещают и изменяют, мы можем начать получать более полную картину того, как они работают.

Короче говоря, компьютер — это устройство, которое считывает, изменяет и записывает последовательности чисел. Эти три функции — чтение, изменение и запись — — это три самые основные функции, которые выполняет компьютер, и все компоненты машины предназначены для помощи в их выполнении. Эта последовательность «чтение-изменение-запись» на самом деле присуща трем центральным пунктам нашего исходного определения файла-клерка компьютера. Вот последовательность, явно отображеная на определение файл-клерка:

Компьютер — это устройство, которое перемещает числа с места на место, считывая, записывая, стирая и перезаписывая разные числа в разных местах в соответствии с набором входных данных [чтение], фиксированным набором правил для обработки этих входных данных [изменить], и предыдущую историю всех входных данных, которые компьютер видел с момента его последней перезагрузки [записи], до тех пор, пока не будет выполнен заранее определенный набор критериев, которые заставят компьютер остановиться.

Это подводит итог тому, что делает компьютер. И, по сути, это все, что делает компьютер. Играете ли вы в игру или слушаете музыку, все, что происходит под капотом компьютера, вписывается в эту модель.

ПРИМЕЧАНИЕ Все это пока довольно просто, и я даже немножко повторил объяснение. Чтобы довести до ума базовую структуру чтения-изменения-записи всех компьютерных операций. Важно понять эту структуру в ее простоте, потому что по мере увеличения уровня сложности нашей вычислительной модели мы увидим, что эта структура повторяется на каждом уровне.

Компьютер с хранимой программой

Все компьютеры состоят как минимум из трех основных типов структур, необходимых для выполнения последовательности чтения-изменения-записи:

Хранилище

Сказать, что компьютер «читает» и «записывает» числа, означает, что существует по крайней мере одна структура, содержащая числа, из которой он считывает и

пишет в. На всех компьютерах есть место для хранения чисел — область памяти, которую можно читать и записывать.

Арифметико-логическое устройство (ALU)

Точно так же, когда говорят, что компьютер «изменяет» числа, подразумевается, что компьютер содержит устройство для выполнения операций с числами. Это устройство — АЛУ, часть компьютера, выполняющая арифметические операции (сложение, вычитание и т. д.) над числами из области памяти. Сначала числачитываются из памяти в порт ввода данных АЛУ. Оказавшись внутри ALU, они модифицируются с помощью арифметических вычислений, а затем записываются обратно в хранилище через выходной порт ALU.

На самом деле ALU — это трехпортовое устройство зеленого цвета в центре рис. 1-1. Обратите внимание, что ALU обычно не понимаются как имеющие порт ввода кода вместе с портом ввода данных и портом вывода результатов. Они, конечно, имеют линии ввода команд, которые позволяют компьютеру указать, какую операцию должно выполнять ALU с данными, поступающими на его порт ввода данных, поэтому изображение порта ввода кода на ALU на рисунке идентичен, он не вводит в заблуждение.

Автобус

Для перемещения чисел между ALU и хранилищем требуются некоторые средства передачи чисел. Таким образом, ALU считывает и записывает в область хранения данных с помощью шины данных, которая представляет собой сеть линий передачи для перемещения чисел внутри компьютера. Инструкции поступают в ALU через шину команд, но мы не будем рассматривать то, как инструкции поступают в ALU, до главы 2. Пока нас интересует только шина данных.

Кодовый поток на рис. 1-1 поступает в ALU в виде последовательности арифметических инструкций (сложение, вычитание, умножение и т. д.). Операнды для этих инструкций составляют поток данных, который проходит пошине данных из области хранения в ALU. По мере того, как ALU выполняет операции над входящими операндами, поток результатов выходит из ALU и возвращается в область хранения через шину данных. Этот процесс продолжается до тех пор, пока кодовый поток не перестанет поступать в ALU. Рисунок 1-3 расширяет Рисунок 1-1 и показывает область хранения.

Данные поступают в ALU из специальной области хранения, а откуда берется кодовый поток? Можно представить, что он исходит от клавиатуры какого-то человека, стоящего за компьютером и вводящего последовательность инструкций, каждая из которых затем передается на порт ввода кода ALU, или, возможно, что кодовый поток представляет собой заранее записанный список инструкций, которые передаются в ALU по одной инструкции за раз каким-либо ручным или автоматизированным механизмом. На рис. 1.3 кодовый поток представлен в виде заранее записанного списка инструкций, который хранится в специальной области хранения точно так же, как и поток данных, и современные компьютеры хранят кодовый поток именем способ.

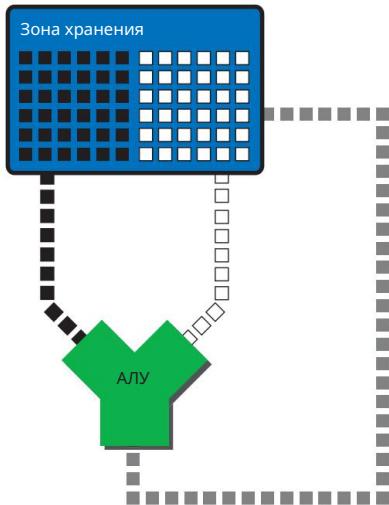


Рисунок 1-3: Простой компьютер с АЛУ и областью для хранения инструкций и данных

ПРИМЕЧАНИЕ Более продвинутые читатели могут заметить, что на рис. 1-3 (и на рис. 1-4 позже) я разделил код и данные в основной памяти на манер кэша первого уровня гарвардской архитектуры. На самом деле блоки кода и данных смешиваются в основной памяти, но сейчас я решил проиллюстрировать их как логически разделенные.

Способность современного компьютера хранить и повторно использовать предварительно записанные последовательности команд принципиально отличает его от предшествующих ему более простых вычислительных машин. До изобретения первой хранимой в памяти компьютеров¹ программы все вычислительные устройства, от счетов до первых электронных вычислительных машин должны были управляться оператором или группой операторов, которые вручную вводили определенную последовательность команд каждый раз, когда они хотели произвести определенный расчет. Напротив, современные компьютеры хранят и повторно используют такие последовательности команд, и поэтому они обладают уровнем гибкости и полезности, который отличает их от всего, что было раньше. В оставшейся части этой главы вы из первых рук увидите, как концепция хранимой программы влияет на конструкцию и возможности современного компьютера.

Уточнение модели File-Clerk

Давайте подробнее рассмотрим взаимосвязь между кодом, данными и потоками результатов с помощью быстрого примера. В этом примере кодовый поток состоит из одной инструкции add, которая сообщает АЛУ сложить два числа вместе.

¹ В 1944 г. Дж. Преспер Эккерт, Джон Мочли и Джон фон Нейман предложили первый компьютер с хранимой программой, EDVAC (электронный автоматический компьютер с дискретными переменными), а в 1949 г. такая машина, EDSAC, была построена Морисом Уилксом из Кембриджского университета.

Команда добавления перемещается из хранилища кода в АЛУ. Пока не будем интересоваться, как инструкция попадает из хранилища кода в АЛУ; давайте просто предположим, что он появляется в порту ввода кода ALU, объявляя, что есть добавление, которое должно быть выполнено немедленно. АЛУ выполняет следующую последовательность шагов:

1. Получите два добавляемых числа (входные операнды) из данных хранилище.
2. Добавьте числа.
3. Поместите результаты обратно в хранилище данных.

Предыдущий пример, вероятно, звучит просто, но он передает основные способ, которым компьютеры — все компьютеры — работают. Компьютерам подается последовательность инструкций одна за другой, и для их выполнения компьютер должен сначала получить необходимые данные, затем выполнить расчет, указанный в инструкции, и, наконец, записать результат в место, где конечный пользователь может найти Это. Эти три шага выполняются миллиарды раз в секунду на современном процессоре снова, и снова, и снова. Только потому, что компьютер выполняет эти шаги так быстро, он может создать иллюзию того, что происходит что-то гораздо более концептуально сложное.

Возвращаясь к нашей аналогии с файл-клерком, компьютер похож на файл-клерка, который весь день сидит за своим столом в ожидании сообщений от своего босса. В конце концов, босс отправляет ему сообщение, в котором просит выполнить вычисление пары чисел. В сообщении ему сообщается, какой расчет выполнить, и где в его личной картотеке находятся нужные цифры. Таким образом, клерк сначала извлекает числа из своей картотеки, затем выполняет расчет и, наконец, помещает результаты обратно в картотеку. Это скучная, бессмысленная, повторяющаяся задача, которая повторяется бесконечно, изо дня в день, и именно поэтому мы изобрели машину, которая может выполнять ее эффективно и не жаловаться.

Регистрационный файл

Поскольку числа должны быть сначала извлечены из хранилища, прежде чем их можно будет добавить, мы хотим, чтобы наше пространство для хранения данных было как можно более быстрым, чтобы операция могла выполняться быстро. Поскольку АЛУ является частью процессора, которая фактически выполняет сложение, мы хотели бы разместить хранилище данных как можно ближе к АЛУ, чтобы оно могло считывать операнды почти мгновенно. Однако практические соображения, такие как ограниченная площадь ЦП, ограничивают размер области хранения, которую мы можем разместить рядом с АЛУ. Это означает, что в реальной жизни большинство компьютеров имеют относительно небольшое количество очень быстрых мест хранения данных, подключенных к АЛУ. Эти места хранения называются регистрами, и на первых компьютерах с архитектурой x86 их было всего восемь. Эти регистры, расположенные в структуре хранения, называемой регистровым файлом, хранят только небольшое подмножество данных, необходимых потоку кода (и мы вскоре поговорим о том, где живут остальные эти данные).

Опираясь на наше предыдущее трехэтапное описание того, что происходит, когда АЛУ компьютера получает команду добавить два числа, мы можем изменить его следующим образом. Чтобы выполнить команду добавления, АЛУ должен выполнить следующие шаги:

1. Получите два добавляемых числа (входные операнды) из двух исходных регистров.
2. Добавьте числа.
3. Поместите результаты обратно в регистр назначения.

В качестве конкретного примера давайте рассмотрим сложение на простом компьютере только с четырьмя регистрами с именами A, B, C и D. Предположим, что каждый из этих регистров содержит число, и мы хотим сложить содержимое двух регистров вместе и перезаписать содержимое третьего регистра с результирующей суммой, как в следующей операции:

Код	Комментарии
A + B = C	Сложите содержимое регистров A и B и поместите результат в C, перезаписав что бы там ни было.

Получив команду на выполнение этой операции сложения, АЛУ в нашем простом компьютере выполнит следующие три знакомых шага:

1. Считайте содержимое регистров A и B.
2. Добавьте содержимое A и B.
3. Запишите результат в регистр C.

ПРИМЕЧАНИЕ. Вы должны понимать, что эти три шага представляют собой более конкретную форму последовательности чтения-изменения-записи из предыдущей, где общий шаг изменения заменяется операцией сложения.

Эта последовательность из трех шагов довольно проста, но она лежит в основе того, как на самом деле работает микропроцессор. На самом деле, если вы заглянете в обсуждение конвейера PowerPC 970 в главе 10, вы увидите, что он фактически имеет отдельные этапы для каждой из этих трех операций: этап 12 — это этап чтения регистра, этап 13 — фактический этап выполнения, и этап 14 является этапом обратной записи. (Не волнуйтесь, если вы не знаете, что такое конвейер, потому что это тема для главы 3.) Таким образом, ALU 970 считывает два операнда из файла регистров, складывает их вместе и записывает сумму обратно в файл регистров. Если бы мы остановили наше обсуждение прямо здесь, вы бы уже поняли три основных этапа основного целочисленного конвейера 970 — все остальные этапы являются либо просто подготовкой к этому моменту, либо они являются чистовой работой после него.

Оперативная память: когда одних только регистров недостаточно

Очевидно, что четыре (или даже восемь) регистров даже близко не соответствуют теоретически бесконечному объему памяти, о котором я упоминал ранее в этой главе. Чтобы сделать жизнеспособный компьютер, выполняющий полезную работу, вам нужно иметь возможность хранить очень больши-

наборы данных. Именно здесь вступает в действие основная память компьютера . Основная память, которая в современных компьютерах всегда представляет собой оперативную память (ОЗУ), хранит набор данных, с которым работает компьютер, и только небольшую часть этого набора данных в определенный момент времени. время перемещается в регистры для легкого доступа из АЛУ (как показано на рис. 1-4).

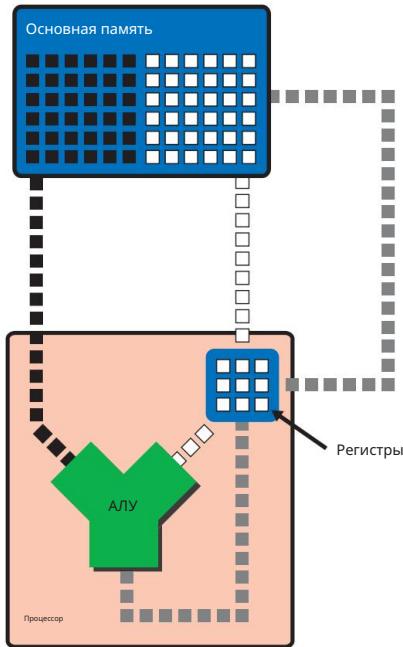


Рисунок 1-4: Компьютер с регистровым файлом

Рисунок 1-4 дает лишь малейшее представление об этом, но основная память расположена немного дальше от АЛУ, чем регистры. По сути, АЛУ и регистры — это внутренние части микропроцессора, а вот оперативная память — это совершенно отдельный компонент компьютерной системы, который подключается к процессору через шину памяти. Передача данных между оперативной памятью и регистрами по шине памяти занимает значительное время. Таким образом, если бы не было регистров и АЛУ приходилось считывать данные непосредственно из оперативной памяти для каждого вычисления, компьютеры работали бы очень медленно. Однако, поскольку регистры позволяют компьютеру хранить данные рядом с АЛУ, где к ним можно получить доступ почти мгновенно, скорость вычислений компьютера несколько отделена от скорости основной памяти.

(Мы более подробно обсудим проблему скорости доступа к памяти и вычислительной производительности в главе 11, когда будем говорить о кэшах.)

Модель File-Clerk пересмотрена и расширена

Возвращаясь к нашей метафоре файл-клерка, мы можем думать об оперативной памяти как о помещении для хранения документов, расположенном на другом этаже, а регистры — как о маленьком личном картотечном шкафу, куда файл-клерк кладет бумаги, над которыми он в данный момент работает. Клерк на самом деле ничего не знает

о помещении для хранения документов — что это такое и где оно находится, — потому что его письменный стол и его личный картотечный шкаф — это все, чем он занимается. Для документов, которые находятся в хранилище, есть еще один офисный работник, офисный секретарь, чья работа заключается в том, чтобы находить файлы в хранилище и забирать их для клерка.

Этот секретарь представляет собой несколько различных подразделений внутри процессора, с каждым из которых мы познакомимся в главе 4. А пока достаточно сказать, что, когда начальник хочет, чтобы клерк работал с файлом, которого нет в личном картотеке клерка, Секретарю сначала нужно приказать через сообщение от босса забрать файл из хранилища и поместить его в кабинет клерка, чтобы клерк мог получить к нему доступ, когда он получит приказ начать работу над ним.

Пример: сложение двух чисел

Чтобы перевести этот офисный пример в компьютерные термины, давайте посмотрим, как компьютер использует оперативную память, регистровый файл и АЛУ для сложения двух чисел.

Чтобы добавить два числа, хранящиеся в основной памяти, компьютер должен выполнить следующие действия:

1. Загрузите два операнда из основной памяти в два исходных регистра.
2. Добавьте содержимое исходных регистров и поместите результаты в регистр назначения, используя АЛУ. Для этого АЛУ должен выполнить следующие шаги: а. Считайте содержимое регистров А и В во входные порты ALU.
6. Добавьте содержимое А и В в ALU.
- в. Запишите результат в регистр С через выходной порт АЛУ.
3. Сохраните содержимое регистра назначения в основной памяти.

Поскольку все шаги 2a, 2b и 2c занимают незначительное время по сравнению с шагами 1 и 3, мы можем их игнорировать. Следовательно, наше дополнение выглядит так:

1. Загрузите два операнда из основной памяти в два исходных регистра.
2. Добавьте содержимое исходных регистров и поместите результаты в целевой регистр, используя АЛУ.
3. Сохраните содержимое регистра назначения в основной памяти.

Существование основной памяти означает, что пользователь — босс в нашей аналогии с регистратором — должен управлять потоком информации между основной памятью и регистрами ЦП. Это означает, что пользователь должен давать инструкции не только АЛУ процессора; он или она также должны выдавать инструкции частям ЦП, которые обрабатывают трафик памяти.

Таким образом, предыдущие три шага представляют виды инструкций, которые вы обнаружите, внимательно изучив поток кода.

Пристальный взгляд на поток кода: программа

В начале этой главы я определил поток кода как состоящий из «упорядоченной последовательности операций», и это определение вполне приемлемо. Но чтобы копнуть глубже, нам нужна более подробная картина того, что такое поток кода и как он работает.

Термин «операции» предполагает ряд простых арифметических операций, таких как сложение или вычитание, но кодовый поток состоит не только из арифметических операций. Поэтому правильнее было бы сказать, что кодовый поток состоит из упорядоченной последовательности инструкций. Инструкции, вообще говоря, — это команды, которые сообщают всему компьютеру — не только АЛУ, но и некоторым частям машины — какие именно действия следует выполнять. Как мы видели, список возможных действий компьютера включает в себя больше, чем просто арифметические операции.

Общие типы инструкций

Инструкции сгруппированы в упорядоченные списки, которые, если рассматривать их как единое целое, сообщают различным частям компьютера, как работать вместе для выполнения определенной задачи, такой как градация изображения в оттенках серого или воспроизведение медиафайла. Эти упорядоченные списки инструкций называются программами и состоят из нескольких основных типов инструкций.

В современных микропроцессорах RISC процесс перемещения данных между памятью и регистрами находится под явным контролем потока кода или программы. Таким образом, если программист хочет сложить два числа, которые находятся в основной памяти, а затем сохранить результат обратно в основную память, он или она должен написать список инструкций (программу), чтобы сказать компьютеру, что именно делать. Программа должна состоять из: инструкции загрузки [для](#) перемещения двух чисел из памяти в

регистры

[з](#) инструкция добавления , чтобы сказать АЛУ добавить два числа

[з](#) инструкция сохранения , чтобы сказать компьютеру поместить результат сложения обратно в память, перезаписав все, что было там ранее

Эти операции делятся на две основные категории:

Арифметические инструкции

Эти инструкции предписывают АЛУ выполнить арифметическое вычисление (например, сложение, подпрограмма, муль, деление).

Инструкции по доступу к памяти

Эти инструкции сообщают частям процессора, которые имеют дело с основной памятью, перемещать данные из основной памяти и в нее (например, загружать и хранить).

ПРИМЕЧАНИЕ Вскоре мы обсудим третий тип инструкций, инструкцию ветвления. Ответвляться

Инструкции технически представляют собой особый тип инструкций доступа к памяти, но они обращаются к хранилищу кода, а не к хранилищу данных. Тем не менее, проще относиться к ветвям как к третьей категории инструкций.

Арифметическая инструкция соответствует нашей метафоре калькулятора и является типом инструкции, наиболее знакомой всем, кто работал с компьютерами. К этой общей категории относятся такие инструкции, как сложение, вычитание, умножение и деление целых чисел и чисел с плавающей запятой.

ПРИМЕЧАНИЕ. Чтобы упростить обсуждение и сократить количество терминов, я временно включаю логические операции, такие как И, ИЛИ, НЕ, ИЛИ и т. д., под общим заголовком арифметических инструкций. Разница между арифметическими и логическими операциями будет представлена в главе 2.

Инструкция доступа к памяти так же важна, как и арифметическая инструкция, потому что без доступа к областям хранения данных основной памяти компьютер не смог бы получить данные в регистровый файл или из него.

Чтобы показать вам, как доступ к памяти и арифметические операции работают вместе в контексте потока кода, в оставшейся части этой главы будет использоваться ряд все более подробных примеров. Все примеры основаны на простом гипотетическом компьютере, который я назову DLW-1.2.

Базовая архитектура DLW-1 и формат арифметических инструкций

Микропроцессор DLW-1 состоит из АЛУ (вместе с несколькими другими блоками, которые я опишу позже), подключенного к четырем регистрам, которые для удобства названы A, B, C и D. DLW-1 подключается к банку основной памяти, расположенному в виде строки из 256 ячеек памяти, пронумерованных от #0 до #255. (Число, которое идентифицирует отдельную ячейку памяти, называется адресом.)

Формат арифметических инструкций DLW-1

Все арифметические инструкции DLW-1 имеют следующий формат инструкций:

инструкция источник1, источник2, пункт назначения

Этот формат инструкций состоит из четырех частей, каждая из которых называется полем. Поле инструкции определяет тип выполняемой операции (например, сложение, вычитание, умножение и т. д.). Два исходных поля сообщают компьютеру, в каких регистрах хранятся два числа, над которыми выполняются операции, или операнды. Наконец, поле назначения сообщает компьютеру, в какой регистр поместить результат.

В качестве быстрой иллюстрации, инструкция сложения, которая складывает числа в регистрах A и B (два исходных регистра) и помещает результат в регистр C. (регистр назначения) будет выглядеть так:

Код	Комментарии
добавить A, B, C	Сложите содержимое регистров A и B и поместите результат в C, перезаписав все, что там было ранее.

² «DLW» в честь архитектуры DLX, использованной Хеннесси и Паттерсоном в их книгах по компьютерной архитектуре.

Формат инструкций памяти DLW-1

Чтобы заставить процессор переместить два операнда из основной памяти в исходные регистры, чтобы их можно было добавить, вам нужно явно указать процессору, что вы хотите переместить данные из двух определенных ячеек памяти в два определенных регистра. Эта операция «заполнения» выполняется с помощью инструкции доступа к памяти, называемой загрузкой.

Как следует из названия, инструкция загрузки загружает соответствующие данные из основной памяти в соответствующие регистры, чтобы эти данные были доступны для последующих арифметических инструкций. Инструкция сохранения обратна инструкции загрузки, она берет данные из регистра и сохраняет их в ячейке основной памяти, перезаписывая все, что было там ранее.

Все инструкции доступа к памяти для DLW-1 имеют следующий формат инструкций:

источник инструкции, назначение

Для всех обращений к памяти в поле инструкции указывается тип выполняемой операции с памятью (либо загрузка, либо сохранение). В случае загрузки поле источника сообщает компьютеру, из какого адреса памяти следует извлечь данные, а поле назначения указывает, в какой регистр их поместить. И наоборот, в случае хранилища поле источника сообщает компьютеру, какой регистр для получения данных, а поле назначения указывает, в какой адрес памяти записывать данные.

Пример программы DLW-1

Теперь рассмотрим программу 1-1, которая является частью кода DLW-1. Каждая из строк в программе должна выполняться последовательно для достижения желаемого результата.

Код линии	Комментарии
1	нагрузка №12, А Считайте содержимое ячейки памяти №12 в регистр А.
2	груз №13, Б Считайте содержимое ячейки памяти №13 в регистр В.
3	добавить А, В, С Сложите числа в регистрах А и В и сохраните результат в С.
4 магазин С, №14	Запишите результат сложения из регистра С в ячейку памяти №14.

Программа 1-1: Программа для добавления двух чисел из основной памяти

Предположим, что перед запуском основная память выглядела следующим образом.
Программа 1-1:

№11 №12 №13 №14

12	6	2	3
----	---	---	---

После выполнения нашего сложения и сохранения результатов память будет изменена таким образом, что содержимое ячейки № 14 будет перезаписано суммой ячеек № 12 и № 13, как показано здесь:

№11 №12 №13 №14

12	6	2	8
----	---	---	---

Пристальный взгляд на доступ к памяти: регистр против немедленного

До сих пор примеры предполагали, что программист знает точное место в памяти каждого числа, которое он или она хочет загрузить и сохранить. Другими словами, предполагается, что при составлении каждой программы программист имеет в своем распоряжении список содержимого ячеек памяти с №0 по №255 .

Хотя такой точный снимок начального состояния основной памяти может быть осуществимым для небольшого примерного компьютера с 256 ячейками памяти, в реальном мире таких моментальных снимков почти никогда не существует. Реальные компьютеры имеют миллиарды возможных местоположений, в которых могут храниться данные, поэтому программистам нужен более гибкий способ доступа к памяти, способ, который не требует, чтобы при каждом доступе к памяти численно указывался точный адрес памяти.

Современные компьютеры позволяют использовать содержимое регистра в качестве памяти. адрес, ход, который обеспечивает программисту желаемую гибкость. Но прежде чем обсуждать эффекты этого хода более подробно, давайте еще раз взглянем на базовую инструкцию добавления .

Непосредственные значения

До сих пор все арифметические инструкции требовали в качестве входных данных два исходных регистра. Однако можно заменить один или оба исходных регистра явным числовым значением, называемым непосредственным значением. Например, чтобы увеличить какое-либо число в регистре A на 2, нам не нужно загружать значение 2 во второй исходный регистр, такой как B, из какой-либо ячейки основной памяти, содержащей это значение. Скорее, мы можем просто сказать компьютеру добавить 2 к A.

непосредственно следующим образом:

Код	Комментарии
добавить A, 2, A	Добавьте 2 к содержимому регистра A и поместите результат обратно в A, перезаписав все, что там было.

На самом деле я всегда использовал немедленные значения в своих примерах, но только не в каких-либо арифметических инструкциях. Во всех предыдущих примерах при каждой загрузке и сохранении используется непосредственное значение для указания адреса памяти.

Таким образом, #12 в инструкции загрузки в строке 1 программы 1-1 — это просто непосредственное значение (обычное целое число), перед которым стоит знак #, чтобы дать компьютеру понять, что это конкретное непосредственное значение является адресом памяти, обозначающим адрес памяти. ячейка памяти.

Адреса памяти — это обычные целые числа, специально помеченные знаком #.

Поскольку это обычные целые числа, их можно хранить в регистрах — и хранить в памяти — точно так же, как и любые другие числа.

Таким образом, целочисленное содержимое регистра, такого как D, может быть истолковано компьютером как представление адреса памяти.

Например, предположим, что мы сохранили число 12 в регистре D и намерены использовать содержимое регистра D в качестве адреса ячейки памяти в программе 1-2.

Код линии	Комментарии
1	нагрузка #D, A Считайте содержимое ячейки памяти, обозначенной номером, хранящимся в D (где D = 12), в регистр A.
2	груз №13, B Считайте содержимое ячейки памяти №13 в регистр B.
3	добавить A, B, C Сложите числа в регистрах A и B и сохраните результат в C.
4	магазин C, №14 Запишите результат сложения из регистра C в ячейку памяти №14.

Программа 1-2: Программа для сложения двух чисел из основной памяти с использованием адреса, хранящегося в регистре.

Программа 1-2 по сути такая же, как Программа 1-1, и с учетом того же ввода, он дает те же результаты. Разница только в строке 1:

Программа 1-1, строка 1	Программа 1-2, строка 1
нагрузка №12, A	нагрузка #D, A

Поскольку содержимое D — это число 12, мы можем сказать компьютеру, чтобы найдите в D адрес ячейки памяти, заменив имя регистра (на этот раз отмеченное знаком # для использования в качестве адреса) на фактический номер ячейки памяти в инструкции загрузки строки 1. Таким образом, первые строки программ 1-1 и 1-2 функционально эквивалентны.

Этот же трюк работает и для инструкций магазина . Например, если мы поместите число 14 в D , мы можем изменить команду store в строке 4 программы 1-1, чтобы она читалась следующим образом: store C, #D. Опять же, эта модификация не изменит вывод программы.

Поскольку адреса памяти — это обычные числа, их можно хранить как в ячейках памяти, так и в регистрах. Программа 1-3 иллюстрирует использование адреса памяти, хранящегося в другой ячейке памяти. Если мы возьмем входные данные для программы 1-1 и применим их к программе 1-3, мы получим тот же вывод, как если бы мы просто запустили программу 1-1 без изменений:

Код линии	Комментарии
1	load #11, D Прочитать содержимое ячейки памяти #11 в D.
2	нагрузка №D, A Прочитать содержимое ячейки памяти, обозначенной цифрой в D (где D = 12) в регистр A.
3	груз №13, B Считайте содержимое ячейки памяти №13 в регистр B.
4 добавить A, B, C Сложите числа в регистрах A и B и сохраните результат в C.	
5	магазин C, №14 Запишите результат сложения из регистра C в ячейку памяти №14.

Программа 1-3: Программа для сложения двух чисел из памяти, используя адрес, хранящийся в ячейке памяти.

Первая инструкция в программе 1-3 загружает из памяти число 12. ячейку № 11 в регистр D. Затем вторая инструкция использует содержимое D (значение 12) в качестве адреса памяти для загрузки регистра A в ячейку памяти #12.

Но зачем утруждать себя хранением адресов памяти в ячейках памяти, а затем загрузкой адресов из основной памяти в регистры до того, как они, наконец, будут снова готовы к использованию для доступа к памяти? Не слишком ли это сложный способ?

На самом деле, эти возможности предназначены для облегчения жизни программистов, потому что при использовании с описанной ниже методикой адресации относительно регистров они значительно упрощают управление кодом и трафиком данных между процессором и большими объемами оперативной памяти.

Регистровая адресация

В реальных программах при загрузке и сохранении чаще всего используется адресация относительно регистра, которая представляет собой способ указания адресов памяти относительно регистра, содержащего фиксированный базовый адрес.

Например, мы использовали D для хранения адресов памяти, поэтому предположим, что на DLW-1 мы можем предположить, что, если явно не указано иное, операционная система всегда загружает начальный адрес (или базовый адрес).) сегмента данных программы в D. Помните, что код и данные логически разделены в основной памяти, и что данные поступают в процессор из области хранения данных, а код поступает в процессор из специальной области хранения кода. Сама основная память представляет собой всего лишь один длинный ряд недифференцированных ячеек памяти, каждая шириной в один байт , в которых хранятся числа. Компьютер делит этот длинный ряд байтов на несколько сегментов, в некоторых из которых хранится код, а в других — данные.

Сегмент данных — это блок смежных ячеек памяти, в котором программа хранит все свои данные, поэтому, если программист знает начальный адрес сегмента данных (базовый адрес) в памяти, он или она может получить доступ ко всем другим ячейкам памяти в этом сегменте. отрезок по этой формуле:

базовый адрес + смещение

где смещение — это расстояние в байтах от желаемой ячейки памяти до базового адреса сегмента данных.

Таким образом, инструкции по загрузке и сохранению в ассемблере DLW-1 обычно выглядеть примерно так:

Код	Комментарии
нагрузка #(D + 108), A	Считайте содержимое ячейки памяти в ячейке #(D + 108) в A.
магазин B, #(D + 108)	Запишите содержимое B в ячейку памяти в ячейке #(D + 108).

В случае нагрузки процессор берет число в D, т.е. базовый адрес сегмента данных, добавляет к нему 108 и использует результат в качестве адреса назначения загрузки . Магазин работает точно так же.

Конечно, этот метод требует, чтобы операция быстрого сложения (называемая вычислением адреса) была частью выполнения инструкции загрузки , поэтому блоки загрузки-сохранения в современных процессорах содержат очень быстрое оборудование для сложения целых чисел. (Как мы узнаем из главы 4, модуль загрузки-сохранения — это исполнительный модуль, отвечающий за выполнение инструкций загрузки и сохранения , точно так же, как арифметико-логический модуль отвечает за выполнение арифметических инструкций.)

Используя относительную адресацию регистров вместо абсолютной адресации (в которой адреса памяти задаются как непосредственные значения), программист может писать программы, не зная точного местоположения данных в памяти. Все, что нужно программисту, это знать, в какой регистр операционная система поместит базовый адрес сегмента данных, и он или она сможет выполнять все обращения к памяти относительно этого базового адреса. В ситуациях, когда программист использует абсолютную адресацию, когда операционная система загружает программу в память, все непосредственные значения адреса программы должны быть изменены, чтобы отразить фактическое расположение сегмента данных в памяти.

Поскольку и адреса памяти, и обычные целые числа хранятся в одних и тех же регистрах, эти регистры называются регистрами общего назначения (GPR).

На DLW-1 все A, B, C и D являются георадарами.

2

МЕХАНИКА ПРОГРАММЫ ИСПОЛНЕНИЕ

Теперь, когда мы познакомились с основами компьютерной организации, пришло время поближе рассмотреть основные моменты того, как хранимые программы на самом деле выполняются компьютером. С этой целью в этой главе будут рассмотрены основные концепции программирования, такие как машинный язык, модель программирования, архитектура набора команд, инструкции ветвления и цикл выборка-выполнение.

Коды операций и машинный язык

Если вы до сих пор следили за обсуждением, вас не должно удивить, что и адреса памяти, и инструкции являются обычными числами, которые можно хранить в памяти. Все инструкции в такой программе, как Программа 1-1, представлены внутри компьютера в виде строк чисел.

Действительно, программа представляет собой одну длинную строку чисел, хранящуюся в ряде ячеек памяти.

Как программа, подобная программе 1-1, представляется в числовом записи, чтобы ее можно было сохранить в памяти и выполнить на компьютере? Ответ проще, чем вы думаете.

Как вы, возможно, уже знаете, компьютер на самом деле понимает только 1 и 0 (или «высокое» и «низкое» электрическое напряжение), а не английские слова, такие как сложение, загрузка и сохранение, или буквы и числа с основанием 10, такие как A, B, 12 и 13. Таким образом, чтобы компьютер мог выполнять программу, все ее инструкции должны быть представлены в двоичной записи. Подумайте о переводе английских слов в точки и тире азбуки Морзе, и у вас будет некоторое представление о том, о чем я говорю.

Машинный язык на DLW-1

Перевод программ любой сложности на этот машинный язык , основанный на двоичном коде, — это масштабная задача, которую должен выполнять компьютер, но я покажу вам основы того, как это работает, чтобы вы могли понять, что происходит. Следующий пример упрощен, но, тем не менее, полезен.

Английские слова в программе, такие как « добавить », « загрузить » и « сохранить », — это мнемоники . (что означает, что их легко запомнить людям), и все они отображаются в строки двоичных чисел, называемые кодами операций, которые компьютер может понять.

Каждый код операции обозначает другую операцию, которую может выполнять процессор.

В таблице 2-1 все мнемоники, использованные в главе 1, сопоставлены с 3-битным кодом операции для гипотетического микропроцессора DLW-1. Мы также можем сопоставить имена четырех регистров с 2-битными двоичными кодами, как показано в таблице 2-2.

Таблица 2-1: Сопоставление мнемоники с

Коды операций для DLW-1

Мнемоника	Опкод
добавить	000
суб	001
нагрузка	010
хранить	011

Таблица 2-2: Сопоставление регистров с

Двоичные коды для DLW-1

регистр	Бинарный код
A	00
Б	01
С	10
Д	11

Двоичные значения, представляющие как коды операций, так и коды регистров, располагаются в одном из нескольких 16-битных (или 2-байтовых) форматов, чтобы получить полную инструкцию машинного языка, которая представляет собой двоичное число, которое можно хранить в ОЗУ и использовать. процессором.

ПРИМЕЧАНИЕ. Поскольку инструкции, написанные программистом, должны быть переведены в двоичные коды, прежде чем компьютер сможет их прочитать, обычно можно увидеть программы в любом формате — двоичном, на ассемблере или на языке высокого уровня, таком как BASIC или C, в общем называемом «кодом», или «коды». Поэтому программисты иногда говорят о «коде ассемблера», «двоичном коде» или «коде C», имея в виду программы, написанные на ассемблере, двоичном коде или языке C. Программисты также часто описывают процесс программирования как «написание кода» или «кодирование». Я принял эту терминологию в этой книге и впредь буду регулярно использовать термин «код» для общего обозначения последовательностей инструкций и программ.

Двоичное кодирование арифметических инструкций

Арифметические инструкции имеют простейшие форматы инструкций машинного языка, поэтому мы начнем с них. На рис. 2-1 показан формат кодирования на машинном языке арифметической инструкции регистрационного типа .

01234567								
Режим	код операции			источник1			источник2	

Байт 1

8	9	10	11	12	13	14	15
назначения	000000						

Байт 2

Рисунок 2-1: Формат машинного языка для инструкции регистрационного типа

В арифметической инструкции регистрационного типа (то есть арифметической инструкции, которая использует только регистры и не использует непосредственные значения) первый бит инструкции является битом режима. Если бит режима установлен в 0, то инструкция является инструкцией регистрационного типа; если он установлен в 1, то инструкция имеет непосредственный тип.

Биты 1–3 инструкции определяют код операции, который сообщает компьютеру, какой тип операции представляет инструкция. Биты 4–5 определяют первый исходный регистр инструкции, 6–7 — второй исходный регистр, а 8–9 — целевой регистр. Последние шесть битов не нужны регистру для регистрации арифметических инструкций, поэтому они дополняются нулями (они обнуляются).

на компьютерном жаргоне) и игнорируется.

Теперь давайте воспользуемся двоичными значениями в таблицах 2-1 и 2-2, чтобы перевести сложение инструкции в строке 3 программы 1-1 в 2-байтовую (или 16-битную) инструкцию машинного языка:

Инструкция на языке ассемблера	Инструкция на машинном языке
добавить A, B, C	00000001 10000000

Вот еще несколько примеров арифметических инструкций, чтобы вы могли освоиться:

Инструкция на языке ассемблера	Инструкция на машинном языке
добавить С, Д, А	00001011 00000000
добавить Д, В, С	00001101 10000000
суб А, Д, С	00010011 10000000

Увеличение количества двоичных цифр в полях кода операции и регистра увеличивает общее количество инструкций, которые машина может использовать, и количество регистров, которые она может иметь. Например, если вы что-то знаете о двоичной записи, то вы, вероятно, знаете, что 3-битный код операции позволяет процессору отображать до 23 мнемоник, что означает, что он может иметь до 23 , или же 8, инструкций в своем наборе инструкций; увеличение размера кода операции до 8 бит позволило бы набору инструкций процессора содержать до 256, инструкций. Точно так же увеличение количества битов в поле регистра увеличивает возможное количество регистров, которые может иметь машина.

Арифметические инструкции, содержащие непосредственное значение, используют формат инструкции непосредственного типа , который немного отличается от формата регистрарного типа, который мы только что видели. В инструкции немедленного типа первый байт содержит код операции, исходный регистр и регистр назначения, а второй байт содержит непосредственное значение, как показано на рис. 2-2.

01234567							
Режим	код операции			источник		назначения	

Байт 1

8	9	10	11	12	13	14	15
8-битное непосредственное значение							

Байт 2

Рисунок 2-2: Формат машинного языка для инструкции немедленного типа

Вот несколько арифметических инструкций непосредственного типа, переведенных с языка ассемблера на машинный язык:

Инструкция на языке ассемблера	Инструкция на машинном языке
добавить С, 8, А	10001000 00001000
добавить 5, А, С	10000010 00000101
саб 25, Д, С	10011110 00011001

Двоичное кодирование инструкций доступа к памяти

Инструкции доступа к памяти используют форматы инструкций как регистрового, так и непосредственного типа точно так же, как показано для арифметических инструкций. Единственная разница заключается в том, как они их используют. Сначала возьмем случай нагрузки.

Инструкция по загрузке

Ранее мы видели два типа нагрузки, первый из которых был немедленным. Загрузка немедленного типа (см. рис. 2-3) использует формат инструкции немедленного типа, но поскольку источником загрузки является непосредственное значение (адрес памяти), а не регистр, поле источника не нужно и должно быть обнулено. (Однако исходное поле не игнорируется, и через мгновение мы увидим, что произойдет, если оно не будет обнулено.)

01234567								
Режим	код операции			00	назначения			

Байт 1

8	9	10	11	12	13	14	15
8-битный непосредственный адрес источника							

Байт 2

Рисунок 2-3: Формат машинного языка для загрузки немедленного типа

Теперь давайте переведем загрузку немедленного типа в строку 1 программы 1-1 (12 — это 1100 в двоичной записи):

Инструкция на языке ассемблера	Инструкция на машинном языке
нагрузка №12, А	10100000 00001100

2-байтовая инструкция машинного языка справа представляет собой двоичное представление. Отправка инструкции на ассемблере слева. Первый байт соответствует инструкции загрузки немедленного типа , которая принимает регистр A в качестве адресата. Второй байт представляет собой двоичное представление числа 12, которое является исходным адресом в памяти, из которого должны быть загружены данные.

Второй тип нагрузки, который мы видели, — это регистровый тип. Тип регистра load использует формат инструкций регистра типа, но поле source2 обнуляется и игнорируется, как показано на рис. 2-4.

На рис. 2-4 поле source1 указывает регистр, содержащий адрес памяти, из которого процессор должен загрузить данные, а поле назначения указывает регистр, в который должны быть помещены загруженные данные.

01234567							
Режим	код операции			источник1		00	

Байт 1

8	9	10	11	12	13	14	15
назначения		000000					

Байт 2

Рисунок 2-4: Формат машинного языка для загрузки регистрового типа

Для загрузки с адресацией относительно регистра мы используем версию формата инструкций прямого типа, показанную на рис. 2-5, с базовым полем, указывающим регистр, который содержит базовый адрес и смещение, хранящееся во втором байте инструкции.

01234567							
Режим	код операции			база		назначения	

Байт 1

8	9	10	11	12	13	14	15
8-битное немедленное смещение							

Байт 2

Рисунок 2-5: Формат машинного языка для загрузки относительно регистра

Вспомним из Таблицы 2-2, что 00 — это двоичное число, обозначающее регистр A. Следовательно, в результате особой схемы кодирования машинного языка DLW-1 любой регистр, кроме A, теоретически может использоваться для хранения базового адреса регистра. относительная нагрузка.

Инструкция магазина

Двоичный формат регистрового типа для инструкции сохранения такой же, как и для загрузки, за исключением того, что поле назначения указывает регистр, содержащий адрес памяти назначения, а поле источник1 указывает регистр, содержащий данные, которые должны быть сохранены в памяти. Память.

Формат машинного языка непосредственного типа для сохранения, показанный на рис. 2-6, также аналогичен формату немедленного типа для загрузки, за исключением того, что регистр назначения не требуется (место назначения — это непосредственный адрес памяти), поле назначения обнуляется, а поле источника указывает, в каком регистре хранятся данные для сохранения.

01234567							
Режим	код операции			источник		00	

Байт 1

8	9	10	11	12	13	14	15
8-битный непосредственный адрес назначения							

Байт 2

Рисунок 2-6: Формат машинного языка для хранилища немедленного типа

С другой стороны, хранилище относительно регистра использует ту же самую непосредственную память. тип формата инструкции, используемый для загрузки относительно регистра (рис. 2-5), но в поле назначения установлено ненулевое значение, а смещение сохраняется во втором байте. Опять же, базовый адрес для хранилища относительно регистра теоретически может храниться в любом регистре, кроме А, хотя по соглашению он хранится в D.

Перевод примера программы на машинный язык

Для нашего простого компьютера с четырьмя регистрами, тремя инструкциями и 256 ячейками памяти утомительно, но тривиально перевести программу 1-1 в машиночитаемое двоичное представление, используя предыдущие таблицы и форматы инструкций. Программа 2-1 показывает перевод.

Язык линейного ассемблера	Машинный язык
1 нагрузка №12, А	10100000 00001100
2 груз №13, Б	10100001 00001101
3 добавить А, В, С	00000001 10000000
4 магазин С, №14	10111000 00001110

Программа 2-1: перевод программы 1-1 на машинный язык.

Единицы и нули в крайнем правом столбце программы 2-1 представляют высокое и низкое напряжения, которыми «думает» компьютер.

Инструкции реального машинного языка обычно длиннее и сложнее, чем простые, которые я дал здесь, но основная идея точно такая же. Программные инструкции переводятся на машинный язык механическим, предопределенным образом, и даже в случае полностью современного микропроцессора выполнение таких переводов вручную — это просто вопрос знания форматов инструкций и доступа к нужным схемам и таблицам.

Конечно, по большей части единственные люди, которые делают такие переводы вручную, — это студенты-выпускники компьютерной инженерии или компьютерных наук, которым задали домашнюю работу. Однако так было не всегда.

Модель программирования и ISA

В старые недобрые времена программистам приходилось вводить программы в компьютер непосредственно на машинном языке (после того, как они прошли пять миль по снегу в гору на работу). На самых ранних стадиях вычислений это делалось с помощью переключателей. Программист поместил строки из 1 и 0 в очень ограниченную память компьютера, запустил программу, а затем проанализировал полученные строки из 1 и 0, чтобы расшифровать ответ.

Как только объемы памяти и вычислительная мощность увеличились до такой степени, что время и усилия программиста были достаточно ценны по сравнению с вычислительным временем и объемом памяти, ученые-компьютерщики разработали способы, позволяющие компьютеру использовать часть своей мощности и памяти, чтобы взять на себя часть времени по созданию загадочного ввода и вывода немного более человечным. -дружелюбный.

Короче говоря, утомительная задача преобразования удобочитаемых программ в машиночитаемый двоичный код была автоматизирована; отсюда и рождение программирования на ассемблере . Теперь программы можно было писать с использованием мнемоники, имен регистров и ячеек памяти, прежде чем ассемблер преобразовывал их в машинный язык для обработки.

Чтобы писать программы на ассемблере для машины, вы должны понимать доступные ресурсы машины: сколько у нее регистров, какие инструкции она поддерживает и так далее. Другими словами, вам нужна четко определенная модель машины, которую вы пытаетесь запрограммировать.

Модель программирования

Программная модель представляет собой интерфейс программиста к микропроцессору. Он скрывает все сложные детали реализации процессора за относительно простым, чистым уровнем абстракции, открывающим программисту все функциональные возможности процессора. (Подробнее об истории и развитии модели программирования см. в главе 4.)

На рис. 2-7 показана схема модели программирования для восьмиregistровой машины. К настоящему времени большинство частей диаграммы должны быть вам знакомы. АЛУ выполняет арифметические операции, регистры хранят числа, а блок ввода-вывода (модуль ввода-вывода) отвечает за взаимодействие с памятью и остальной частью системы (через загрузку и сохранение). Части процессора, с которыми мы еще не встречались, лежат в блоке управления. Из них мы сейчас рассмотрим счетчик программ и регистр инструкций .

Регистр команд и счетчик команд

Поскольку программы хранятся в памяти в виде упорядоченных последовательностей инструкций, а память организована в виде линейной последовательности адресов, каждая инструкция в программе живет по своему собственному адресу памяти. Чтобы пройти и выполнить строки программы, компьютер просто начинает с начального адреса программы, а затем проходит через каждую последующую ячейку памяти, извлекая каждую последующую инструкцию из памяти, помещая ее в специальный регистр и выполняя ее, как показано. на рис. 2-8.

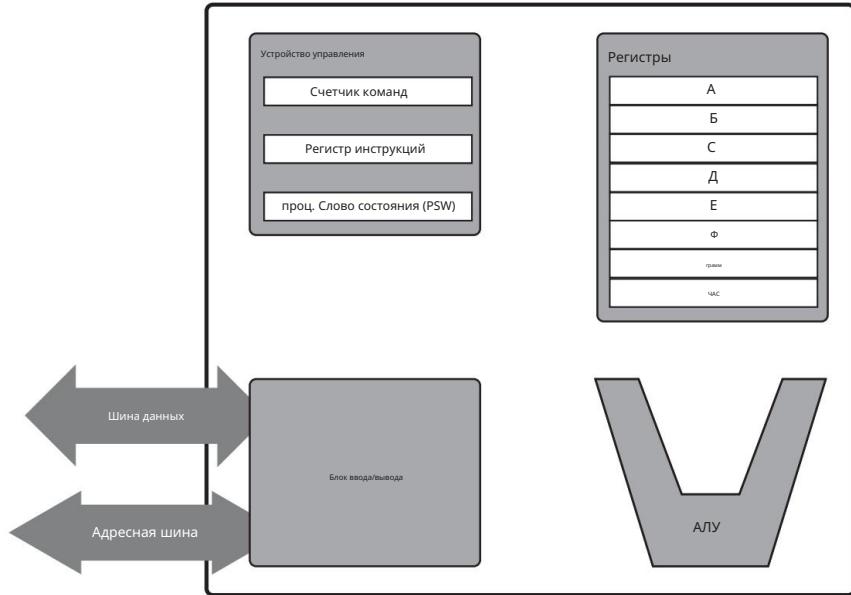


Рисунок 2-7: Модель программирования для простой восьмиregistровой машины

Инструкции в нашем компьютере DLW-1 имеют длину два байта. Если мы предположим, что каждая ячейка памяти содержит один байт, то DLW-1 должен шагать по памяти, выбирая инструкции из двух ячеек одновременно.

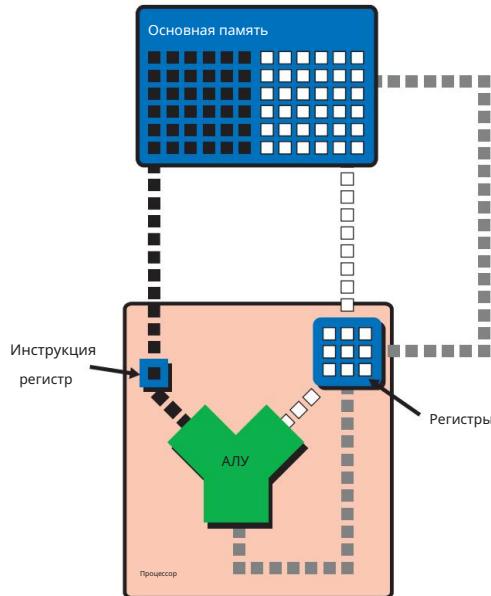


Рисунок 2-8: Простой компьютер с регистрами команд и данных

Например, если бы начальный адрес в программе 1-1 был #500, в памяти это выглядело бы как на рис. 2-9 (конечно, с инструкциями, отображаемыми на машинном языке, а не на ассемблере).

#500 #501 #502 #503 #504 #505 #506 #507

загрузить № 12,	А загрузить № 13,	В добавить А, В,	С сохранить С, № 14
-----------------	-------------------	------------------	---------------------

Рисунок 2-9: Иллюстрация программы 1-1 в памяти, начиная с адреса #500

Выборка инструкций: загрузка регистра инструкций

Выборка инструкций — это особый тип загрузки, который происходит автоматически для каждой инструкции. Он всегда принимает адрес, который в данный момент находится в регистре счетчика программ, как источник, а регистр команд — как место назначения. Блок управления использует выборку для загрузки каждой инструкции программы из памяти в регистр инструкций, где эта инструкция декодируется перед выполнением; и пока эта инструкция декодируется, процессор помещает адрес следующей инструкции в программный счетчик, увеличивая адрес, который в данный момент находится в программном счетчике, так что вновь увеличенный адрес указывает на следующую инструкцию в последовательности. В случае нашего DLW-1 программный счетчик увеличивается на два каждый раз при выборке инструкции, потому что двухбайтовые инструкции начинаются с каждого второго байта в памяти.

Запуск простой программы: цикл выборки-выполнения

В главе 1 мы обсуждали шаги, предпринимаемые процессором для выполнения вычислений с числами с использованием АЛУ в сочетании с полученной арифметической командой. Теперь давайте посмотрим, какие шаги предпринимает процессор, чтобы получить серию инструкций — программу — и передать их либо в АЛУ (в случае арифметических инструкций), либо в аппаратное обеспечение доступа к памяти (в случае загрузки и сохранения) :

1. Получить следующую инструкцию по адресу, хранящемуся в программном счетчике, и загрузить эту инструкцию в регистр инструкций. Увеличьте счетчик программы.
2. Декодировать инструкцию в регистре инструкций.
3. Выполните инструкцию в регистре инструкций, используя следующие правила:
 - a. Если инструкция является арифметической, выполните ее, используя АЛУ и регистровый файл.
 - b. Если инструкция является инструкцией доступа к памяти, выполните ее, используя аппаратное обеспечение доступа к памяти.

Эти три шага довольно просты, и с одной модификацией они описывают способ, которым микропроцессоры выполняют программы (как мы увидим в разделе «Инструкции перехода» на стр. 30). Ученые-компьютерщики часто

назовите эти шаги циклом выборки-выполнения или циклом выборки-выполнения. Цикл выполнения выборки повторяется до тех пор, пока компьютер включен. Машина повторяет весь цикл, от шага 1 до шага 3, снова и снова много миллионов или миллиардов раз в секунду, чтобы запускать программы.

Давайте пройдем три шага с нашей программой-примером, как показано на рис. 2-9. (В этом примере предполагается, что #500 уже находится в счетчике программ.) Вот что делает процессор по порядку:

1. Выбрать инструкцию, начинающуюся с #500, и загрузить в регистр инструкции загрузку #12, A.
Увеличьте счетчик программ до #502.
2. Декодировать загрузку #12, A в регистре инструкций.
3. Выполнить загрузку #12, A из регистра инструкций, используя память аппаратное обеспечение доступа.
4. Получить инструкцию, начинающуюся с #502, и загрузить загрузку #13, B в регистр инструкций. Увеличьте счетчик программ до #504.
5. Декодировать загрузку #13, B в регистре инструкций.
6. Выполнить загрузку #13, B из регистра инструкций, используя память аппаратное обеспечение доступа.
7. Вызовите команду, начинающуюся с #504, и загрузите сложение A, B, C в регистр инструкций.
Увеличьте счетчик программ до #506.
8. Декодируйте добавление A, B, C в регистр инструкций.
9. Выполните добавление A, B, C из регистра инструкций, используя ALU и регистрационный файл.
10. Выбрать инструкцию по адресу #506 и загрузить в память C, #14 регистр инструкций.
Увеличьте счетчик программ до #508.
11. Декодируйте сохранение C, #14 в регистре команд.
12. Выполнить сохранение C, #14 из регистра инструкций, используя память аппаратное обеспечение доступа.

ПРИМЕЧАНИЕ . Чтобы увеличить этапы выполнения предыдущей последовательности, вернитесь к главе 1 и особенно разделы «Уточнение модели File-Clerk» на стр. 6 и «ОЗУ: когда одни регистры не помогут» на стр. 8. программа на любой машине. Конечно, для большей части того, что я здесь представил, существуют важные машинно-специфические вариации, но общие черты (и даже приличное количество деталей) одинаковы.

Часы

Шаги с 1 по 12 в предыдущем разделе не требуют произвольного количества времени для выполнения. Скорее, они выполняются в соответствии с импульсом часов, который управляет каждым действием, предпринимаемым процессором.

Этот тактовый импульс, генерируемый модулем тактового генератора на материнской плате и подаваемый в процессор извне, синхронизирует работу процессора так, что, по крайней мере, на DLW-1 все три шага выборки выполнение цикла завершается ровно за один такт. Таким образом

Программе на рис. 2.9, как я проследил ее выполнение в предыдущем разделе, требуется ровно четыре такта, чтобы закончить выполнение, потому что новая инструкция вызывается с каждым тактом.

Одним из очевидных способов ускорить выполнение программ на DLW-1 было бы ускорение его тактового генератора, чтобы выполнение каждого шага занимало меньше времени. В целом это относится ко всем микропроцессорам, отсюда и гонка среди разработчиков микропроцессоров по созданию и продаже чипов с еще более высокими тактовыми частотами. (Подробнее о взаимосвязи между тактовой частотой и производительностью мы поговорим в главе 3.)

Инструкции ветки

Как я уже представил, процессор последовательно перемещается по каждой строке программы, пока не достигнет конца программы, после чего вывод программы становится доступным для пользователя.

Однако в потоке команд есть определенные инструкции, которые позволяют процессору перейти к строке программы, находящейся вне последовательности. Например, вставив инструкцию ветвления в строку 5 программы, мы можем заставить блок управления процессора перепрыгнуть вниз до строки 20 и начать выполнение там (прямая ветвь), или мы можем заставить его вернуться назад к строке 1 (обратная ветвь). Поскольку программа представляет собой упорядоченную последовательность инструкций, включая инструкции прямого и обратного ветвления, мы можем произвольно перемещаться по программе. Это мощная способность, а ветви — неотъемлемая часть вычислений.

Вместо того, чтобы думать о прямых или обратных ветвлениях, для наших текущих целей полезнее разделить все ветвления на один из следующих двух типов: условные ветвления или безусловные ветвления.

Безусловная ветвь

Инструкция безусловного перехода состоит из двух частей: инструкции перехода и целевого адреса.

прыжок #цель

Для безусловного перехода `#target` может быть либо непосредственным значением, либо пример #12, или адрес, хранящийся в регистре, например `#D`.

Безусловные переходы довольно легко выполнить, поскольку все, что компьютер должен сделать после декодирования такого перехода в регистре команд, — это заставить блок управления заменить адрес, находящийся в данный момент в программном счетчике, целевым адресом перехода. Затем в следующий раз, когда процессор будет получать инструкцию по адресу, заданному программным счетчиком, вместо этого он извлечет адрес цели перехода.

Условная ветвь

Хотя она имеет тот же базовый формат инструкции, что и безусловный переход (инструкция `#target`), инструкция условного перехода представляет собой

немного сложнее, потому что он предполагает переход к целевому адресу только в том случае, если выполняется определенное условие.

Например, предположим, что мы хотим перейти на новую строку программы, только если результат предыдущей арифметической инструкции равен нулю; если результат отличен от нуля, мы хотим продолжить выполнение в обычном режиме. Мы бы использовали инструкцию условного перехода, которая сначала проверяет, не дала ли ранее выполненная арифметическая инструкция нулевой результат, а затем записывает цель перехода в программный счетчик, если да.

Из-за таких условных переходов нам нужен специальный регистр или набор регистров, в которых можно хранить информацию о результатах арифметических инструкций — например, был ли предыдущий результат нулевым или ненулевым, положительным или отрицательным и так далее.

Разные архитектуры обрабатывают это по-разному, но в нашем DLW-1 это функция регистра слова состояния процессора (PSW). В DLW-1 каждая арифметическая операция после завершения сохраняет различные типы данных о ее результате в PSW. Чтобы выполнить условный переход, DLW-1 должен сначала оценить условие, от которого зависит переход (например, «является ли результат предыдущей арифметической инструкции нулевым?» в предыдущем примере), проверив соответствующий бит в PSW, чтобы убедиться, что это условие истинно или ложно. Если условие ветвления оценивается как истинное, то блок управления заменяет адрес в программном счетчике целевым адресом ветвления. Если условие ветвления оценивается как ложное, то программный счетчик остается как есть, а в следующем цикле выбирается следующая инструкция в нормальной последовательности программы.

Например, предположим, что мы только что вычли число в А из числа в В, и если результат был равен нулю (то есть, если два числа были равны), мы хотим перейти к инструкции по адресу памяти #106. В программе 2-2 показано, как может выглядеть ассемблерный код для такого условного перехода.

Код линии	Комментарии
16	sub A, B, C Вычтите число в регистре А из числа в регистре В и сохраните результат в С.
17 прыжков #106	Проверьте PSW и, если результат предыдущей инструкции был нулевым, перейдите к инструкции по адресу #106. Если результат не равен нулю, перейдите к строке 18.
18	добавить A, B, C Сложить числа в регистрах А и В и сохранить результат в С.

Программа 2-2: Ассемблерный код для условного перехода

Инструкция `jumpz` заставляет процессор проверять PSW, чтобы определить равен ли определенный бит 1 (истина) или 0 (ложь). Если бит равен 1, результатом команды вычитания был 0, и программный счетчик должен быть загружен с целевым адресом перехода. Если бит равен 0, программный счетчик увеличивается, чтобы указать на следующую инструкцию в последовательности (это инструкция добавления в строке 18).

В PSW есть и другие биты, которые определяют другие типы информации о результате предыдущей операции (независимо от того, является ли он положительным или отрицательным, слишком велик для хранения в регистрах и т. д.). Соответственно существуют и другие

типы инструкций условного перехода, которые проверяют эти биты. Например, инструкция `jumpr` выполняет переход к целевому адресу, если результат предыдущей арифметической операции был отрицательным; инструкция `jumpro` переходит к целевому адресу, если результат предыдущей операции был слишком большим и переполнил регистр. Если бы формат инструкций машинного языка DLW-1 мог вместить более восьми возможных инструкций, мы могли бы добавить больше типов условных переходов.

[Инструкции ветвления и цикл выборки-выполнения](#)

Теперь, когда мы рассмотрели основы ветвления, мы можем изменить нашу сводку из трех шагов выполнения программы, включив в нее возможность инструкции ветвления:

1. Получить следующую инструкцию по адресу, хранящемуся в программном счетчике, и загрузить эту инструкцию в регистр инструкций.

Увеличьте счетчик программы.

2. Декодировать инструкцию в регистре инструкций.

3. Выполните инструкцию в регистре инструкций, используя следующие правила:

- a. Если инструкция является арифметической инструкцией, то выполните ее, используя АЛУ и регистровый файл.
- б. Если инструкция является инструкцией доступа к памяти, то выполните ее, используя аппаратное обеспечение памяти.
- в. Если инструкция является инструкцией ветвления, то выполнить ее с помощью блока управления и счетчика команд. (Для выполненной ветви запишите целевой адрес ветви в программный счетчик.)

Короче говоря, можно сказать, что инструкции ветвления позволяют программисту перенаправить процессор, когда он проходит через поток команд. Ветви указывают процессору на разные участки потока кода, манипулируя его блоком управления, который, поскольку он содержит регистр команд и программный счетчик, является рулём процессора.

[Ветвь-инструкция как особый вид нагрузки](#)

Напомним, что выборка инструкций — это особый тип загрузки, который происходит автоматически для каждой инструкции и всегда принимает адрес в программном счетчике в качестве источника, а регистр инструкций — в качестве пункта назначения. Имея это в виду, вы можете думать об инструкции ветвления как об аналогичном типе нагрузки, но под управлением программиста, а не процессора. Инструкция ветвления — это загрузка, которая принимает адрес, указанный параметром `#target`, в качестве источника, а регистр инструкции — в качестве пункта назначения.

Как и при обычной загрузке, инструкция ветвления может принимать в качестве цели адрес хранится в регистре. Другими словами, инструкции ветвления могут использовать относительную адресацию регистров точно так же, как и обычные инструкции загрузки. Эта возможность полезна, поскольку позволяет компьютеру хранить блоки кода в произвольных местах памяти. Программисту не нужно знать адрес, по которому

блок кода завершится до написания инструкции ветвления, которая переходит к этому конкретному блоку; все, что ему или ей нужно, — это способ добраться до области памяти, где операционная система, отвечающая за управление памятью, сохранила начальный адрес нужного блока кода.

Рассмотрим программу 2-3, в которой программист знает, что операционная система поместила адрес цели ветвления в строку 17 регистра С. Достигнув строки 17, компьютер переходит к адресу, хранящемуся в С, путем копирования содержимого С в регистр инструкций.

Код линии	Комментарии
16 суб А, Б, А	Вычтите число в регистре А из числа в регистре В и сохраните результат в А.
17 прыжков #С	Проверьте PSW и, если результат предыдущей инструкции был равен нулю, перейдите к инструкции по адресу, хранящемуся в С. Если результат не равен нулю, перейдите к строке 18.
18 добавить А, 15, А	Добавьте 15 к числу в А и сохраните результат в А.

Программа 2-3: условный переход, использующий адрес, хранящийся в регистре.

Когда программист использует относительную адресацию регистров с инструкцией ветвления, операционная система должна загрузить в определенный регистр базовый адрес сегмента кода, в котором находится программа. Как и сегмент данных, сегмент кода представляет собой непрерывный блок ячеек памяти, но в его ячейках вместо данных хранятся инструкции. Таким образом, чтобы перейти к строке 15 в текущей программе, предполагая, что операционная система поместила базовый адрес сегмента кода в С, программист может использовать следующую команду:

Код	Комментарии
прыжок #(С + 30)	Перейти к инструкции, расположенной в 30 байтах от начала сегмента кода. (Каждая инструкция имеет длину 2 байта, так что получается 15-я инструкция.)

Инструкции и этикетки ответвлений

В программах, написанных для реальных архитектур, цели ветвления обычно не принимают форму ни немедленных значений, ни значений, относительных к регистру. Вместо этого программист помещает метку на строку кода, к которой он или она хочет перейти, а затем помещает эту метку в целевое поле ветви. В программе 2-4 показан фрагмент кода на языке ассемблера, в котором используются метки.

```

суб А, Б, А
прыжок LBL1
добавить А, 15, А
магазин А, #(D + 16)
LBL1: добавить А, В, Б
магазин В, #(D + 16)

```

Программа 2-4: Код на языке ассемблера, использующий метки

В этом примере, если содержимое A и B равно, компьютер перейти к инструкции с меткой LBL1 и начать выполнение там, пропуская инструкции между переходом и добавлением с меткой . Точно так же, как абсолютные адреса памяти, используемые в инструкциях загрузки и сохранения , изменяются во время загрузки, чтобы соответствовать местоположению в памяти сегмента данных программы, метки, такие как LBL1, изменяются во время загрузки на адреса памяти, которые отражают местоположение в памяти кода программы. сегмент.

Экскурс: загрузка

Если вы какое-то время работали с компьютерами, вы слышали термины «перезагрузка » или «загрузка», используемые в связи либо со сбросом компьютера в исходное состояние, либо с его первоначальным включением. Термин загрузка является сокращенной версией термина начальная загрузка, который сам по себе является ссылкой на кажущуюся невыполнимой задачу, которую компьютер должен выполнить при запуске, а именно «подтягиваться за счет собственных загрузочных ремней».

Я говорю «по-видимому, невозможно», потому что при первом включении компьютера в памяти нет программы, но программы содержат инструкции, которые заставляют компьютер работать. Если у процессора нет запущенной программы при первом включении, то как он узнает, откуда взять первую команду?

Решение этой дилеммы заключается в том, что микропроцессор в состоянии по умолчанию при включении жестко запрограммирован на выборку этой первой инструкции по заранее определенному адресу в памяти. Эта первая инструкция, которая загружается в регистр инструкций процессора, является первой строкой программы, называемой BIOS, которая находится в специальном наборе ячеек памяти — небольшом модуле постоянной памяти (ПЗУ), прикрепленном к материнской плате компьютера. Задача BIOS — выполнять базовые тесты оперативной памяти и периферийных устройств, чтобы убедиться, что все работает правильно. Затем процесс загрузки может продолжаться.

В конце программы BIOS находится инструкция перехода, цель которой где находится программа загрузчика . Используя переход, BIOS передает управление системой этой второй программе, чья работа заключается в поиске и загрузке операционной системы компьютера с жесткого диска.

Операционная система (ОС) загружает и выгружает все другие программы, работающие на компьютере, поэтому после запуска ОС компьютер готов взаимодействовать с пользователем.

3

КОНВЕЙЕРНОЕ ВЫПОЛНЕНИЕ

Все процессорные архитектуры, которые вы рассматривали до сих пор, относительно просты и отражают самые ранние этапы эволюции компьютеров. Эта глава приблизит вас к эпохе современных вычислений, представив одно из ключевых нововведений, лежащих в основе быстрого роста производительности, характерного для развития микропроцессоров за последние несколько десятилетий: конвейерное выполнение.

Конвейерное выполнение — это метод, который позволяет разработчикам микропроцессоров увеличить скорость работы процессора, тем самым уменьшая время, необходимое процессору для выполнения программы. В этой главе сначала будет представлена концепция конвейерной обработки посредством аналогии с фабрикой, а затем аналогия будет применена к микропроцессорам. Затем вы узнаете, как оценить преимущества конвейерной обработки, прежде чем я закончу обсуждением ограничений и затрат метода.

ПРИМЕЧАНИЕ. В этой главе обсуждение конвейерного выполнения сосредоточено исключительно на выполнении арифметические инструкции. Инструкции памяти и инструкции ветвления конвейеризированы с использованием тех же фундаментальных принципов, что и арифметические инструкции, и в последующих главах будут рассмотрены особенности фактического процесса выполнения каждого из этих двух типов инструкций.

Жизненный цикл инструкции

Из предыдущей главы вы узнали, что компьютер снова и снова повторяет три основных шага, чтобы выполнить программу:

1. Получить следующую инструкцию по адресу, хранящемуся в программном счетчике, и загрузить эту инструкцию в регистр инструкций.
Увеличьте счетчик программы.
2. Декодировать инструкцию в регистре инструкций.
3. Выполнить инструкцию в регистре инструкций.

Вы также должны помнить, что шаг 3, шаг выполнения, сам по себе может состоять из нескольких подшагов, в зависимости от типа выполняемой инструкции (арифметика, доступ к памяти или переход). В случае арифметической инструкции добавить A, B, C, пример, который мы использовали в прошлый раз, три подэтапа следующие:

1. Считайте содержимое регистров A и B.
2. Добавьте содержимое A и B.
3. Запишите результат обратно в регистр C.

Таким образом, расширенный список действий, необходимых для выполнения арифметической инструкции, выглядит следующим образом (замените любую другую арифметическую инструкцию на добавление в следующем списке, чтобы увидеть, как это выполняется):

1. Получить следующую инструкцию по адресу, хранящемуся в программном счетчике, и загрузить эту инструкцию в регистр инструкций.
Увеличьте счетчик программы.
2. Декодировать инструкцию в регистре инструкций.
3. Выполнить инструкцию в регистре инструкций. Поскольку инструкция является не командой перехода, а арифметической инструкцией, отправьте ее в арифметико-логическое устройство (ALU).
 - a. Прочтите содержимое регистров A и B.
 - b. Добавьте содержимое A и B.
 - c. Запишите результат обратно в регистр C.

На данный момент мне нужно внести изменения в предыдущий список. По причинам, которые мы подробно обсудим, когда будем говорить об окне команд в главе 5, большинство современных микропроцессоров обрабатывают подэтапы За и Зб как группу, а шаг Зс — запись в регистр — отдельно. Чтобы отразить это концептуальное и архитектурное разделение, этот список следует изменить, чтобы он выглядел следующим образом:

1. Получить следующую инструкцию по адресу, хранящемуся в программном счетчике, и загрузить эту инструкцию в регистр инструкций.
Увеличьте счетчик программы.
2. Декодировать инструкцию в регистре инструкций.
3. Выполнить инструкцию в регистре инструкций. Поскольку инструкция является не инструкцией ветвления, а арифметической инструкцией, отправьте ее в АЛУ.
 - а. Прочитайте содержимое регистров А и В.
 - б. Добавьте содержимое А и В.
4. Запишите результат обратно в регистр С.

В современном процессоре эти четыре шага повторяются снова и снова, пока программа не завершит выполнение. Фактически это четыре этапа классического конвейера RISC1. (Вскоре я дам определение термину «конвейер», а пока представьте себе конвейер как серию этапов, через которые должна пройти каждая инструкция в потоке кода при выполнении потока кода.) Вот четыре этапа в их сокращенном виде. форму, форму, в которой вы чаще всего будете их видеть:

1. Получить
2. Расшифровать
3. Выполнить
4. Пишите (или «отписывайтесь»)

Можно сказать, что каждый из этих этапов представляет собой одну фазу жизненного цикла . инструкции. Инструкция начинается с фазы выборки, переходит к фазе декодирования, затем к фазе выполнения и, наконец, к фазе записи. Как я уже упоминал в разделе «Часы» на стр. 29, каждая фаза занимает фиксированное, но не одинаковое количество времени. В большинстве примеров процессоров, с которыми вы будете работать в этой главе, все четыре фазы занимают одинаковое количество времени; обычно это не так в реальных процессорах. В любом случае, если DLW-1 занимает ровно 1 наносекунду (нс) для завершения каждой фазы, то DLW-1 может выполнять одну инструкцию каждые 4 нс.

¹ Термин RISC является акронимом для вычислений с сокращенным набором команд. Я расскажу об этом термине подробнее подробно в главе 5.

Основной поток инструкций

Одно полезное деление, которое компьютерные архитекторы часто используют, когда говорят о ЦП, — это разделение на интерфейсную часть и серверную часть. Как вы уже знаете, когда инструкции извлекаются из основной памяти, их необходимо декодировать для выполнения. Эта выборка и декодирование происходят во внешнем интерфейсе процессора.

На рис. 3-1 видно, что передняя часть примерно соответствует блоки управления и ввода-вывода на схеме модели программирования DLW-1 в предыдущей главе. ALU и регистры составляют заднюю часть DLW-1. Инструкции проходят от передней части вниз через заднюю часть, где выполняется работа по обработке чисел.

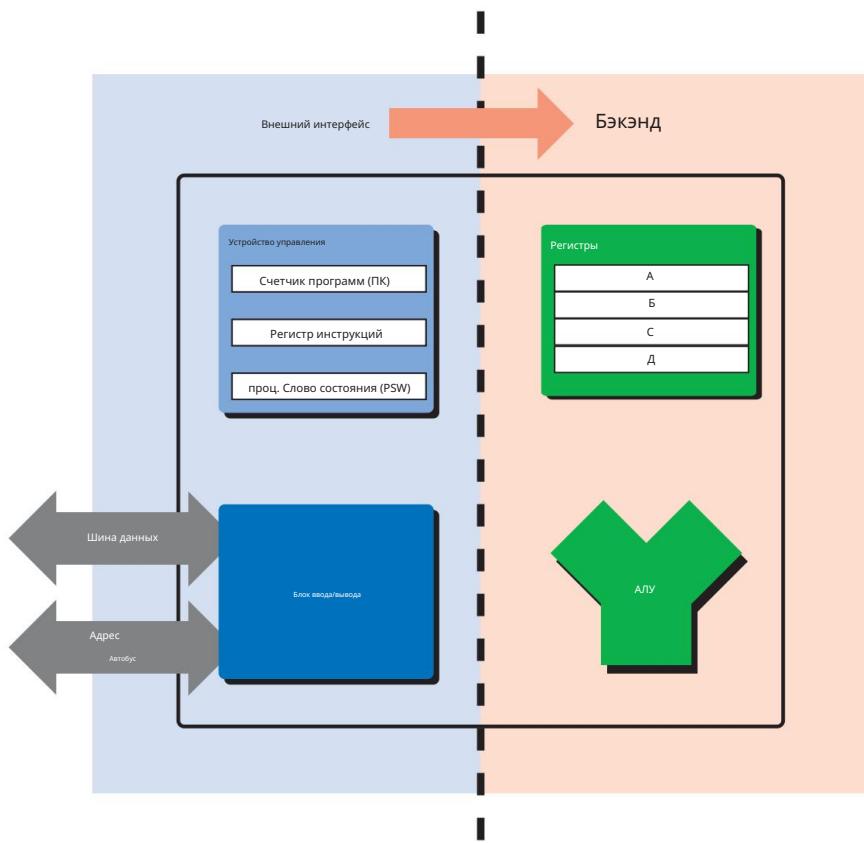


Рисунок 3-1: Передняя часть по сравнению с задней частью

Теперь мы можем изменить рисунок 1-4, чтобы показать все четыре этапа выполнения (см. рисунок 3-2).

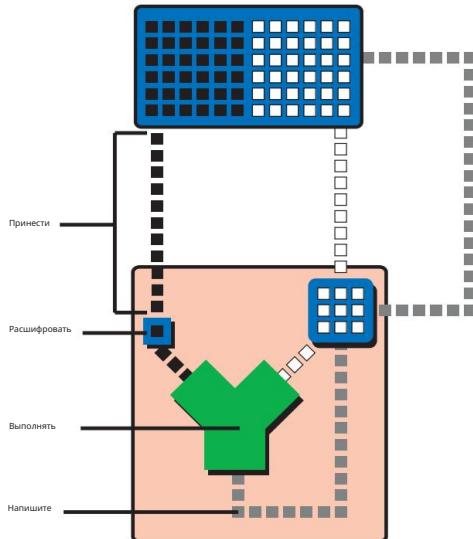


Рисунок 3-2: Четыре фазы выполнения

С этого момента мы собираемся сосредоточиться в первую очередь на потоке кода, и, более конкретно, о том, как инструкции поступают и проходят через микропроцессор, поэтому диаграммы должны будут полностью исключить потоки данных и результатов. На рис. 3.3 представлен основной поток инструкций микропроцессора в простой, но легко детализированной форме.

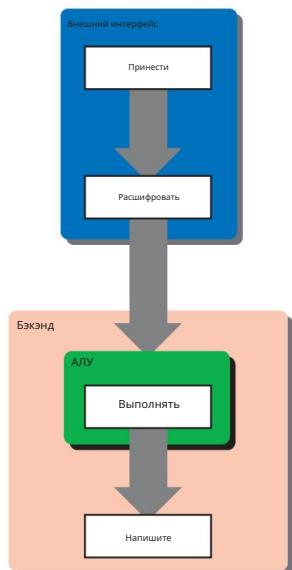


Рисунок 3-3: Основной поток инструкций

На рис. 3-3 инструкции переходят от фаз выборки и декодирования внешнего интерфейса к фазам выполнения и записи серверной части. (Не волнуйтесь, если это покажется вам слишком простым. По мере увеличения уровня сложности обсуждаемых архитектур будет возрастать и сложность диаграмм.)

Объяснение конвейерной обработки

Допустим, мы с друзьями решили заняться производством автомобилей и что нашим первым продуктом должен стать внедорожник (SUV). После некоторых исследований мы определили, что процесс создания внедорожника состоит из пяти этапов:

Этап 1: Соберите шасси.

Этап 2: Вставьте двигатель в шасси.

Этап 3: Установить двери, капот и обшивку на шасси.

Этап 4: Прикрепите колеса.

Этап 5: Покраска внедорожника.

Каждый из этих этапов требует привлечения высококвалифицированных рабочих с очень специализированным набором навыков — рабочие, хорошо разбирающиеся в сборке шасси, мало разбираются в двигателях, кузовах, колесах или покраске, а также сборщики двигателей, маляры и другие специалисты. экипажи. Поэтому, когда мы делаем нашу первую попытку собрать завод по производству внедорожников, мы нанимаем и обучаем пять бригад специалистов, по одной на каждый этап процесса сборки внедорожников. Одна бригада занимается сборкой шасси, одна — установкой двигателей, одна — установкой дверей, капота и обшивки шасси, еще одна — колесами, и бригада покраски. Наконец, поскольку бригады настолько специализированы и эффективны, каждый этап процесса сборки внедорожника занимает у бригады ровно один час.

Теперь, поскольку мои друзья и я компьютерщики, а не промышленные инженеры, нам нужно было многое узнать об эффективном использовании производственных ресурсов.

Мы построили работу нашего первого завода по следующему плану: поместите все пять бригад в линию в цеху, и пусть первая бригада заведет внедорожник на Этапе 1. После того, как Этап 1 завершен, бригада Этапа 1 проходит частично законченный внедорожник передается бригаде Этапа 2, а затем отправляется в комнату отдыха, чтобы поиграть в мяч, в то время как бригада Этапа 2 собирает двигатель и опускает его. команда Этапа 3 берет на себя управление, а команда Этапа 2 присоединяется к бригаде Этапа 1 в комнате отдыха.

Таким образом, внедорожник движется по линии через все пять стадий. только одна бригада работает на одной сцене в любой момент времени, а остальные бригады простояивают. Как только готовый внедорожник завершает этап 5, экипаж этапа 1 начинает работу над другим внедорожником. При такой скорости на изготовление одного внедорожника уходит ровно пять часов, а наш завод собирает один внедорожник каждые пять часов.

На рис. 3-4 вы можете увидеть, как внедорожник проходит все пять стадий. Внедорожник въезжает в заводской цех в начале первого часа, где над ним начинает работу бригада Этапа 1. Обратите внимание, что все остальные бригады сидят без дела, пока бригада Этапа 1 выполняет свою работу. В начале второго часа команда Этапа 2 вступает во владение, а остальные четыре бригады бездействуют в ожидании.

Стадия 2. Этот процесс продолжается по мере движения внедорожника по линии до тех пор, пока в начале шестого часа один внедорожник не останется завершенным, а другой не перейдет на этап 1.

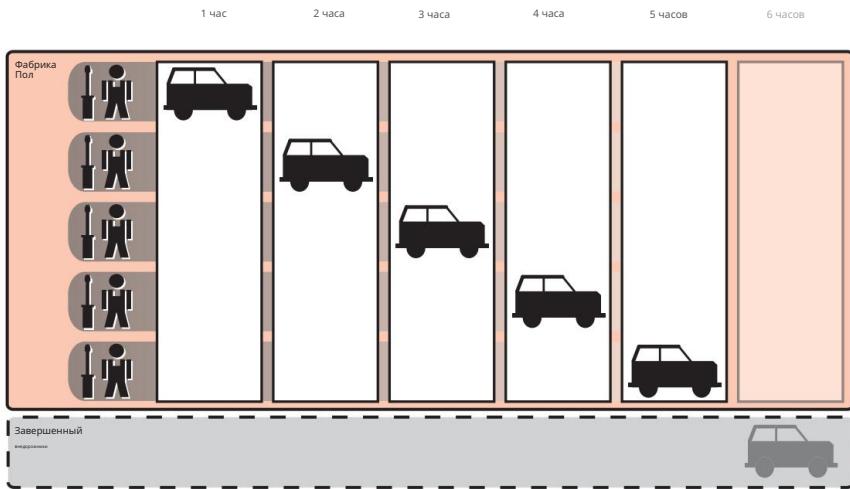


Рисунок 3-4: Жизненный цикл внедорожника на неконвейерном заводе

Перенесемся на год вперед. Наш внедорожник Extinction LE продается как . . . ну, он продается как внедорожник, а это значит, что у него все хорошо. На самом деле, наш внедорожник продается настолько хорошо, что мы привлекли внимание военных и нам предложили контракт на поставку внедорожников армии США на постоянной основе. Армия любит заказывать несколько внедорожников одновременно; один заказ может прийти на 10 внедорожников, а другой — на 500 внедорожников. Чем больше таких заказов мы сможем выполнить за каждый финансовый год, тем больше денег мы сможем заработать за тот же период и тем лучше будет выглядеть наш баланс. Это, конечно, означает, что нам нужно найти способ увеличить количество внедорожников, которые наш завод может выпускать в час, известное как скорость выпуска внедорожников нашего завода. Выполняя больше внедорожников в час, мы можем быстрее выполнять заказы армии и ежегодно зарабатывать больше денег.

Самый интуитивный способ увеличения парка внедорожников на нашем заводе. ставка заключается в том, чтобы попытаться сократить время производства каждого внедорожника. Если мы сможем заставить бригады работать в два раза быстрее, наш завод сможет производить в два раза больше внедорожников за то же время. Однако наши бригады уже работают изо всех сил, поэтому, если не произойдет технологический прорыв, который повысит их производительность, этот вариант пока не рассматривается.

Поскольку мы не можем ускорить работу наших бригад, мы всегда можем использовать метод грубой силы и просто выкинуть деньги на решение проблемы, построив вторую сборочную линию. Если мы найдем и обучим пять новых бригад, чтобы сформировать вторую сборочную линию, которая также способна производить один автомобиль каждые пять часов, мы сможем собирать в общей сложности два внедорожника каждые пять часов из заводского цеха — вдвое больше, чем у нас. настоящий завод. Однако это не похоже на очень эффективное использование заводских ресурсов, поскольку у нас не только в два раза больше бригад, работающих одновременно, но и в два раза больше бригад одновременно в комнате отдыха. Там должен быть лучший способ.

Столкнувшись с отсутствием вариантов, мы нанимаем команду консультантов, чтобы найти разумный способ повысить общую производительность завода без удвоения числа бригад или увеличения производительности каждой отдельной бригады. Спустя год и тысячи оплачиваемых часов консультанты нашли решение.

Зачем позволять нашим бригадам проводить четыре пятых своего рабочего дня в комнате отдыха, когда они могут в это время заниматься полезной работой? При правильном графике работы существующих пяти бригад наш завод может выпускать один внедорожник каждый час, что значительно повышает как эффективность, так и производительность нашей сборочной линии. Пересмотренный рабочий процесс будет выглядеть следующим образом:

1. Бригада Этапа 1 строит шасси.
2. Когда шасси готово, они отправляют его бригаде Этапа 2.
3. Бригада Этапа 2 получает шасси и начинает установку двигателя, а бригада Этапа 1 начинает работу на новом шасси.
4. Когда бригады Этапа 1 и 2 закончат работу, работа бригады Этапа 2 переходит к Этапу 3, работа бригады Этапа 1 переходит к Этапу 2, а бригада Этапа 1 приступает к работе на новом шасси.

Рисунок 3-5 иллюстрирует этот рабочий процесс в действии. Обратите внимание, что у нескольких бригад одновременно на заводе работает несколько внедорожников. Сравните этот рисунок с рис. 3-4, где одновременно работает только одна бригада и только один внедорожник находится в цеху одновременно.

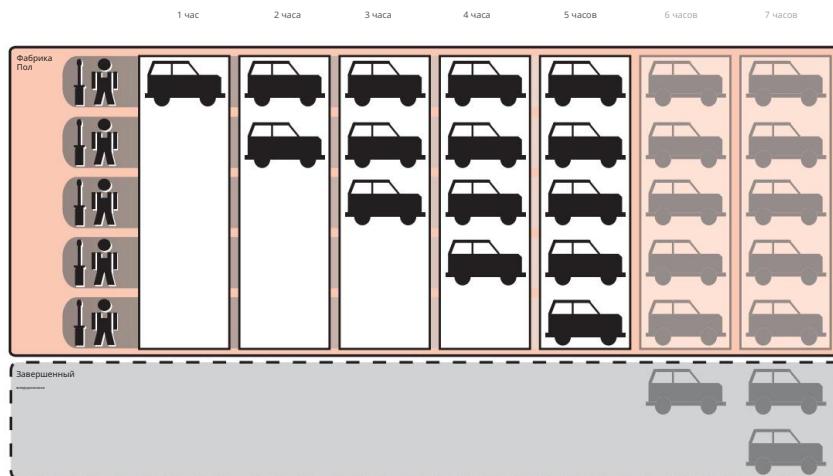


Рисунок 3-5: Жизненный цикл внедорожника на конвейерной фабрике

По мере того, как сборочная линия начинает заполняться внедорожниками на разных стадиях производства, все больше бригад начинают работать одновременно, пока все бригады не будут работать над другим автомобилем на другой стадии производства.

(Конечно, именно так большинство из нас сегодня, в эпоху после Форда, ожидают, что хорошая, эффективная сборочная линия будет работать.) Если мы сможем поддерживать сборочную линию полной и обеспечить одновременную работу всех пяти бригад, мы сможем производить один внедорожник, каждый час: пятикратное улучшение скорости выполнения внедорожников по сравнению с предыдущей скоростью выполнения одного внедорожника каждые пять часов. Это, в двух словах, конвейерная обработка.

Хотя общее количество времени, затрачиваемого на производство каждого отдельного внедорожника, не изменилось по сравнению с первоначальными пятью часами, скорость, с которой фабрика в целом собирает внедорожники, резко увеличилась. Кроме того, скорость, с которой завод может выполнять заказы армии на партии внедорожников, также резко возросла. Конвейерная обработка творит чудеса, оптимально используя уже существующие ресурсы. Нам не нужно ни ускорять каждый отдельный этап производственного процесса, ни резко увеличивать количество ресурсов, которые мы бросаем на решение проблемы; все, что необходимо, — это получить больше работы из ресурсов, которые уже есть.

ПОЧЕМУ ЗАВОД ВНЕДОРОЖНИКОВ?

В предыдущем обсуждении для объяснения конвейерной обработки используется аналогия с фабрикой. В других книгах для объяснения этой техники используются более простые аналогии, например, стирка белья, но есть несколько причин, по которым я выбрал более сложную и длинную аналогию, чтобы проиллюстрировать относительно простую концепцию. Во-первых, в этой книге я использую аналогии с фабриками, потому что читатели легко могут себе представить фабрики, основанные на сборочных линиях, и есть много места для того, чтобы дополнить ментальный образ интересными способами, чтобы сделать множество связанных моментов. Во-вторых, и, возможно, даже более важно, многие проблемы, связанные с планированием, очередями и управлением ресурсами, с которыми сталкиваются проектировщики фабрик, имеют прямые аналогии в компьютерной архитектуре. Во многих случаях проблемы и решения точно такие же, просто переведенные в другую область. (Подобные пары проблема/решение, связанные с очередями, также возникают в сфере услуг, поэтому аналогии с супермаркетами и ресторанами быстрого питания также являются моими любимыми.)

Применение аналогии

Возвращаясь к нашему обсуждению микропроцессоров, должно быть легко увидеть, как эта концепция применима к четырем фазам жизненного цикла инструкции. Точно так же, как владельцы завода в нашей аналогии хотели увеличить количество внедорожников, которые завод мог бы выпустить за определенный период времени, разработчики микропроцессоров всегда ищут способы увеличить количество инструкций, которые ЦП может выполнить за заданный период времени. промежуток времени. Когда вы вспоминаете, что программа — это упорядоченная последовательность инструкций, становится ясно, что увеличение количества инструкций, выполняемых в единицу времени, — это один из способов уменьшить общее количество времени, которое требуется для выполнения программы. (Другой способ уменьшить время выполнения программы — уменьшить количество инструкций в программе, но в этой главе мы рассмотрим этот подход позже.) По аналогии с программой, программа подобна порядку внедорожников из военный; точно так же, как увеличение производительности нашего завода внедорожников в час позволило нам быстрее выполнять заказы, увеличение скорости выполнения инструкций процессора (количество инструкций, выполняемых в единицу времени) позволяет ему быстрее запускать программы.

Неконвейерный процессор

В предыдущей главе кратко описано, как описанные выше простые процессоры (например, DLW-1) используют часы для определения времени своих внутренних операций. Эти

неконвейерные процессоры работают с одной инструкцией за раз, перемещая каждую инструкцию через все четыре фазы своего жизненного цикла в течение одного тактового цикла. Таким образом, неконвейерные процессоры также называются однотактными . процессоров, потому что для полного выполнения всех инструкций требуется ровно один тактовый цикл (т. е. для прохождения всех четырех фаз их жизненного цикла).

Поскольку процессор выполняет инструкции со скоростью одна за такт, вы хотите, чтобы часы ЦП работали как можно быстрее, чтобы скорость выполнения инструкций процессора была как можно выше.

Таким образом, вам нужно рассчитать максимальное количество времени, которое требуется для выполнения инструкции, и сделать время тактового цикла эквивалентным этому промежутку времени. Так уж получилось, что на гипотетическом примере ЦП четыре фазы жизненного цикла инструкции занимают в общей сложности 4 нс. Поэтому вы должны установить продолжительность тактового цикла ЦП на 4 нс, чтобы ЦП мог завершить жизненный цикл инструкции — от выборки до обратной записи — за один такт. (Тактовый цикл ЦП часто для краткости называют просто часами .)

На рис. 3.6 синяя инструкция покидает область хранения кода, поступает в процессор, а затем продвигается по фазам своего жизненного цикла в течение тактового периода 4 нс, пока в конце четвертой наносекунды не завершит свое выполнение. последний этап, и его жизненный цикл завершен. Конец четвертой наносекунды также является концом первого тактового цикла, поэтому теперь, когда первый тактовый цикл завершен и синяя инструкция завершила свое выполнение, красная инструкция может войти в процессор в начале нового тактового цикла. и пройти тот же процесс. Эта последовательность шагов длительностью 4 нс повторяется до тех пор, пока через 16 нс (или четыре тактовых цикла) процессор не завершит все четыре инструкции со скоростью выполнения 0,25 инструкций/нс (= 4 инструкции/нс).

16 нс.

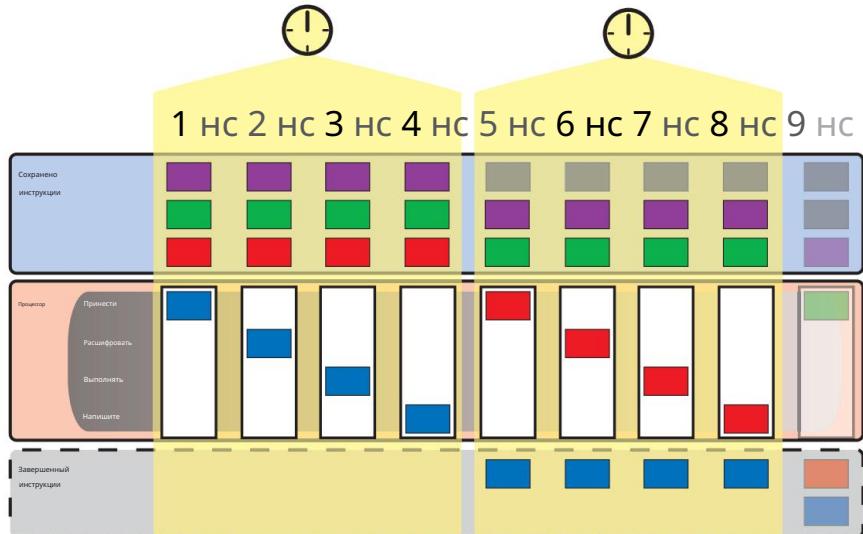


Рисунок 3-6: Однотактный процессор

Однотактные процессоры, подобные изображенному на рис. 3.6, просты в разработке, но они тратят впустую много аппаратных ресурсов. Все это пустое пространство на диаграмме представляет аппаратное обеспечение процессора, которое бездействует, ожидая завершения выполнения инструкции, находящейся в данный момент в процессоре. Конвейеризируя процессор на этом рисунке, вы можете задействовать все больше аппаратных средств каждую наносекунду, тем самым повышая эффективность процессора и его производительность при выполнении программ.

Прежде чем двигаться дальше, я должен прояснить несколько концепций, показанных на рис. 3-6. Внизу находится область с надписью «Выполненные инструкции». Завершенные инструкции на самом деле никуда не уходят после завершения их выполнения; как только они выполнили свою работу по указанию процессору, как модифицировать поток данных, они просто удаляются из процессора. Таким образом, поле «Завершенные инструкции» не представляет собой реальную часть компьютера, поэтому я обвел его пунктирной линией. Эта область — просто место, где вы можете отслеживать, сколько инструкций процессор выполнил за определенный промежуток времени, или скорость выполнения инструкций процессора (или , для краткости, скорость выполнения), чтобы при сравнении различных типов процессоров , у вас будет место, где вы сможете быстро увидеть, какой процессор работает лучше. Чем больше инструкций процессор выполняет за заданный промежуток времени, тем лучше он справляется с программами, которые представляют собой упорядоченную последовательность инструкций. Думайте о поле «Выполненные инструкции» как о своего рода табло для отслеживания скорости выполнения каждого процессора, и установите флажок на каждом из последующих рисунков, чтобы увидеть, сколько времени требуется процессору для заполнения этого поля.

Следуя предыдущему пункту, вам может быть любопытно, почему синяя инструкция, которая завершилась за четвертую наносекунду, не появляется в поле «Завершенные инструкции» до пятой наносекунды.

Причина проста и проистекает из характера диаграммы. Поскольку инструкция тратит одну полную наносекунду, от начала до конца, на каждом этапе выполнения, синяя инструкция входит в фазу записи в начале четвертой наносекунды и выходит из фазы записи в конце четвертой наносекунды. Это означает, что пятая наносекунда — это первая полная наносекунда, в течение которой выполняется синяя инструкция. Таким образом, в начале пятой наносекунды (что совпадает с концом четвертой наносекунды) процессор выполнил одну инструкцию.

[Конвейерный процессор](#)

Конвейеризация процессора означает разбиение процесса выполнения инструкций — то, что я называю жизненным циклом инструкции — на серию дискретных этапов конвейера , которые могут последовательно выполняться специализированным оборудованием. Вспомните, как мы разбили процесс сборки внедорожника на пять отдельных этапов, причем для выполнения каждого этапа была назначена одна бригада. и вы получите идею.

Поскольку жизненный цикл инструкции состоит из четырех довольно разных фаз, вы можете начать с разбивки выполнения инструкции однотактным процессором.

процесса в последовательность из четырех отдельных этапов конвейера, где каждый этап конвейера соответствует фазе стандартного жизненного цикла инструкции:

Этап 1: Получить инструкцию из хранилища кода.

Этап 2: Расшифруйте инструкцию.

Этап 3: Выполните инструкцию.

Этап 4: Запишите результаты инструкции обратно в регистровый файл.

Обратите внимание, что количество стадий конвейера называется глубиной конвейера. Таким образом, у четырехэтапного конвейера глубина конвейера равна четырем.

Для удобства предположим, что каждая из этих четырех стадий конвейера занимает ровно 1 нс, чтобы закончить свою работу над инструкцией, точно так же, как каждой бригаде в нашей аналогии с конвейером требуется один час, чтобы закончить свою часть работы над внедорожником. Таким образом, процесс выполнения исходного однотактного процессора длительностью 4 нс теперь разбит на четыре дискретных последовательных этапа конвейера по 1 нс каждый.

Теперь давайте вместе рассмотрим другую диаграмму, чтобы увидеть, как конвейерная ЦП будет выполнять четыре инструкции, изображенные на рис. 3-7.

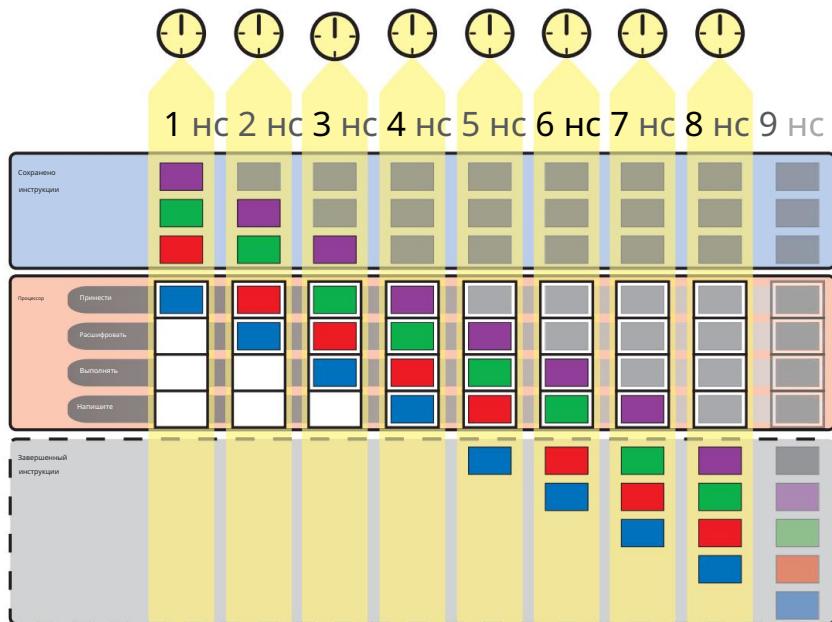


Рисунок 3-7: Четырехэтапный конвейер

В начале первой наносекунды синяя инструкция переходит в стадию выборки. После завершения этой наносекунды начинается вторая наносекунда, и синяя инструкция переходит к этапу декодирования, а следующая инструкция, красная, начинает свой путь из хранилища кода в процессор (т. е. входит в этап выборки). . В начале третьей наносекунды синяя инструкция переходит к этапу выполнения, красная инструкция переходит к этапу декодирования, а зеленая инструкция переходит к этапу выборки. На четвертой наносекунде синяя инструкция переходит к записи

На этапе красная инструкция переходит на этап выполнения, зеленая инструкция переходит на этап декодирования, а фиолетовая инструкция переходит на этап выборки. После того, как четвертая наносекунда полностью истекла и началась пятая наносекунда, синяя инструкция прошла из конвейера и завершила выполнение. Таким образом, мы можем сказать, что по истечении 4 нс (= четырех тактов) конвейерный процессор, изображенный на рис. 3-7, выполнил одну инструкцию.

В начале пятой наносекунды конвейер заполнен, и процессор может начать выполнять инструкции со скоростью одна инструкция в наносекунду.

Эта скорость выполнения одной инструкции/нс является четырехкратным улучшением по сравнению со скоростью выполнения однотактного процессора, равной 0,25 инструкции/нс (или четыре инструкции каждые 16 нс).

[Уменьшение часов](#)

На рис. 3-7 видно, что роль тактового генератора процессора немного меняется в конвейерном процессоре по сравнению с однотактным процессором, показанным на рис. 3-6. Поскольку все этапы конвейера теперь должны работать вместе одновременно и быть готовыми в начале каждой новой наносекунды передать результаты своей работы следующему этапу конвейера, часы необходимы для координации деятельности всего конвейера. Это делается очень просто: сократить время тактового цикла, чтобы оно соответствовало времени, которое требуется каждому этапу для завершения своей работы, чтобы в начале каждого тактового цикла каждый этап конвейера передал инструкцию, над которой он работал, следующему этапу. в трубопроводе.

Поскольку для завершения работы каждого этапа конвейера в примере процессора требуется 1 нс, вы можете установить длительность тактового цикла равной 1 нс.

Этот новый метод тактирования процессора означает, что новая инструкция не обязательно будет выполняться в конце каждого тактового цикла, как это было в случае с однотактным процессором. Вместо этого новая инструкция будет завершена по завершении только тех тактов, в которых стадия записи работала над командой. Любой тактовый цикл с пустой стадией записи не добавит новых инструкций в поле «Завершенные инструкции», а любой тактовый цикл с активной стадией записи добавит в поле одну новую инструкцию. Конечно, это означает, что когда конвейер впервые начнет работать над программой, будет несколько тактов — точнее три, — в течение которых никакие инструкции не будут выполнены. Но как только начинается четвертый такт, первая инструкция переходит в стадию записи, и конвейер может начать выполнение новых инструкций в каждом такте, что, поскольку каждый такт составляет 1 нс, соответствует скорости выполнения одной инструкции в наносекунду.

[Сокращение времени выполнения программы](#)

Обратите внимание, что общее время выполнения каждой отдельной инструкции не изменяется при конвейерной обработке. По-прежнему требуется инструкция 4 нс, чтобы пройти весь путь через процессор; что 4 нс можно разделить на четыре такта по 1 нс каждый, или он может покрыть один более длинный такт, но это все те же 4 нс. Таким образом, конвейерная обработка не ускоряет время выполнения инструкций, но ускоряет время выполнения программы (количество наносекунд, которое требуется для выполнения всей программы) за счет увеличения количества инструкций, выполняемых на единицу.

времени. Точно так же, как конвейерная обработка нашей гипотетической сборочной линии внедорожников позволила нам выполнять заказы армии за более короткий промежуток времени, даже несмотря на то, что каждый отдельный внедорожник по-прежнему провел на сборочной линии в общей сложности пять часов, конвейерная обработка позволяет процессору выполнять программы за короткое время. меньшее количество времени, даже если каждая отдельная инструкция по-прежнему тратит одинаковое количество времени на перемещение через ЦП. Конвейерная обработка позволяет более эффективно использовать существующие ресурсы ЦП, заставляя все его блоки работать одновременно, что позволяет ему выполнять больше общей работы каждую наносекунду.

[Ускорение от конвейерной обработки](#)

В общем, ускорение скорости выполнения по сравнению с реализацией с одним циклом, полученное за счет конвейерной обработки, в идеале равно количеству стадий конвейерной обработки. Четырехступенчатый конвейер дает четырехкратное ускорение скорости выполнения по сравнению с однотактным конвейером, пятиступенчатый конвейер дает пятикратное ускорение, двенадцатиступенчатый конвейер дает двенадцатикратное ускорение и так далее. Это ускорение возможно потому, что чем больше стадий конвейера в процессоре, тем больше инструкций процессор может обрабатывать одновременно и тем больше инструкций он может выполнить за заданный период времени. Таким образом, чем мельче вы сможете разделить эти четыре фазы жизненного цикла инструкции, тем больше оборудования, используемого для реализации этих фаз, вы сможете использовать в любой момент.

Возвращаясь к нашей аналогии со сборочным конвейером, предположим, что каждая бригада состоит из шести рабочих и что каждую из часовных задач, выполняемых каждой бригадой, можно легко разделить на две более короткие 30-минутные задачи. Таким образом, мы можем удвоить производительность нашего завода, разделив каждую бригаду на две более мелкие, более специализированные бригады по три человека в каждой, а затем заставив каждую меньшую бригаду выполнять одну из более коротких задач на одном внедорожнике за 30 минут.

Этап 1: Соберите шасси.

- [z Бригада 1a:](#) Соедините части шасси и выполните точечную сварку соединений.
- [z Бригада 1b:](#) Полностью сварить все части шасси.

Этап 2: Вставьте двигатель в шасси.

- [z Бригада 2a:](#) Поместите двигатель в шасси и установите его на место.
- [z Бригада 2b:](#) Подсоедините двигатель к движущимся частям автомобиля.

Этап 3: Установите двери, капот и обшивку на шасси.

- [z Бригада 3a:](#) Установите двери и капот на шасси.
- [z Бригада 3b:](#) Установите другие покрытия на шасси.

Этап 4: Прикрепите колеса.

- [z Бригада 4a:](#) Прикрепите два передних колеса.
- [z Бригада 4b:](#) Прикрепите два задних колеса.

Этап 5: Покраска внедорожника.

- [z Бригада 5a:](#) Покрасьте борта внедорожника.
- [z Бригада 5b:](#) Покрасьте верхнюю часть внедорожника.

После модификаций, описанных здесь, 10 небольших бригад на нашем заводе теперь имеют в общей сложности 10 внедорожников в работе в течение любого заданного 30-минутного периода. Кроме того, наш завод теперь может выпускать новый внедорожник каждые 30 минут, что в десять раз больше, чем на нашем первом заводе, когда один внедорожник производился каждые пять часов. Таким образом, упорядочив нашу сборочную линию еще более глубоко, мы задействовали еще больше ее рабочих в настоящее время, тем самым увеличив количество внедорожников, над которыми можно работать одновременно, и увеличив количество внедорожников, которые могут быть завершены в течение заданного периода времени.

Углубление конвейера четырехкаскадного процессора работает по аналогии принципов и оказывает аналогичное влияние на показатели завершения. Точно так же, как пять стадий сборочной линии нашего внедорожника можно разбить на более длинную последовательность более специализированных стадий, процесс выполнения, через который проходит каждая инструкция, можно разбить на серию, состоящую не только из четырех отдельных стадий. Разбивая четырехстадийный конвейер процессора на более длинную серию более коротких и более специализированных этапов, еще больше специализированного оборудования процессора может работать одновременно с большим количеством инструкций и, таким образом, увеличивать количество инструкций, которые конвейер выполняет каждую наносекунду.

Сначала мы перешли от однотактного процессора к конвейерному, взяв период времени 4 нс, затраченный инструкцией на прохождение через процессор, и разделив его на четыре дискретных этапа конвейера по 1 нс каждый. Эти четыре дискретных этапа конвейера соответствуют четырем фазам жизненного цикла инструкции. Однако стадии конвейера процессора не всегда точно соответствуют четырем фазам жизненного цикла процессора.

Некоторые процессоры имеют конвейер с пятью этапами, некоторые — с шестью этапами, а многие имеют конвейеры глубже, чем 10 или 20 этапов. В таких случаях разработчик ЦП должен разделить жизненный цикл инструкции на желаемое количество этапов таким образом, чтобы все этапы были одинаковой длины.

Теперь давайте возьмем процесс выполнения длительностью 4 нс и разделим его на восемь дискретных этапов. Поскольку все восемь этапов конвейера должны иметь одинаковую продолжительность для работы конвейера, каждый из восьми этапов конвейера должен иметь длину $4 \text{ нс} \div 8 = 0,5 \text{ нс}$. Поскольку сейчас мы работаем с идеализированным примером, давайте представим, что разделение четырехфазного жизненного цикла процессора на восемь одинаково длинных (0,5 нс) стадий конвейера является тривиальной задачей, и что результаты выглядят так, как показано на рис. 3. 8. (На самом деле эта задача нетривиальна и включает в себя ряд компромиссов. В качестве уступки этой реальности я решил использовать восемь этапов реального конвейера — конвейера MIPS — на рис. 3-8., вместо того, чтобы просто разделить каждый из четырех традиционных этапов на два.)

Поскольку конвейеризация требует, чтобы каждый этап конвейера выполнял ровно один тактового цикла, тактовый цикл теперь может быть сокращен до 0,5 нс, чтобы соответствовать длине восьми этапов конвейера. В нижней части рисунка 3-8 вы можете увидеть влияние увеличения количества стадий конвейера на количество инструкций, выполняемых в единицу времени.

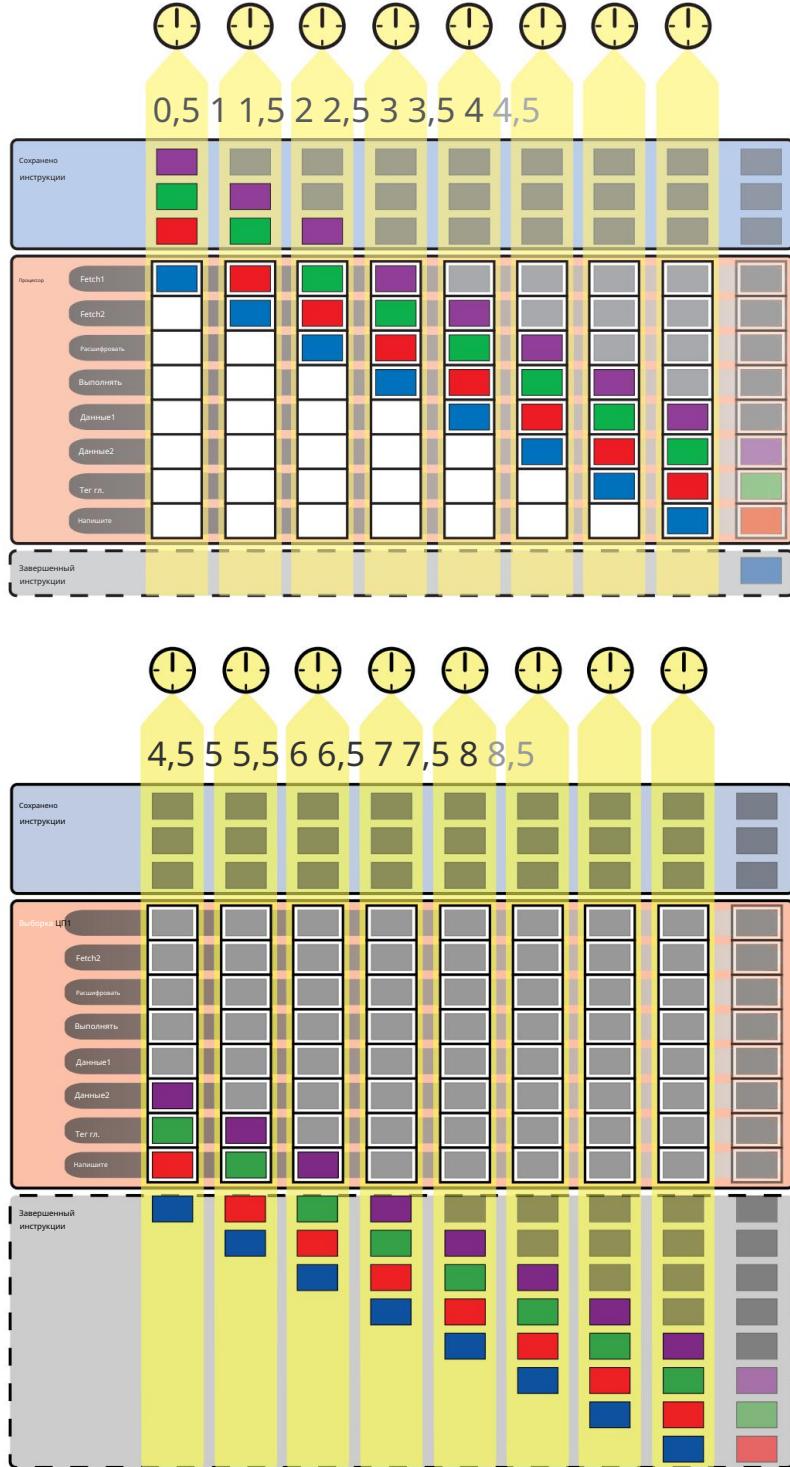


Рисунок 3-8: Восьмиэтапный конвейер

Однотактный процессор может выполнять одну инструкцию каждые 4 нс со скоростью выполнения 0,25 инструкции/нс, а четырехэтапный конвейерный процессор может выполнять одну инструкцию каждую наносекунду со скоростью выполнения одна инструкция/нс. Восьмиступенчатый процессор, изображенный на рис. 3-8, улучшает оба из них, выполняя одну команду каждые 0,5 нс, что обеспечивает скорость выполнения двух инструкций в нс. Обратите внимание, что поскольку выполнение каждой инструкции по-прежнему занимает 4 нс, первые 4 нс восьмистадийного процессора по-прежнему предназначены для заполнения конвейера. Но как только конвейер заполнен, процессор может начать выполнение инструкций в два раза быстрее, чем четырехступенчатый процессор, и в восемь раз быстрее, чем одноступенчатый процессор.

Это восьмикратное увеличение скорости выполнения по сравнению с однотактной конструкцией означает, что восьмиступенчатый процессор может выполнять программы намного быстрее, чем однотактный или четырехэтапный процессор. Но приводит ли восьмикратное увеличение скорости выполнения к восьмикратному увеличению производительности процессора? Не совсем.

[Время выполнения программы и скорость выполнения](#)

Если бы программа, которую выполняет однотактный процессор на рис. 3-6, состояла только из четырех изображенных инструкций, эта программа имела бы время выполнения программы 16 нс, или $4 \text{ инструкции} \div 0,25 \text{ инструкции/нс}$. Если бы программа состояла, скажем, из семи инструкций, время ее выполнения составило бы $7 \text{ инструкций} \div 0,25 \text{ инструкций/нс} = 28 \text{ нс}$. Как правило, время выполнения программы равно общему количеству инструкций в программе, деленному на скорость выполнения инструкций процессора (количество инструкций, выполняемых за наносекунду), как в следующем уравнении:

$$\text{время выполнения программы} = \frac{\text{количество инструкций в программе}}{\text{скорость выполнения инструкций}}$$

В большинстве случаев, когда я говорю в этой книге о производительности процессора, я имею в виду время выполнения программы. Один процессор работает лучше, чем другой, если он выполняет все инструкции программы за меньшее время, поэтому сокращение времени выполнения программы является ключом к повышению производительности процессора.

Из предыдущего уравнения должно быть ясно, что выполнение программы время может быть сокращено одним из двух способов: за счет уменьшения количества инструкций на программу или за счет увеличения скорости выполнения процессора. А пока предположим, что количество инструкций в программе фиксировано и что с этим членом уравнения ничего нельзя поделать. Таким образом, наше внимание в этой главе будет сосредоточено на повышении показателей выполнения инструкций.

В случае неконвейерного однотактного процессора скорость выполнения инструкций (x инструкций на 1 нс) просто обратна времени выполнения инструкции (y нс на 1 инструкцию), где x и y имеют разные числовые значения. ценности. Поскольку взаимосвязь между скоростью выполнения и временем выполнения инструкции проста и прямая в однотактном процессоре,

п - кратное улучшение одного означает п-кратное улучшение другого. Таким образом, повышение производительности однотактного процессора на самом деле связано с сокращением времени выполнения инструкций.

В конвейерных процессорах взаимосвязь между временем выполнения инструкций и скоростью выполнения более сложная. Как обсуждалось ранее, конвейерные процессоры позволяют увеличить скорость выполнения процессора без изменения времени выполнения инструкций. Конечно, сокращение времени выполнения инструкции по-прежнему приводит к увеличению скорости выполнения, но не всегда верно обратное. На самом деле, как вы узнаете позже, повышение скорости выполнения конвейерной обработки часто достигается ценой увеличения времени выполнения инструкций. Это означает, что для повышения производительности конвейерной обработки скорость выполнения процессора должна быть как можно выше в ходе выполнения программы.

Взаимосвязь между скоростью выполнения и временем выполнения программы

Если вы посмотрите на блок «Завершенные инструкции» четырехэтапного процессора на рис. 3.7, то увидите, что всего пять инструкций были выполнены в начале девятой наносекунды. Напротив, неконвейерный процессор, показанный на рис. 3.6, выполняет две завершенные инструкции в начале девятой наносекунды. Пять выполненных инструкций за 8 нс, очевидно, не являются четырехкратным улучшением по сравнению с двумя выполненными инструкциями за тот же период времени, так что же дает?

Помните, что изначально конвейерному процессору потребовалось 4 нс, чтобы заполнить инструкции; конвейерный процессор не завершил свою первую инструкцию до конца четвертой наносекунды. Следовательно, он выполнил меньше инструкций за первые 8 нс выполнения этой программы, чем если бы конвейер был заполнен на все 8 нс.

Когда процессор выполняет программы, состоящие из тысяч инструкций, то по мере того, как количество наносекунд растягивается до тысяч, влияние на время выполнения программы тех четырех начальных наносекунд, в течение которых была выполнена только одна инструкция, начинает исчезать, и конвейерная обработка преимущества процессора начинает приближаться к четырехкратной отметке. Например, через 1000 нс неконвейерный процессор выполнит 250 инструкций ($1000 \text{ нс} \div 0,25 \text{ инструкций/нс} = 250 \text{ инструкций}$), а конвейерный процессор выполнит 996 инструкций [$(1000 \text{ нс} - 4 \text{ нс}) \div 1 \text{ инструкций/нс} = 996 \text{ инструкций}$] — улучшение в 3,984 раза.

То, что я только что описал на этом конкретном примере, — это разница между максимальной теоретической скоростью завершения трубопровода и его реальной средней скоростью завершения. В предыдущем примере максимальная теоретическая скорость выполнения четырехступенчатого процессора, т. е. скорость выполнения в циклах, когда весь его конвейер заполнен, составляет одну команду/нс. Однако средняя скорость выполнения процессора в течение первых 8 нс составляет 5 инструкций/8 нс = 0,625 инструкций/нс. Средняя скорость выполнения процессора улучшается по мере того, как он проходит больше тактовых циклов с заполненным конвейером, пока при 1000 нс его средняя скорость выполнения не составит 996 инструкций/1000 нс = 0,996 инструкций/нс.

На этом этапе полезно взглянуть на график средней скорости выполнения четырехступенчатого конвейера по мере увеличения числа наносекунд, как показано на рис. 3-9.

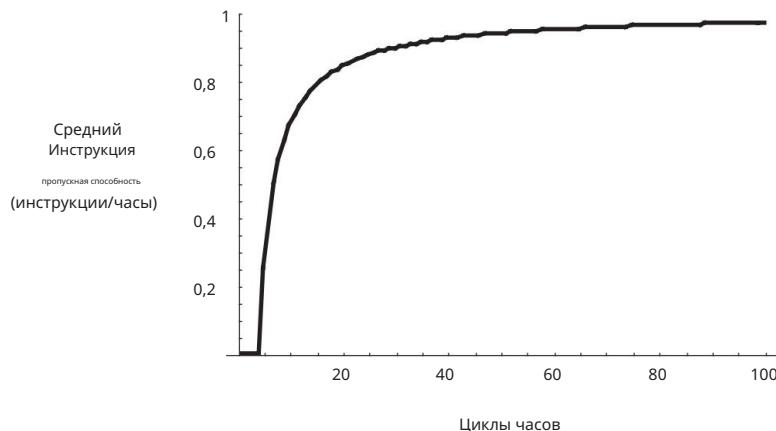


Рисунок 3-9: Средняя скорость завершения четырехступенчатого конвейера

Вы можете видеть, как средняя скорость выполнения процессора остается равной нулю до тех пор, пока отметка 4 нс, после которой конвейер заполняется, и процессор может начинать выполнять новую инструкцию каждую наносекунду, в результате чего средняя скорость выполнения всей программы увеличивается вверх и в конечном итоге приближается к максимальной скорости выполнения одной инструкции/нс .

Таким образом, конвейерный процессор может приблизиться к своей идеальной скорости выполнения только в том случае, если он может длительно работать с полным конвейером на каждом такте.

[Пропускная способность инструкций и остановки конвейера](#)

Однако конвейерная обработка не является полностью «бесплатной». Конвейерная обработка усложняет логику управления микропроцессором, поскольку все эти этапы должны синхронизироваться. Однако еще более важным для настоящего обсуждения является тот факт, что конвейерная обработка усложняет способы оценки производительности процессора.

[Пропускная способность инструкций](#)

До сих пор мы говорили о производительности микропроцессора в основном с точки зрения скорости выполнения инструкций или количества инструкций, которые конвейер процессора может выполнять каждую наносекунду. Более распространенной метрикой производительности в реальном мире является пропускная способность конвейера, или количество инструкций, которые процессор выполняет за каждый такт. Вы можете подумать, что пропускная способность конвейера всегда должна составлять одну инструкцию/такт, потому что ранее я говорил, что конвейерный процессор завершает новую инструкцию в конце каждого тактового цикла , в течение которого был активен этап записи. Но обратите внимание, как подчеркнутая часть этого определения немного уточняет его; вы уже видели, что этап записи неактивен во время

тактовых циклов, в которых конвейер заполняется, поэтому в этих тактовых циклах пропускная способность процессора составляет 0 инструкций/такт. Напротив, когда конвейер инструкций заполнен и стадия записи активна, производительность конвейерного процессора составляет 1 инструкцию/такт.

Таким образом, точно так же, как существовала разница между максимальной теоретической скоростью выполнения процессора и его средней скоростью выполнения, существует также разница между максимальной теоретической пропускной способностью процессора и его средней пропускной способностью:

[Пропускная способность инструкций](#)

Количество инструкций, которые процессор заканчивает выполнять за каждый такт. Вы также увидите пропускную способность инструкций, называемую количеством инструкций за такт (IPC).

[Максимальная теоретическая пропускная способность команд](#)

Теоретическое максимальное количество инструкций, которые процессор может выполнить за каждый такт. Для описанных выше простых видов конвейерных и неконвейерных процессоров это число всегда равно одной инструкции за цикл (одна инструкция/такт или один IPC).

[Средняя пропускная способность инструкций](#)

Среднее количество инструкций за такт (IPC), фактически выполненных процессором за определенное количество циклов.

Пропускная способность процессора тесно связана со скоростью выполнения его инструкций: чем больше инструкций процессор выполняет за каждый такт (инструкций/такт), тем больше инструкций он также выполняет за заданный период времени (инструкций/с).

Мы еще поговорим о взаимосвязи между этими двумя показателями чуть позже, а пока просто помните, что более высокая пропускная способность приводит к более высокой скорости выполнения инструкций и, следовательно, к лучшей производительности.

[Трубопроводные киоски](#)

В реальном мире конвейер процессора может находиться в большем количестве состояний, чем только два описанных до сих пор: полный конвейер или заполняемый конвейер. Иногда инструкции зависают на одном этапе конвейера в течение нескольких циклов. Существует ряд причин, по которым это может произойти — мы обсудим многие из них в этой книге, — но когда это происходит, конвейер, как говорят, останавливается. Когда конвейер останавливается или зависает на определенном этапе, все инструкции на этапах ниже того, на котором произошла остановка, продолжают выполняться в обычном режиме, в то время как остановленная инструкция просто остается на своем этапе, а все инструкции, стоящие за ней, резервируются.

На рис. 3.10 оранжевая инструкция останавливается на два дополнительных цикла на этапе выборки. Поскольку инструкция останавливается, перед ней в конвейере открывается новый пробел для каждого цикла, в котором она останавливается. Как только инструкция снова начинает продвигаться по конвейеру, промежутки в конвейере, созданные остановкой — промежутки, которые обычно называют «конвейерными пузырями», — перемещаются вниз по конвейеру перед ранее остановленной инструкцией, пока они в конечном итоге не покинут конвейер.

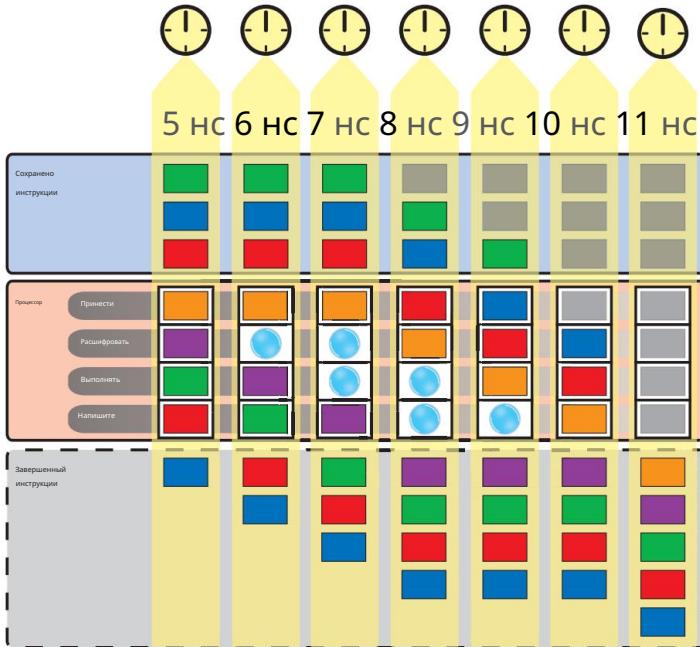


Рисунок 3.10. Задержки конвейера в четырехступенчатом конвейере выглядели бы по-другому без эффекта «пузырей».

Задержки конвейера — или пузыри — снижают среднюю пропускную способность конвейера, потому что они не позволяют конвейеру достичь максимальной пропускной способности одной завершенной инструкции за цикл. На рис. 3.10 оранжевая инструкция застряла на этапе выборки на два дополнительных цикла, создав два пузырька, которые будут распространяться по конвейеру. (Опять же, пузырь — это просто способ показать, что этап конвейера, на котором находится пузырь, не выполняет никакой работы в течение этого цикла.) Как только инструкции под пузырьком завершены, процессор не будет выполнять новые инструкции, пока пузырьки не исчезнут. трубопровода. Таким образом, в конце тактов 9 и 10 в область «Завершенные инструкции» не добавляются новые инструкции; обычно две новые инструкции добавляются в область в конце этих двух циклов. Однако из-за пузырей процессор отстает от графика на две инструкции, когда он достигает 11-го такта и снова начинает накапливать завершенные инструкции.

Чем больше таких пузырей возникает в конвейере, тем дальше фактическая производительность процессора от его максимальной пропускной способности. В предыдущем примере процессор в идеале должен выполнить семь инструкций к тому времени, когда он закончит 10-й такт, при средней пропускной способности 0,7 инструкций за такт. (Помните, что максимально возможная пропускная способность инструкций в идеальных условиях составляет одну инструкцию за такт, но для достижения этого максимума потребуется гораздо больше циклов без пузырей.) Но из-за остановки конвейера процессор выполняет только пять инструкций за 10 тактов, при средней пропускной способности 0,5 инструкций за такт. 0,5 инструкции за такт — это половина

Теоретическая максимальная пропускная способность инструкций, но, конечно, процессор потратил несколько тактов на заполнение конвейера, так что за 10 тактов он не смог бы этого добиться даже в идеальных условиях. Более важным является тот факт, что 0,5 инструкции за такт — это только 71 процент пропускной способности, которой можно было бы достичь, если бы не было задержек (т. е. 0,7 инструкции за такт). Поскольку остановки конвейера снижают среднюю пропускную способность процессора, они увеличивают время, необходимое для выполнения текущей программы.

Если бы программа в предыдущем примере состояла только из семи показанных инструкций, то задержка конвейера привела бы к увеличению времени выполнения программы на 29 %.

Посмотрите на график на рис. 3-11; это показывает, что эта двухтактная остановка делает с средней пропускной способностью команд.

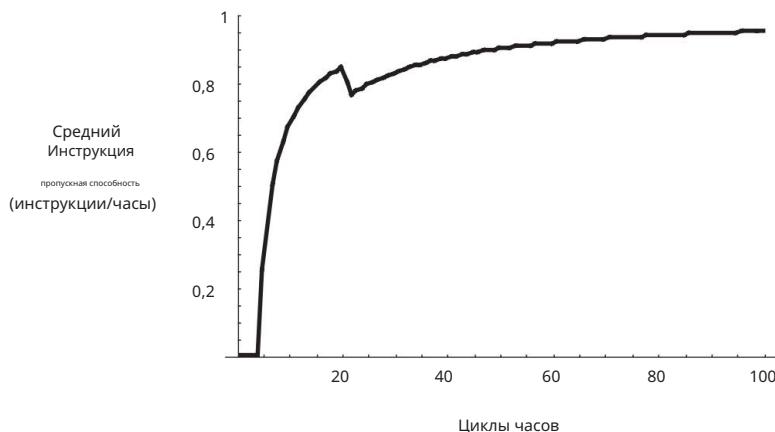


Рисунок 3-11: Средняя пропускная способность четырехступенчатого конвейера с двухтактным остановом.

Средняя производительность процессора перестает расти и начинает резко падать, когда первый пузырек достигает стадии записи, и не восстанавливается до тех пор, пока пузыри не покинут конвейер.

Чтобы получить еще лучшее представление о влиянии простоев на среднюю пропускную способность конвейера, давайте теперь посмотрим, какое влияние окажет 10-тактный останов (начиная с этапа выборки 18-го цикла) в течение 100 циклов.циклов в четырехступенчатом конвейере, описанном до сих пор. Посмотрите на график на рис. 3-12.

После того, как первый пузырек останова достигает стадии записи на 20-м такте, средняя пропускная способность инструкций перестает увеличиваться и начинает уменьшаться. Для каждого такта, в котором есть пузырек на этапе записи, пропускная способность конвейера равна 0 инструкций/такт, поэтому его средняя пропускная способность инструкций за весь период продолжает снижаться. После того, как последний пузырек вышел из стадии записи, конвейер снова начинает выполнять новые инструкции со скоростью одна инструкция/цикл, и его средняя пропускная способность инструкций начинает расти. И когда инструкция процессора через `put` начинает расти, то же самое происходит и с его скоростью выполнения и его производительностью в программах.

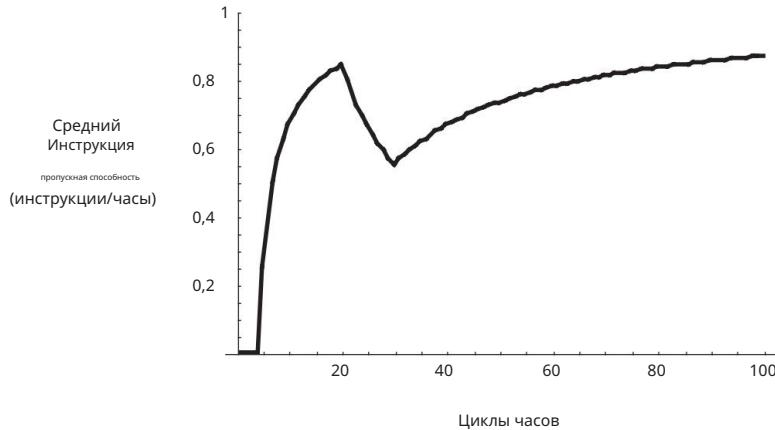


Рисунок 3-12: Средняя производительность команд четырехэтапного конвейера с 10-тактовым остановом

Теперь 10 или 20 циклов остановов здесь и там могут показаться не такими уж большими, но они складываются. Однако еще более важным является тот факт, что числа в предыдущих примерах будут увеличены в 30 или более раз в реальных сценариях выполнения. На момент написания этой статьи процессор может тратить от 50 до 120 наносекунд на ожидание данных из основной памяти. Для процессора с частотой 3 ГГц, время тактового цикла которого составляет доли наносекунды, доступ к основной памяти за 100 нс превращается в несколько тысяч тактовых циклов — и это только для одного обращения к основной памяти из многих миллионов, которые программа может сделать в ходе ее выполнения.

В следующих главах мы рассмотрим причины остановов конвейера и множество приемов, которые используют компьютерные архитекторы для их преодоления.

Задержка инструкций и конвейерные остановки

Прежде чем завершить обсуждение остановов конвейера, я должен ввести еще один термин, который вы будете периодически встречать в остальной части книги: задержка выполнения инструкций. Задержка инструкции — это количество тактов, которое требуется инструкции для прохождения по конвейеру. Для однотактного процессора все инструкции имеют задержку в один такт. Напротив, для описанного выше простого четырехступенчатого конвейера все инструкции имеют задержку в четыре цикла. Чтобы получить наглядное представление об этом, еще раз взгляните на синюю инструкцию на рис. 3-6 ранее в этой главе; эта инструкция занимает четыре такта для продвижения со скоростью один такт на этап через каждый из четырех этапов конвейера. Точно так же инструкции имеют задержку в восемь циклов на восьмиэтапном конвейере, 12 циклов на 12-этапном конвейере и так далее.

В реальных процессорах задержка инструкций не обязательно является фиксированным числом, равным количеству этапов конвейера. Поскольку инструкции могут зависнуть на одном или нескольких этапах конвейера в течение нескольких циклов, каждый дополнительный цикл, который они тратят на ожидание на этапе конвейера, добавляет еще один цикл к их задержке. Таким образом, задержки команд, указанные в предыдущем абзаце (т. е. четыре цикла для четырехэтапного конвейера, восемь циклов для восьмиэтапного конвейера и

так далее) представляют собой минимальные задержки инструкций. Фактические задержки инструкций в конвейерах любой длины могут быть больше, чем глубина конвейера, в зависимости от того, останавливается ли инструкция на одном или нескольких этапах.

[Ограничения конвейерной обработки](#)

Как вы, вероятно, догадываетесь, существуют некоторые практические пределы того, насколько глубоко вы можете конвейеризировать сборочную линию или процессор, прежде чем фактическое ускорение скорости завершения, которое вы получаете от конвейеризации, не станет значительно меньше идеального ускорения, которое вы могли бы ожидать. В реальном мире различные фазы жизненного цикла инструкции нелегко разбить на произвольно большое количество более коротких стадий совершенно одинаковой продолжительности. Некоторые этапы по своей природе более сложны и занимают больше времени, чем другие.

Но поскольку для завершения каждого этапа конвейера требуется ровно один тактовый цикл, тактовый импульс, который координирует все этапы, не может быть быстрее, чем самый медленный этап конвейера. Другими словами, количество времени, необходимое для завершения самого медленного этапа в конвейере, будет определять продолжительность тактового цикла ЦП и, следовательно, продолжительность каждого этапа конвейера. Это означает, что самый медленный этап конвейера будет работать весь такт, в то время как более быстрые этапы будут простоять часть тактового цикла. Это не только тратит ресурсы впустую, но и увеличивает общее время выполнения каждой инструкции, затягивая некоторые фазы жизненного цикла, чтобы они занимали больше времени, чем если бы процессор не был конвейерным — все остальные этапы должны ждать немного больше времени каждый цикл, в то время как самая медленная стадия играет додогнялки.

Таким образом, по мере того, как вы нарежете конвейер более тонко, чтобы добавить этапы и увеличить пропускной способности отдельные этапы становятся все менее и менее однородными по длине и сложности, в результате чего общее время выполнения инструкций процессором увеличивается. Из-за этой особенности конвейерной обработки одна из самых сложных и важных задач, с которой сталкивается разработчик ЦП, состоит в том, чтобы сбалансировать конвейер так, чтобы ни один этап не должен был выполнять больше работы, чем любой другой.

Разработчик должен равномерно распределить работу по обработке инструкции на каждый этап, чтобы ни один этап не занимал слишком много времени и тем самым не замедлял весь конвейер.

[Тактовый период и скорость завершения](#)

Если время тактового цикла конвейерного процессора или тактовый период больше, чем его идеальная длина (т. е. время выполнения неконвейерной инструкции/глубина конвейера), а это всегда так, то скорость выполнения процессора будет страдать. Если пропускная способность инструкций остается фиксированной, скажем, на уровне одной инструкции/такт, то по мере увеличения тактового периода скорость выполнения уменьшается. Поскольку новые инструкции могут выполняться только в конце каждого тактового цикла, более длительный тактовый цикл приводит к меньшему количеству операций, выполняемых за наносекунду, что, в свою очередь, приводит к увеличению времени выполнения программы.

Чтобы лучше понять взаимосвязь между скоростью выполнения, пропускной способностью инструкций и временем тактового цикла, давайте возьмем восьмиэтапный конвейер с рис. 3.8 и увеличим его время тактового цикла до 1 нс вместо 0,5 нс. Тогда его первые 9 нс выполнения будут выглядеть так, как показано на рис. 3-13.

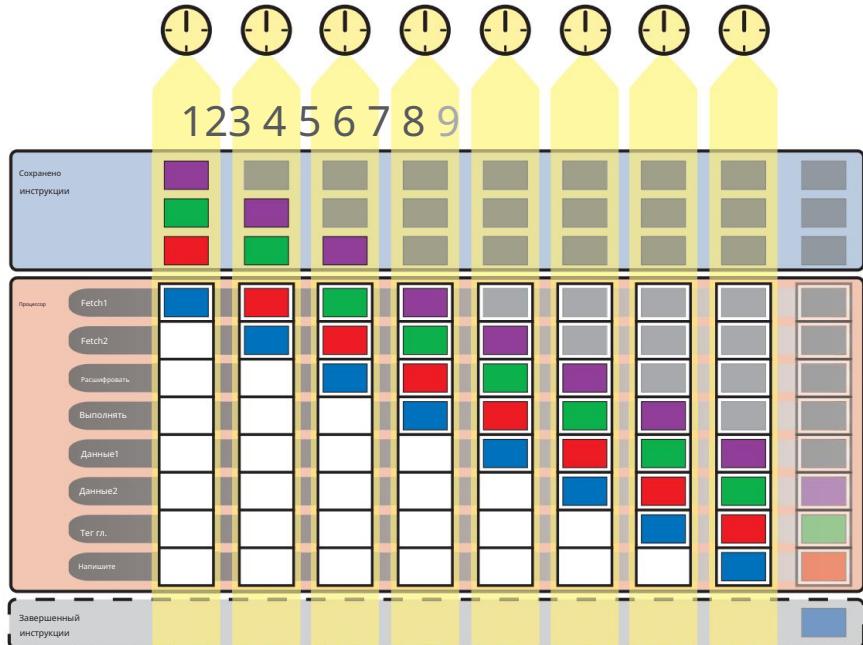


Рис. 3-13. Восьмиэтапный конвейер с тактовым периодом 1 нс.

Как видите, время выполнения инструкции теперь увеличилось с исходного времени 4 нс до нового времени 8 нс, что означает, что конвейер из восьми стадий не завершает свою первую инструкцию до конца восьмой наносекунды. Когда конвейер заполнен, процессор, изображенный на рис. 3.13, начинает выполнять инструкции со скоростью одна инструкция в наносекунду. Эта скорость выполнения составляет половину скорости завершения идеального восьмиступенчатого конвейера с временем тактового цикла 0,5 нс. Это также точно такая же скорость выполнения, как скорость выполнения одной инструкции/нс идеального четырехступенчатого конвейера. Короче говоря, более длительное время тактового цикла нового восьмиступенчатого конвейера лишило более глубокий конвейер его преимущества в скорости выполнения.

Кроме того, восьмиэтапный трубопровод теперь заполняется в два раза дольше.

Взгляните на график на рис. 3-14, чтобы увидеть, как это удвоенное время выполнения влияет на среднюю кривую скорости завершения восьмиэтапного конвейера по сравнению с той же кривой для четырехэтапного конвейера.

Для заполнения более медленного восьмиступенчатого конвейера требуется больше времени, что означает, что его средняя скорость выполнения — и, следовательно, его производительность — увеличивается медленнее, когда конвейер впервые заполняется инструкциями. Существует множество ситуаций, в которых процессор, выполняющий программу, должен полностью очистить свой конвейер, а затем начать его повторное заполнение из другой точки потока кода. В таких случаях эта более медленная кривая скорости завершения приводит к снижению производительности.

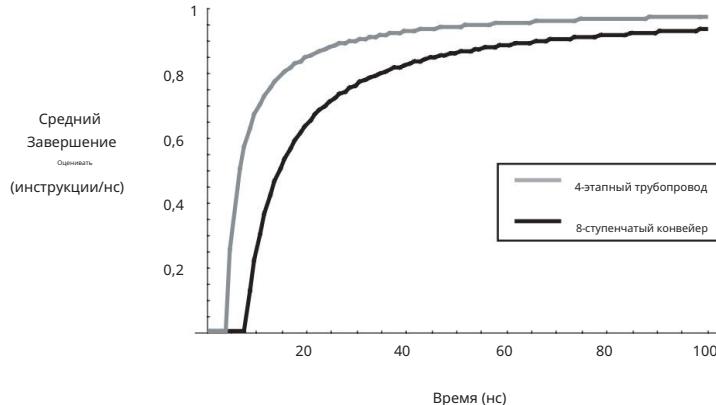


Рисунок 3-14: Средняя скорость выполнения инструкций для четырех- и восьмиступенчатых конвейеров с тактовым периодом 1 нс

В конце концов, прирост производительности, вызванный конвейерной обработкой, зависит от двух вещей:

1. Следует избегать остановок трубопровода. Как вы видели ранее, конвейер останавливается привести к падению скорости выполнения и производительности процессора. Доступ к основной памяти является основной причиной остановок конвейера, но эту проблему можно значительно облегчить с помощью кэширования. Мы подробно рассмотрим кэширование в главе 11. Другие причины зависаний, такие как различные типы опасностей, будут рассмотрены в конце главы 4.
2. Следует избегать повторного заполнения трубопровода. Промывка трубопровода и его повторное заполнение серьезно сказывается как на скорости выполнения, так и на производительности. После того, как конвейер заполнен, он должен оставаться заполненным как можно дольше, чтобы поддерживать среднюю скорость завершения.

Когда в последующих главах мы более внимательно рассмотрим конвейерные процессоры реального мира, вы увидите, что эти две проблемы возникают снова и снова. На самом деле большая часть оставшейся части книги будет посвящена тому, как различные обсуждаемые архитектуры работают, чтобы поддерживать свои конвейеры заполненными, предотвращая простой и обеспечивая непрерывный и непрерывный поток инструкций в процессор из области хранения кода.

[Стоимость конвейерной обработки](#)

В дополнение к неотъемлемым ограничениям повышения производительности, которые мы только что рассмотрели, конвейерная обработка требует нетривиальной дополнительной логики учета и буферизации для реализации, поэтому она влечет за собой накладные расходы на транзисторы и место на кристалле. Кроме того, эти накладные расходы увеличиваются с глубиной конвейера, так что процессор с очень глубоким конвейером (например, Intel Pentium 4) тратит значительную часть своего транзисторного бюджета на связанную с конвейером логику. Эти накладные расходы накладывают некоторые практические ограничения на то, насколько глубоко вы можете конвейеризировать процессор. Я подробнее расскажу о таких ограничениях в главах, посвященных линейке Pentium и Pentium 4.

4

СУПЕРСКАЛЬНОЕ ИСПОЛНЕНИЕ

Главы 1 и 2 описывали процессор так, как его видит программист. Файлы регистров, слово состояния процессора (PSW), арифметико-логическое устройство (ALU) и другие части модели программирования предназначены для того, чтобы предоставить программисту средства для управления процессором и заставить его выполнять полезную работу.

Другими словами, модель программирования по существу представляет собой пользовательский интерфейс для ЦП.

Подобно графическим пользовательским интерфейсам в современных компьютерных системах, под капотом микропроцессора происходит гораздо больше, чем можно было бы предположить из-за простоты модели программирования. В главе 12 я расскажу о различных способах взаимодействия операционной системы и процессора, чтобы обмануть пользователя, заставив его думать, что он выполняет несколько программ одновременно. Подобный трюк используется в программной модели современного микропроцессора, но он предназначен для тог

программисту кажется, что в каждый момент времени происходит только одно, тогда как на самом деле одновременно происходит несколько вещей. Позволь мне объяснить.

В те дни, когда разработчики компьютеров могли разместить относительно небольшое количество транзисторов на одном куске кремния, многие части модели программирования фактически располагались на отдельных микросхемах, прикрепленных к одной печатной плате. Например, одна микросхема содержала АЛУ, другая микросхема содержала блок управления, третья микросхема содержала регистры и так далее. Такие компьютеры были относительно медленными, а тот факт, что они состояли из нескольких микросхем, делал их дорогими. У каждого чипа были свои затраты на производство и упаковку, поэтому чем больше чипов вы устанавливали на плату, тем дороже была система в целом. (Обратите внимание, что это остается верным и сегодня. Стоимость производства систем и компонентов может быть значительно снижена за счет объединения функциональных возможностей нескольких микросхем в одну микросхему.)

С появлением Intel 4004 в 1971 году все изменилось. 4004 был первым в мире микропроцессором на чипе. Разработанный как мозг калькулятора, производимого ныне несуществующей компанией Busicom, 4004 имел 16 четырехбитных регистров, АЛУ, а также логику декодирования и управления, упакованную в один чип из 2300 транзисторов. 4004 был подвигом для своего времени и проложил путь к революции ПК. Однако только четыре года спустя Intel выпустила 8080, и мир увидел первый настоящий ЦП общего назначения.

В течение десятилетий после 8080 количество транзисторов, которые можно было разместить на одном кристалле, росло ошеломляющими темпами. По мере того, как у разработчиков процессоров было все больше и больше транзисторов для работы при разработке новых чипов, они начали придумывать новые способы использования этих транзисторов для повышения производительности вычислений в коде приложений. Одна из первых вещей, которая пришла в голову разработчикам, заключалась в том, что они могут разместить на кристалле более одного ALU и заставить оба ALU работать параллельно для более быстрой обработки кода. Поскольку эти схемы могли выполнять более одной скалярной (или целочисленной, для наших целей) операции одновременно, их называли суперскалярными компьютерами. RS6000 от IBM был выпущен в 1990 году и стал первым в мире коммерчески доступным суперскалярным процессором. Затем в 1993 году компания Intel выпустила процессор Pentium, который с двумя ALU перенес мир x86 в эпоху суперскаляров.

В иллюстративных целях я представлю двухстороннюю суперскалярную версию DLW-1, называемого DLW-2 и показанного на рис. 4-1. DLW-2 имеет два ALU, поэтому он может выполнять две арифметические инструкции параллельно (отсюда и термин «двусторонний суперскаляр»). Эти два ALU совместно используют один и тот же регистровый файл, что в терминах нашей аналогии с файл-клерком соответствует ситуации, когда файл-клерк делит свой личный картотечный шкаф со вторым файл-клерком.

Как вы, вероятно, догадались, глядя на рис. 4-1, суперскалярная обработка немного усложняет конструкцию DLW-2, потому что ему нужна новая схема, позволяющая переупорядочивать поток линейных команд, чтобы некоторые инструкции потока могли выполняться в параллели. Эта схема должна гарантировать «безопасность» отправки двух инструкций параллельно двум исполнительным модулям. Но прежде чем я перейду к обсуждению некоторых причин, по которым параллельное выполнение двух инструкций может быть небезопасным, я должен дать определение только что использованному термину — диспетчеризация.

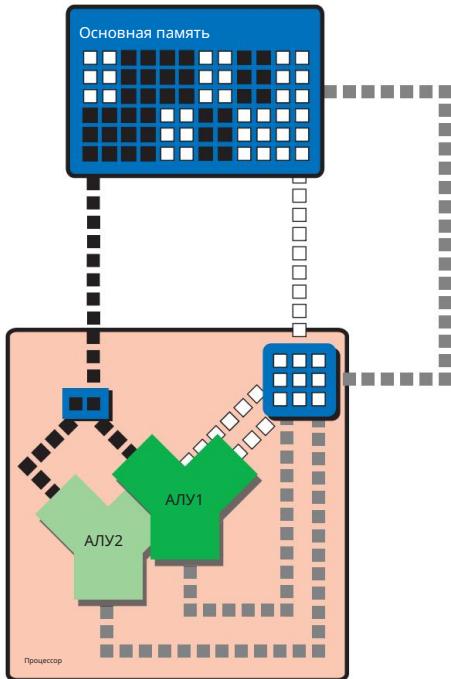


Рисунок 4-1: Суперскаляр DLW-2

Обратите внимание, что на рис. 4-2 я переименовал второй этап конвейера decode/ отправлять. Это связано с тем, что к последней части этапа декодирования прикреплена часть схемы диспетчеризации, задача которой состоит в том, чтобы определить, могут ли две инструкции выполняться параллельно, другими словами, в одном и том же тактовом цикле. Если они могут выполняться параллельно, блок диспетчеризации посыпает одну команду первому целочисленному АЛУ и одну — второму целочисленному АЛУ. Если они не могут быть отправлены параллельно, модуль диспетчеризации отправляет их в программном порядке на первое из двух АЛУ. Есть несколько причин, по которым диспетчер может решить, что две инструкции не могут выполняться параллельно, и мы рассмотрим их в следующих разделах.

Важно отметить, что хотя процессор имеет несколько ALU, модель программирования не меняется. Программист по-прежнему пишет в один и тот же интерфейс, даже несмотря на то, что теперь этот интерфейс представляет принципиально иной тип машины, чем на самом деле является процессор; интерфейс представляет собой машину последовательного выполнения, но процессор на самом деле является машиной параллельного выполнения. Таким образом, несмотря на то, что суперскалярный ЦП выполняет инструкции параллельно, иллюзия последовательного выполнения обязательно должна сохраняться ради программиста. Мы увидим некоторые причины, почему это так, позже, но сейчас важно помнить, что основная память по-прежнему видит один последовательный поток кода, один поток данных и один поток результатов, даже если потоки кода и данных вырезаны. внутри компьютера и протолкнут через два ALU параллельно.

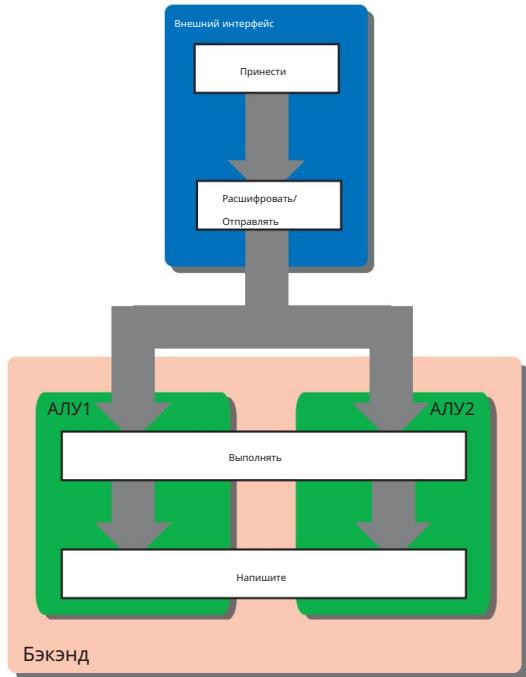


Рисунок 4-2: Конвейер суперскалярного DLW-2

Если процессор должен выполнять несколько инструкций одновременно, он должен иметь возможность получать и декодировать несколько инструкций одновременно. Двунаправленный суперскалярный процессор, такой как DLW-2, может одновременно получать из памяти две инструкции за каждый такт, а также декодировать и отправлять две инструкции за каждый такт. Таким образом, DLW-2 извлекает инструкции из памяти группами по две, начиная с адреса памяти, который отмечает начало сегмента кода текущей программы, и увеличивая программный счетчик на четыре байта вперед каждый раз, когда извлекается новая инструкция. (Помните, что инструкции DLW-2 имеют ширину два байта.)

Как вы можете догадаться, одновременное получение и декодирование двух инструкций усложняет работу DLW-2 с инструкциями ветвления. Что, если первая инструкция в выбранной паре окажется инструкцией ветвления, при которой процессор переходит непосредственно к другой части памяти? В этом случае вторая инструкция в паре должна быть отброшена. Это тратит впустую ширину полосы выборки и создает пузырь в конвейере. Есть и другие проблемы, связанные с суперскалярным выполнением и инструкциями ветвления, и я расскажу о них больше в разделе, посвященном опасностям управления.

Суперскалярные вычисления и IPC

Суперскалярные вычисления позволяют микропроцессору увеличить количество выполняемых инструкций за такт сверх одной инструкции за такт.

Напомним, что максимальная теоретическая пропускная способность конвейерного процессора — одна инструкция за такт, как описано в разделе «Выполнение инструкций» на стр. 53. Поскольку суперскалярная машина может иметь несколько инструкций

в несколько этапов записи в каждом тактовом цикле суперскалярная машина может выполнять несколько инструкций за такт. Если мы адаптируем диаграммы конвейеров из главы 3 для учета суперскалярного выполнения, они будут выглядеть так, как показано на рис. 4-3.

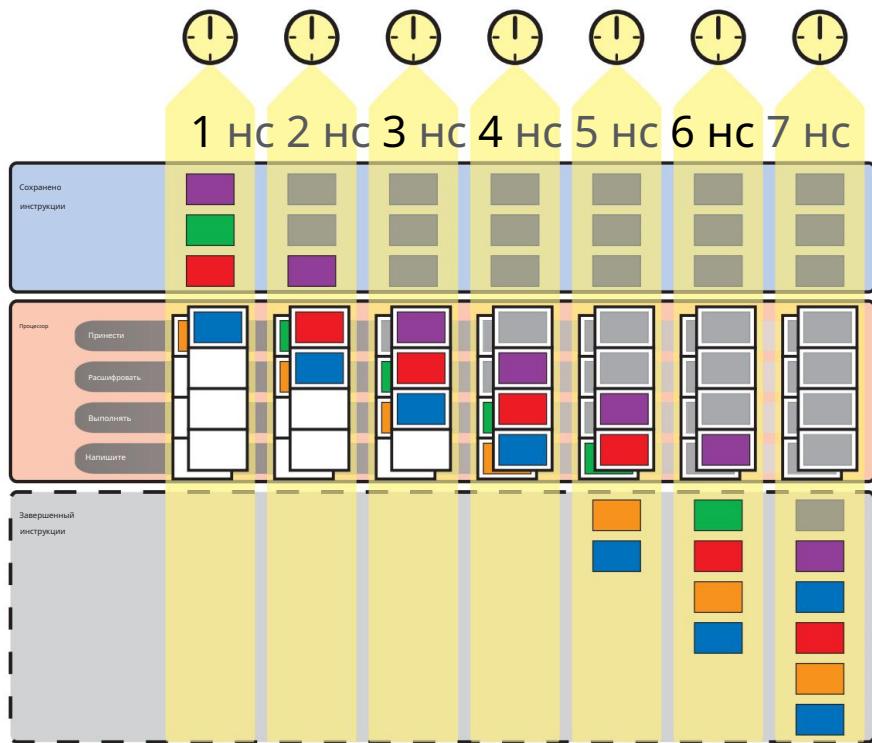


Рисунок 4-3: Объединение суперскалярного исполнения и конвейерной обработки

На рис. 4-3 две инструкции добавлены к завершенным инструкциям .
бок в каждом цикле после заполнения конвейера. Чем больше конвейеров ALU процессор имеет параллельно, тем больше инструкций он может добавить в этот блок за каждый цикл. Таким образом, суперскалярные вычисления позволяют увеличить IPC процессора за счет добавления дополнительного оборудования. Существуют некоторые практические ограничения на количество инструкций, которые могут выполняться параллельно, и мы обсудим их позже.

Расширение суперскалярной обработки с помощью исполнительных модулей

Большинство современных процессоров делают больше с суперскалярным исполнением, чем просто добавляют второе АЛУ. Скорее, они распределяют работу по обработке различных типов инструкций между различными типами исполнительных устройств. Исполнительный блок — это блок схемы в задней части процессора, который выполняет определенную категорию инструкций. Например, вы уже познакомились с арифметико-логическим устройством (АЛУ) — исполнительным устройством, выполняющим арифметические и логические операции над целыми числами. В этом разделе мы более подробно рассмотрим АЛУ, и вы узнаете о некоторых других типах исполнительных устройств для нецелочисленных арифметических операций, доступа к памяти и инструкций ветвления.

Основные числовые форматы и компьютерная арифметика

Типы чисел, с которыми работают современные микропроцессоры, можно разделить на два основных типа: целые числа (также известные как числа с фиксированной запятой) и числа с плавающей запятой. Целые числа — это просто целые числа того типа, с которым вы впервые учились считать в начальной школе. Целое число может быть положительным, отрицательным или нулем, но, конечно, не может быть дробью. Целые числа также называют числами с фиксированной запятой, потому что десятичная точка целого числа не перемещается. Примерами целых чисел являются 1, 0, 500, 27 и 42. Арифметические и логические операции с целыми числами относятся к числу самых простых и быстрых операций, выполняемых микропроцессором. Такие приложения, как компиляторы, базы данных и текстовые процессоры, интенсивно используют операции с целыми числами, потому что числа, с которыми они имеют дело, обычно являются целыми числами.

Число с плавающей запятой — это десятичное число, представляющее дробь.

Примеры чисел с плавающей запятой: 56,5, 901,688 и 41,9999. Как вы можете видеть из этих трех чисел, десятичная точка «плавает» и не фиксируется на одном месте, отсюда и название. Количество знаков после запятой определяет точность числа с плавающей запятой, поэтому числа с плавающей запятой часто являются аппроксимацией дробных значений.

Арифметические и логические операции, выполняемые над числами с плавающей запятой, сложнее и, следовательно, медленнее, чем их целочисленные аналоги. Поскольку числа с плавающей запятой являются аппроксимацией дробных значений, а реальный мир является своего рода приблизительным и дробным, арифметика с плавающей запятой обычно используется в приложениях, ориентированных на реальный мир, таких как моделирование, игры и приложения для обработки сигналов.

И целые числа, и числа с плавающей запятой сами по себе могут быть разделены на один из двух типов: скаляры и векторы. Скаляры — это значения, которые имеют только один числовой компонент, и их лучше всего понять в отличие от векторов.

Вкратце, вектор — это многокомпонентное значение, которое чаще всего рассматривается как упорядоченная последовательность или массив чисел. (Векторы подробно описаны в разделе «Единицы выполнения векторов» на стр. 168.) Вот несколько примеров различных типов векторов и скаляров:

	Целое число	Плавающая точка
Скаляр 14	1.01	
	500	15.234
	37	0,0023
Вектор	{5, 7, 9, 8}	{0,99, 1,1, 3,31}
	{1003, 42, 97, 86, 97}	{50,01, 0,002, 1,4, 1,4}
	{234, 7, 6, 1, 3, 10, 11}	{5.6, 22.3, 44.444, 76.01, 9.9}

Возвращаясь к различию код/данные, мы можем сказать, что поток данных состоит из четырех типов чисел: скалярных целых чисел, скалярных чисел с плавающей запятой, векторных целых чисел и векторных чисел с плавающей запятой. (Обратите внимание, что даже адреса памяти попадают в одну из этих четырех категорий — скалярные целые числа.) Таким образом, кодовый поток состоит из инструкций, которые работают со всеми четырьмя типами чисел.

Виды операций, которые можно выполнять над четырьмя типами чисел, делятся на две основные категории: арифметические операции и логические операции. Когда я впервые представил арифметические операции в главе 1, я для удобства объединил их с логическими операциями.

На этом этапе, однако, полезно отличать два типа операций друг от друга:

- z Арифметические операции — это такие операции, как сложение, вычитание, умножение и деление, все из которых могут быть выполнены с любым типом числа.
- z Логические операции — это логические операции, такие как И, ИЛИ, НЕ и XOR вместе с битовыми сдвигами и поворотами. Такие операции выполняются над скалярными и векторными целыми числами, а также над содержимым специальных регистров, таких как слово состояния процессора (PSW).

Типы операций, выполняемых над этими типами чисел, можно разбить, как показано на рис. 4-4.

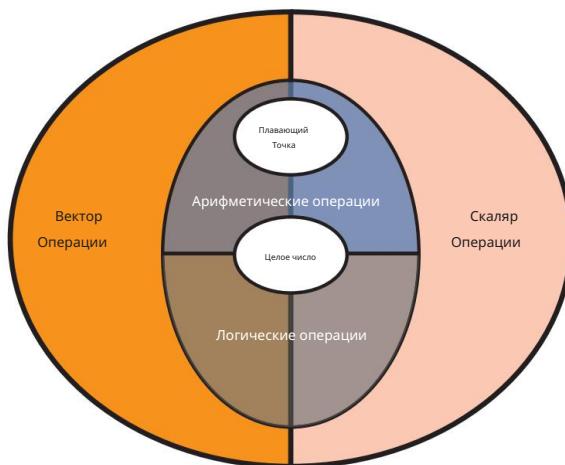


Рисунок 4-4: Числовые форматы и типы операций

По мере того, как вы будете читать остальную часть книги, вам, возможно, захочется время от времени возвращаться к этому разделу. Разные микропроцессоры распределяют эти операции между разными исполнительными блоками по-разному, и все может легко запутаться.

[Арифметико-логические устройства](#)

На ранних микропроцессорах, таких как DLW-1 и DLW-2, все целочисленные арифметические и логические операции обрабатывались АЛУ. Операции с плавающей запятой выполнялись дополнительным чипом, обычно называемым арифметическим сопроцессором, который был прикреплен к материнской плате и предназначен для работы вместе с микропроцессором. Со временем возможности операций с плавающей запятой были интегрированы в ЦП как отдельный исполнительный блок наряду с АЛУ.

Рассмотрим процессор Intel Pentium, изображенный на рис. 4.5, который содержит два целочисленных ALU и ALU с плавающей запятой, а также некоторые другие устройства, которые мы вскоре опишем.

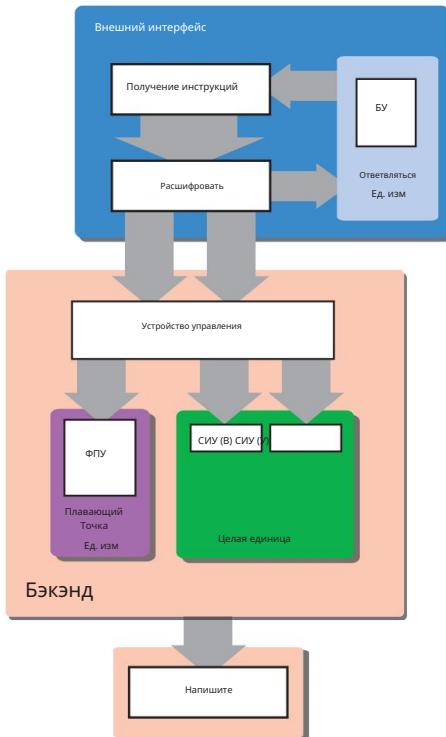


Рисунок 4-5: Intel Pentium

Эта диаграмма представляет собой разновидность рис. 4-2, где стадия выполнения заменена помеченными белыми прямоугольниками (SIU, CIU, FPU, BU и т. д.), которые обозначают тип исполнительного модуля, изменяющего кодовый поток на этапе выполнения.

Заметьте также, что на рисунке немного изменена терминология, которую я должен пояснить, прежде чем мы двинемся дальше.

До сих пор я использовал термин ALU как синоним целочисленного исполнительного блока. Однако из предыдущего раздела мы знаем, что микропроцессор выполняет арифметические и логические операции не только с целочисленными данными, поэтому мы должны быть более точными в нашей терминологии. Отныне ALU — это общий термин для любого исполнительного блока, который выполняет арифметические и логические операции над любым типом данных. Более конкретные метки будут использоваться для идентификации ALU, которые обрабатывают определенные типы инструкций и числовых данных. Например, целочисленный исполнительный блок (IU) — это ALU, который выполняет целочисленные арифметические и логические инструкции, исполнительный блок с плавающей запятой (FPU) — это ALU, который выполняет арифметические и логические инструкции с плавающей запятой, и так далее.

На рис. 4-5 показано, что процессор Pentium имеет два IU — модуль простых целых чисел (SIU) и модуль комплексных целых чисел (CIU) — и один FPU.

Исполнительные блоки могут быть логически организованы в функциональные блоки для простоты ссылки, так что два целочисленных исполнительных устройства могут быть упомянуты

в совокупности как целочисленная единица Pentium. Устройство Pentium с плавающей запятой состоит только из одного FPU, но некоторые процессоры имеют более одного FPU; аналогично с блоком загрузки-накопления (LSU). Блок с плавающей запятой может состоять из двух FPU — FPU1 и FPU2, а блок загрузки-сохранения может состоять из LSU1 и LSU2. В обоих случаях мы часто будем ссылаться на «FPU» или «LSU», когда имеем в виду все исполнительные устройства в этом функциональном блоке, взятые как группа.

Многие современные микропроцессоры также имеют блоки выполнения векторов, которые выполняют арифметические и логические операции над векторами. Однако я не буду здесь подробно описывать векторные вычисления, потому что это обсуждение относится к другой главе.

[Блоки доступа к памяти](#)

Почти во всех процессорах, которые мы рассмотрим в последующих главах, вы увидите пару исполнительных блоков, выполняющих инструкции доступа к памяти: блок загрузки-сохранения и блок выполнения ветвлений. Блок загрузки-сохранения (LSU) отвечает за выполнение инструкций загрузки и сохранения, а также за генерацию адресов. Как упоминалось в главе 1, LSU имеют небольшое упрощенное оборудование для сложения целых чисел, которое может быстро выполнять сложение, необходимое для вычисления адреса.

Блок выполнения ветвей (BEU) отвечает за выполнение условных и инструкций безусловного перехода. BEU серии DLW считывает слово состояния процессора, как описано в главе 1, и решает, следует ли заменить программный счетчик целью перехода. BEU также часто имеет собственный блок генерации адресов для выполнения быстрых расчетов адресов по мере необходимости. Мы еще поговорим об ответвлениях реальных процессоров позже.

[Микроархитектура и ISA](#)

В предыдущем обсуждении суперскалярного исполнения я сделал несколько ссылок на несоответствие между линейной моделью программирования с одним ALU, которую видит программист, и тем, что на самом деле делает аппаратное обеспечение суперскалярного процессора. Пришло время конкретизировать это различие между моделью программирования и фактическим аппаратным обеспечением, введя некоторые концепции и словарь, которые позволят нам более точно говорить о различиях между кажущимся и реальным в компьютерной архитектуре.

В главе 1 была представлена концепция модели программирования как абстрактного представления микропроцессора, которое предоставляет программисту функциональные возможности микропроцессора. Модель программирования DLW-1 состояла из одного целочисленного ALU, четырех регистров общего назначения, счетчика программ, регистра команд, слова состояния процессора и блока управления.

Набор инструкций DLW-1 состоял из нескольких инструкций для работы с различными частями модели программирования: арифметические инструкции (например, добавить и sub) для ALU и регистров общего назначения (GPR), загружать и хранить инструкции по манипуляциям с блоком управления и заполнению георадаров данными,

и инструкции ветки по проверке PSW и смене ПК. Мы можем назвать эту ориентированную на программиста комбинацию модели программирования и набора инструкций архитектурой набора инструкций (ISA).

ISA DLW-1 был прямым отражением его аппаратного обеспечения, которое состоял из одного АЛУ, четырех георадаров, персонального компьютера, PSW и блока управления. Напротив, преемник DLW-1, DLW-2, содержал второе ALU, невидимое для программиста и доступное только для логики декодирования/отправки DLW-2. Логика декодирования / отправки DLW-2 будет проверять пары целочисленных арифметических инструкций, чтобы определить, могут ли они безопасно выполняться параллельно (и, следовательно, вне последовательного порядка выполнения программы). Если бы они могли, он отправил бы их двум целочисленным АЛУ для одновременного выполнения. Теперь DLW-2 имеет ту же архитектуру набора команд, что и DLW-1 — набор инструкций и модель программирования остаются неизменными — но аппаратная реализация этой ISA в DLW-2 существенно отличается тем, что DLW-2 является суперскалярной.

Аппаратная реализация ISA конкретного процессора обычно называется микроархитектурой этого процессора. Мы могли бы назвать ISA, представленную с DLW-1, DLW ISA. Каждая последующая итерация нашей гипотетической линейки компьютеров DLW — DLW-1 и DLW-2 — реализует DLW ISA с использованием другой микроархитектуры. DLW-1 имеет только одно ALU, тогда как DLW-2 представляет собой двухстороннюю суперскалярную реализацию DLW-ISA.

Аппаратное обеспечение Intel x86 претерпело ту же эволюцию: каждое последующее поколение становилось все более сложным, в то время как ISA оставалась в основном неизменной. Относительно включения в Pentium аппаратных средств с плавающей запятой, вы можете задаться вопросом, как программист смог использовать аппаратные средства с плавающей запятой (т. е. FPU плюс регистровый файл с плавающей запятой), если исходная ISA x86 не включала никаких операций с плавающей запятой . операции с точкой или указать любые регистры с плавающей запятой. Разработчикам Pentium пришлось внести следующие изменения в ISA, чтобы приспособить новые функции:

- Г Во-первых, им пришлось изменить модель программирования, добавив FPU и специальные регистры для операций с плавающей запятой. [Z](#) Во-вторых, им пришлось расширить набор инструкций, добавив новую группу арифметические инструкции с плавающей запятой.

Эти типы расширений ISA довольно распространены в компьютерном мире. Intel расширила исходный набор инструкций x86, включив в него расширения x87 с плавающей запятой. x87 включает в себя FPU и файл регистра с плавающей запятой на основе стека, но мы поговорим более подробно об архитектуре x87 на основе стека в следующей главе. Позже Intel снова расширила x86, представив набор инструкций векторной обработки под названием MMX (мультимедийные расширения), а также SSE (потоковые SIMD-расширения) и наборы инструкций SSE2. (SIMD расшифровывается как « одна инструкция, несколько данных» и является другим способом описания векторных вычислений. Мы рассмотрим это более подробно в разделе «Векторные исполнительные блоки» на стр. 168.) Точно так же Apple, Motorola и IBM добавили набор векторные расширения для PowerPC ISA в форме AltiVec, как эти расширения называются Motorola, или VMX, как их называет IBM.

Краткая история ISA

На заре вычислительной техники производители компьютеров, такие как IBM, не создавали целую линейку программно-совместимых компьютерных систем и не нацеливали каждую систему на разное соотношение цена/производительность. Вместо этого каждая из систем производителя была похожа на каждую из сегодняшних игровых консолей, по крайней мере, с точки зрения программиста — программисты писали непосредственно на уникальное оборудование машины, в результате чего программа, написанная для одной машины, не работала ни на конкурирующих машинах, ни на других машинах из другой линейки продукции, выпускавшейся собственной компанией производителя. Точно так же, как Nintendo 64 не будет запускать ни игры PlayStation, ни более старые игры SNES, программы, написанные для одной машины примерно 1960 года, не будут работать ни на какой другой машине, кроме одного конкретного продукта этого конкретного производителя. Модель программирования была разной для каждой машины, и код подходил непосредственно к оборудованию, как ключ к замку (см. рис. 4-6).

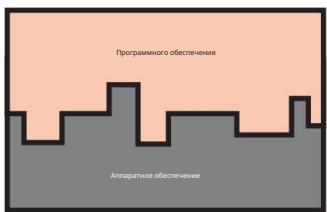


Рис. 4-6. Программное обеспечение настраивалось для каждого поколения аппаратного обеспечения.

Проблемы, связанные с этой ситуацией, очевидны. Каждый раз, когда выходила новая машина, разработчикам программного обеспечения приходилось начинать с нуля. Вы не могли повторно использовать программы, и программистам приходилось изучать тонкости каждого нового устройства, чтобы писать для него код. Это стоило довольно много времени и денег, что делало разработку программного обеспечения очень дорогим мероприятием. Эта ситуация поставила перед разработчиками компьютерных систем следующую проблему: как вы раскрываете (делаете доступными) функциональные возможности ряда связанных аппаратных систем таким образом, чтобы можно было легко разрабатывать программное обеспечение для этих систем и переносить его между этими системами? IBM решила эту проблему в 1960-х, выпустив IBM System/360, которая открыла эру современной компьютерной архитектуры. System/360 представила концепцию ISA как уровня абстракции — или, если хотите, интерфейса — отделенного от микроархитектуры конкретного процессора (см. рис. 4-7). Это означает, что информация, необходимая программисту для программирования машины, была абстрагирована от реальной аппаратной реализации этой машины.

Как только дизайн и спецификация набора инструкций или набора инструкций, доступных программисту для написания программ, были отделены от низкоуровневых деталей конструкции конкретной машины, программы, написанные для конкретной ISA, могли выполняться на любой машине, которая реализовывала это.

Таким образом, ISA предоставила стандартизованный способ представления функций аппаратного обеспечения системы, что позволило производителям вводить новшества и настраивать аппаратное обеспечение для повышения производительности, не беспокоясь о нарушении существующей программной базы. Вы можете выпустить продукт первого поколения с определенной

ISA, а затем работать над ускорением внедрения той же самой ISA для продукта второго поколения, который будет обратно совместим с первым поколением. Сейчас мы принимаем все это как должное, но до IBM System/360 бинарной совместимости между разными машинами разных поколений не существовало.

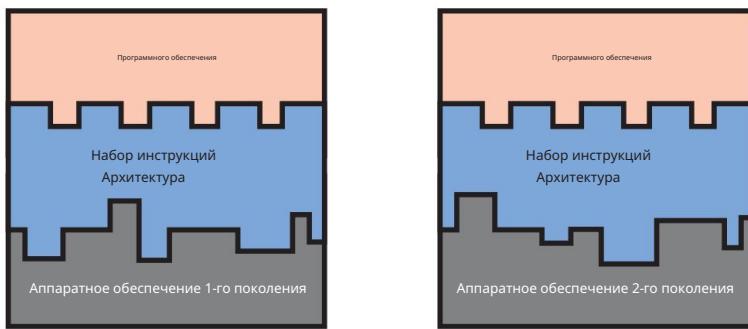


Рисунок 4-7: ISA находится между программным обеспечением и оборудованием, обеспечивая согласованный интерфейс с программным обеспечением для всех поколений оборудования.

Синий слой на рис. 4-7 просто представляет ISA как абстрактную модель машины, для которой программист пишет программы. Как упоминалось ранее, техническое новшество, которое сделало возможным этот абстрактный уровень, называлось механизмом микрокода. Движок микрокода похож на ЦП внутри ЦП. Он состоит из крошечного бита памяти, ПЗУ микрокода, в котором хранятся программы микрокода, и исполнительного блока, который выполняет эти программы. Задача каждой из этих программ микрокода состоит в том, чтобы преобразовать конкретную инструкцию в серию команд, управляющих внутренними частями микросхемы. Когда выполняется инструкция System/360, модуль микрокода считывает инструкцию, обращается к части ПЗУ микрокода, где находится соответствующая программа микрокода этой инструкции, а затем создает последовательность машинных инструкций во внутреннем формате инструкций процессора, которая управляет танцем обращений к памяти и активацией функциональных блоков, который на самом деле выполняет числовую обработку (или что-то еще), что архитектурная инструкция приказала машине сделать.

При таком декодировании инструкций все программы эффективно работают в режиме эмуляции. Это означает, что ISA представляет собой своего рода идеализированную модель, эмулируемую базовым оборудованием, на основе которой программисты могут разрабатывать приложения. Эта эмуляция означает, что между итерациями линейки продуктов поставщик может изменить способ выполнения программы своим процессором, и все, что ему нужно сделать, это каждый раз переписывать программу микрокода, чтобы программисту никогда не приходилось знать об аппаратных различиях, потому что ISA почти не изменилась. Механизмы микрокода все еще используются в современных процессорах. Процессор AMD Athlon использует один для той части своего пути декодирования, которая декодирует более крупные инструкции x86, как и

Ключом к пониманию рис. 4.7 является то, что синий слой представляет собой уровень абстракции, который скрывает от программиста сложность базового оборудования. Синий слой — это не аппаратный уровень (это серый) и не программный уровень (это персиковый), а концептуальный уровень. Думайте об этом как о пользовательском интерфейсе, который скрывает сложность

операционной системы от пользователя. Все, что нужно знать пользователю для использования машины, это как закрывать окна, запускать программы, находить файлы и так далее. Пользовательский интерфейс (и под этим я подразумеваю концептуальную парадигму WIMP — окна, значки, меню, указатель, а не программное обеспечение, реализующее пользовательский интерфейс) предоставляет пользователю возможности и функциональные возможности машины таким образом, который он или она может понять и использовать. И независимо от того, появляется ли этот пользовательский интерфейс на КПК или на настольном компьютере, пользователь все равно знает, как использовать его для управления машиной.

Основным недостатком использования микрокода для реализации ISA является то, что механизм микрокода вначале был медленнее, чем прямое декодирование.
(Современные движки микрокода примерно на 99 процентов быстрее, чем прямое выполнение.) Однако возможность отделить дизайн ISA от микроархитектурной реализации была настолько важной для развития современных вычислений, что небольшое снижение скорости стоило того.

Появление движения вычислений с сокращенным набором команд (RISC) в 1970-х годах привело к нескольким изменениям в схеме, описанной ранее. Прежде всего, RISC был направлен на то, чтобы выбрасывать все за борт во имя скорости.

Итак, первое, что нужно было сделать, это движок микрокода. Микрокод позволил разработчикам ISA разработать наборы инструкций, добавив всевозможные сложные и специализированные инструкции, предназначенные для облегчения жизни программистов, но в действительности редко используемые. Больше инструкций означало, что вам требовалось больше ПЗУ микрокода, что, в свою очередь, означало больший размер кристалла ЦП, более высокое энергопотребление и так далее. Поскольку RISC был больше из меньшего, движок микрокода получил топор. RISC уменьшил количество инструкций в наборе инструкций и уменьшил размер и сложность каждой отдельной инструкции, чтобы этот меньший, более быстрый и более легкий набор инструкций можно было легче реализовать непосредственно на аппаратном уровне без громоздкого механизма микрокода.

Хотя проекты RISC вернулись к старому методу прямого выполнения инструкций, они сохранили концепцию ISA нетронутой. К тому времени компьютерные архитекторы осознали огромную ценность сохранения обратной совместимости со старым программным обеспечением, и они не собирались возвращаться к старым недобрым временам объединения программного обеспечения с одним продуктом. Таким образом, ISA осталась, но в урезанной, значительно упрощенной форме, которая позволила разработчикам реализовать непосредственно в аппаратном обеспечении одну и ту же облегченную ISA на различных типах аппаратного обеспечения.

ПРИМЕЧАНИЕ . Поскольку старые ISA без RISC имели более богатые и сложные наборы инструкций, они были обозначены как ISA со сложным набором инструкций (CISC), чтобы отличить их от новых RISC ISA. x86 ISA — самый популярный пример CISC ISA, а PowerPC, MIPS и Arm — примеры популярных RISC ISA.

[Перенос сложности с аппаратного обеспечения на программное](#)

Машины RISC смогли избавиться от движка микрокода и при этом сохранить преимущества ISA, перенеся сложность с аппаратного обеспечения на программное. В то время как механизм микрокода упростил программирование CISC, предоставив программистам большое разнообразие сложных инструкций, программисты RISC зависели от языков высокого уровня, таких как C, и от компиляторов, чтобы облегчить бремя написания кода для ограниченных наборов инструкций RISC ISA.

Поскольку набор инструкций RISC ISA более ограничен, писать длинные программы на языке ассемблера для процессора RISC сложнее. (Представьте, что вы пытаетесь написать роман, ограничивая себя словарным запасом пятого класса, и вы поймете эту мысль.) Программисту на языке ассемблера RISC может потребоваться использовать множество инструкций для достижения того же результата, что и программисту на языке ассемблера CISC. с одной или двумя инструкциями. Появление языков высокого уровня (HLL), таких как C, и растущая сложность технологии компиляции в совокупности эффективно устранили этот недружественный для программиста аспект вычислений RISC.

ISA была и остается оптимальным решением проблемы легкого и постоянное раскрытие аппаратных функций программистам, чтобы программное обеспечение можно было использовать на самых разных машинах. Величайшим свидетельством мощи и гибкости ISA является долговечность и повсеместное распространение самой популярной и успешной в мире ISA: x86 ISA. Программы, написанные для Intel 8086, чипа, выпущенного в 1978 году, могут работать с относительно небольшими изменениями на новейшем Pentium 4. Однако на микроархитектурном уровне 8086 и Pentium 4 так же отличаются, как Ford Model T и Ford. Мустанг Кобра.

[Проблемы конвейерной обработки и суперскалярного проектирования](#)

Ранее я отмечал, что существуют условия, при которых две арифметические инструкции не могут быть «безопасно» отправлены параллельно для одновременного выполнения двумя АЛУ DLW-2. Такие условия называются опасностями, и все они могут быть отнесены к одной из трех категорий:

- z Опасность данных
- z Структурные опасности
- z Контроль опасностей

Поскольку конвейерная обработка — это форма параллельного выполнения, эти три типа опасностей также могут препятствовать конвейерному выполнению, вызывая появление пузырей в конвейере. В следующих трех разделах я рассмотрю каждый из этих типов опасностей. Я не буду вдаваться в подробности об уловках, которые используют компьютерные архитекторы для их устранения или смягчения их воздействия, потому что мы обсудим их, когда будем рассматривать конкретные микропроцессоры в следующих нескольких главах.

[Опасности данных](#)

Лучший способ объяснить, в чем опасность данных, — проиллюстрировать ее. Рассмотрим программу 4-1:

Номер строки	Код	Комментарии
1	добавить A, B, C	Сложить числа в регистрах A и B и сохранить результат в C.
2	добавить C, D, D	Сложить числа в регистрах C и D и сохранить результат в D.

Программа 4-1: Опасность данных

Поскольку вторая инструкция в программе 4-1 зависит от выходного При первой инструкции две инструкции не могут выполняться одновременно. Скорее, добавление в строке 1 должно завершиться первым, чтобы результат был доступен в C для добавления в строке 2.

Опасности данных являются проблемой как для суперскалярного, так и для конвейерного исполнения. Если программа 4-1 выполняется на суперскалярном процессоре с двумя целочисленными АЛУ, две инструкции сложения не могут выполняться одновременно двумя АЛУ.

Вместо этого АЛУ, выполняющий сложение в строке 1, должен закончить первым, а затем другое АЛУ может выполнить сложение в строке 2. Аналогичным образом, если программа 4.1 выполняется на конвейерном процессоре, второе сложение должно дождаться завершения операции сложения. first add завершает стадию записи, прежде чем перейти к фазе выполнения. Таким образом, схема диспетчеризации должна распознавать зависимость сложения в строке 2 от сложения в строке 1 и не допускать перехода сложения в строке 2 на стадию выполнения до тех пор, пока результат сложения в строке 1 не будет доступен в регистре C.

Большинство конвейерных процессоров могут использовать трюк, называемый переадресацией, который направлен на смягчение последствий этой проблемы. При пересылке процессор берет результат первого добавления из выходного порта АЛУ и возвращает его непосредственно во входной порт АЛУ, минутя стадию записи регистра файла. Таким образом, второе добавление должно дождаться, пока первое дополнение завершит только этап выполнения, а не этапы выполнения и записи, прежде чем оно сможет перейти к самому этапу выполнения.

Переименование регистров — это уловка, помогающая преодолеть опасность данных в суперскалярных вычислениях. машины. Поскольку модель программирования любой данной машины часто определяет меньше регистров, чем может быть реализовано аппаратно, конкретная реализация микропроцессора часто имеет больше регистров, чем количество, указанное в модели программирования. Чтобы получить представление о том, как используется эта группа дополнительных регистров, взгляните на рисунок 4-8.

На рис. 4-8 программист DLW-2 полагает, что он или она использует одно АЛУ с четырьмя архитектурными регистрами общего назначения — A, B, C и D — прикреплен к нему, потому что четыре регистра и один ALU - это все, что определяет модель программирования архитектуры DLW. Однако реальное аппаратное обеспечение суперскалярного DLW-2 имеет два АЛУ и 16 микроархитектурных GPR, реализованных аппаратно. Таким образом, логика переименования регистров DLW-2 может отображать четыре архитектурных регистра на доступные микроархитектурные регистры таким образом, чтобы предотвратить ложные конфликты имен регистров.

На рис. 4.8 инструкция, выполняемая IU1, может думать, что это единственная выполняемая инструкция и что она использует регистры A, B и C, но на самом деле она использует регистры переименования 2, 5 и 10. Аналогично, вторая инструкция, выполняемая одновременно с первой инструкцией, но в IU2 также может думать, что это единственная выполняемая инструкция и что она имеет монополию на файл регистров, но на самом деле она использует регистры 3, 7, 12 и 16. завершили выполнение соответствующих инструкций, логика обратной записи DLW-2 позаботится о передаче содержимого регистров переименования обратно в четыре архитектурных регистра в правильном порядке, чтобы можно было изменить состояние программы.

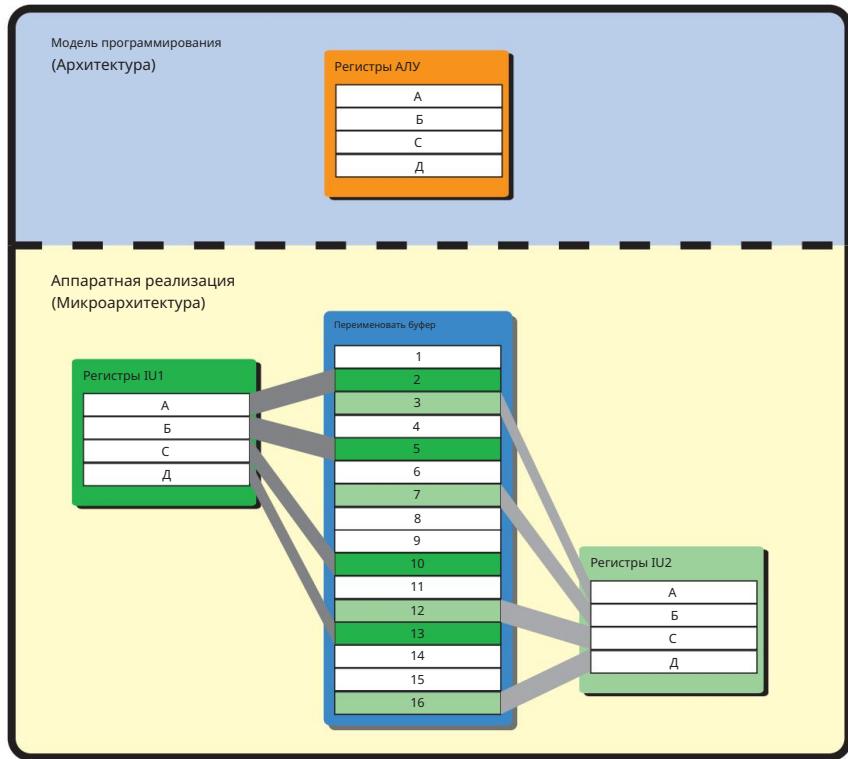


Рисунок 4-8: Переименование регистра

Давайте быстро рассмотрим конфликт ложного имени регистра в программе 4-2.

Номер строки	Код	Комментарии
1	добавить А, В, С	Сложить числа в регистрах А и В и сохранить результат в С.
2	добавить D, В, А	Сложить числа в регистрах В и D и сохранить результат в А.

Программа 4-2: Конфликт ложного имени регистра

В программе 4-2 нет зависимости от данных, и обе инструкции добавления могут выполняться одновременно, за исключением одной проблемы: первая операция добавления считывает содержимое А для ввода, а вторая операция добавления записывает новое значение в А в качестве вывода . . Следовательно, чтение первого добавления обязательно должно происходить до записи второго добавления . Переименование регистра решает этот конфликт имен регистров, позволяя второму дополнению записывать свои выходные данные во временный регистр; после того, как оба добавления выполняются параллельно, результат второго добавления записывается из этого временного регистра в архитектурный регистр А после того, как первое добавление завершило выполнение и записало свои собственные результаты.

Структурные опасности

Программа 4-3 содержит краткий пример кода, демонстрирующий суперскалярное выполнение в действии. Предполагая модель программирования, представленную для DLW-2, рассмотрим следующий фрагмент кода.

Номер строки	Код	Комментарии
15		добавить A, B в Сложить числа в регистрах A и B и сохранить результат в B.
16		добавить C, D Сложить числа в регистрах C и D и сохранить результат в D.

Программа 4-3: Структурная опасность

На первый взгляд кажется, что в программе 4-3 нет ничего плохого.

Нет никакой опасности для данных, потому что две инструкции не зависят друг от друга. Таким образом, должна быть возможность выполнять их параллельно. Однако в этом примере предполагается, что оба ALU используют одну и ту же группу из четырех регистров.

Но для того, чтобы регистровый файл DLW-2 мог вместить несколько ALU, обращающихся к нему одновременно, он должен отличаться от регистрового файла DLW-1 в одном важном отношении: он должен поддерживать две одновременные записи.

В противном случае параллельное выполнение двух инструкций программы 4-3 вызовет так называемую структурную опасность, когда у процессора недостаточно ресурсов для одновременного выполнения обеих инструкций.

Регистрационный файл

В суперскалярной схеме с несколькими ALU потребовалось бы огромное количество проводов для прямого подключения каждого регистра к каждому ALU. Эта проблема усугубляется по мере увеличения количества регистров и ALU. Следовательно, в суперскалярных схемах с большим количеством регистров регистры ЦП группируются в специальный модуль, называемый файлом регистров. Эта единица представляет собой массив памяти, очень похожий на массив ячеек, составляющих основную память компьютера, и доступ к нему осуществляется через специальный интерфейс, который позволяет ALU считывать или записывать в определенные регистры. Этот интерфейс состоит из шины данных и двух типов портов: портов чтения и портов записи. Чтобы прочитать значение из одного регистра в файле регистров, ALU обращается к порту чтения файла регистров и запрашивает, чтобы данные из определенного регистра были помещены на специальную внутреннюю шину данных, которую файл регистров разделяет с ALU. Аналогично, запись в регистровый файл выполняется через порт записи файла.

Один порт чтения позволяет ALU обращаться к одному регистру за раз, поэтому для того, чтобы ALU мог читать из двух регистров одновременно (как в случае инструкции добавления трех операндов), регистровый файл должен иметь два порта чтения. .

Точно так же порт записи позволяет ALU записывать только в один регистр за раз, поэтому одному ALU требуется один порт записи, чтобы иметь возможность записывать результаты операции обратно в регистр. Следовательно, регистровому файлу необходимо два порта чтения и один порт записи для каждого ALU. Таким образом, для суперскалярной схемы с двумя ALU регистровому файлу требуется всего четыре порта чтения и два порта записи.

Так получилось, что объем памяти, который занимает файл регистра, увеличивается примерно пропорционально квадрату числа портов, поэтому существует практический предел числа портов, которые может поддерживать данный файл регистра.

Это одна из причин, по которой современные процессоры используют отдельные файлы регистров для хранения целых чисел, чисел с плавающей запятой и векторных чисел. Поскольку каждый тип математики (целочисленный, с плавающей запятой и вектор) использует свой тип исполнительного модуля, присоединение нескольких целочисленных, с плавающей запятой и векторных исполнительных модулей к одному регистровому файлу приведет к созданию довольно большого файла.

Есть и другая причина использования нескольких регистрационных файлов для размещения различных типов исполнительных устройств. По мере увеличения размера регистрационного файла увеличивается и время, необходимое для доступа к нему. Вы, возможно, помните из «Пересмотренная и расширенная модель File-Clerk» на стр. 9, что мы предполагаем, что чтение и запись регистра происходят мгновенно. Если регистрационный файл становится слишком большим и задержка доступа к регистрационному файлу становится слишком большой, это может замедлить доступ к регистру до такой степени, что такой доступ займет заметное количество времени. Таким образом, вместо того, чтобы использовать один массивный регистрационный файл для каждого типа числовых данных, компьютерные архитекторы используют два или три регистрационных файла, связанных с несколькими различными типами исполнительных устройств.

Между прочим, если вы помните «Коды операций и машинный язык» на стр. 19, DLW-1 использовал серию двоичных чисел для обозначения того, к какому из четырех регистров обращается инструкция. Что ж, в случае чтения регистрационного файла эти числа передаются в интерфейс регистрационного файла, чтобы указать, какой из регистров должен поместить свои данные на шину данных. Взяв в качестве примера наши двухбитные обозначения регистров, порт в нашем файле с четырьмя регистрами будет иметь две строки, которые будут поддерживать либо высокое, либо низкое напряжение (в зависимости от того, был ли бит, помещенный в каждую строку, 1 или 0), и эти строки сообщат файлу, данные какого из его регистров должны быть помещены на шину данных.

[Опасности управления](#)

Опасности управления, также известные как опасности ветвления, — это опасности, которые возникают, когда процессор достигает условного перехода и должен решить, какую инструкцию выбрать следующей. В более примитивных процессорах конвейер останавливается, пока оценивается условие перехода и вычисляется цель перехода. Эта остановка вставляет в конвейер несколько циклов пузырьков, в зависимости от того, сколько времени требуется процессору для идентификации и определения местоположения целевой инструкции ветвления.

Современные процессоры используют технику, называемую предсказанием ветвлений, чтобы обойти эти связанные с ветвлением остановки. Мы обсудим предсказание ветвлений более подробно в следующей главе.

Другая потенциальная проблема, связанная с ветвями, заключается в том, что после того как условие ветвления оценено и адрес следующей инструкции загружен в программный счетчик, требуется несколько циклов, чтобы фактически выбрать следующую инструкцию из памяти. Эта задержка загрузки инструкции добавляется к задержке оценки условия ветвления, обсуждавшейся ранее в этом разделе. В зависимости от того, где находится следующая инструкция — например, в соседнем кэше, в оперативной памяти или на жестком диске — ее получение может занять от нескольких до тысяч циклов. Циклы, которые процессор тратит на ожидание появления этой инструкции, являются мертвыми, потраченными впустую циклами, которые проявляются в виде пузырей в конвейере процессора и убивают производительность.

Компьютерные архитекторы используют кэширование инструкций, чтобы уменьшить последствия задержки загрузки, и мы поговорим об этом методе подробнее в следующей главе.

5

Intel Pentium и ПЕНТИУМ ПРО

Теперь, когда вы ознакомились с основами микропроцессорной архитектуры, давайте взглянем на какое-нибудь реальное аппаратное обеспечение, чтобы увидеть, как производители реализуют две основные концепции, рассмотренные в двух предыдущих главах, — конвейерную обработку и суперскалярное выполнение — и представим совершенно новую концепцию: инструкции окна. Во-первых, мы завершим обсуждение основ микропроцессоров рассмотрением Pentium. Затем мы изучим в Подробно опишите микроархитектуру P6, лежащую в основе процессоров Pentium Pro, Pentium II и Pentium III. Микроархитектура P6 представляет собой фундаментальный отход от конструкций микропроцессоров, которые мы изучали до сих пор, и понимание того, как она работает, даст вам четкое представление о наиболее важных концепциях современной микропроцессорной архитектуры.

Оригинальный пентиум

Оригинальный Pentium по сегодняшним меркам имел чрезвычайно скромный дизайн. Бюджеты транзисторов были меньше, когда чип был представлен в 1993 году, поэтому Pentium не упаковывает в свой кристалл столько аппаратного обеспечения, сколько современный микропроцессор. Таблица 5-1 суммирует его функции.

Табл. 5-1: Обзор функций Pentium

Дата введения	22 марта 1993 г.
Производственный процесс	0,8 мкм
Количество транзисторов	3,1 миллиона
Тактовая частота при введении	60 и 66 МГц
Размеры кэша	L1: инструкция 8 КБ, данные 8 КБ
Расширения x86 ISA	MMX добавлен в 1997 г.

Взглянув на схему Pentium (см. рис. 5.1), вы увидите, что он имеет два целочисленных ALU и ALU с плавающей запятой, а также некоторые другие устройства, которые я опишу позже. Pentium также имеет кэш-память первого уровня — компонент микропроцессора, который вы еще не видели. Прежде чем двигаться дальше, давайте более подробно рассмотрим этот новый компонент, который действует как область хранения кода и данных для процессора.

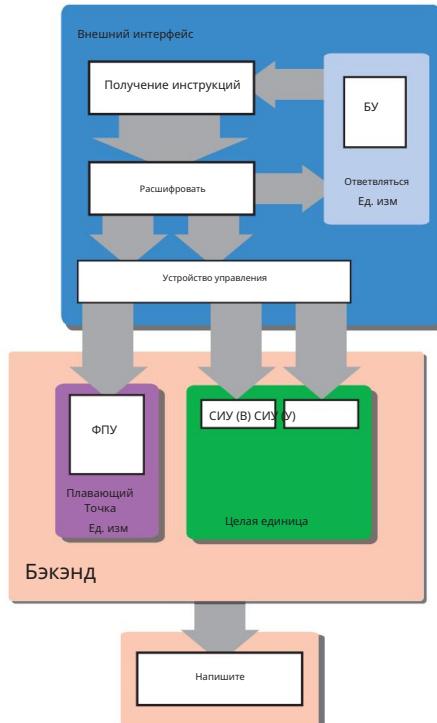


Рисунок 5-1: Базовая микроархитектура оригинального процессора Intel Pentium

Тайники

До сих пор я говорил о коде и данных так, как если бы все они хранились в основной памяти. Хотя это может быть правдой в ограниченном смысле, это не говорит всей истории. Хотя за последние два десятилетия скорости процессоров резко возросли, скорость основной памяти не поспевает за ними. В каждой компьютерной системе, представленной в настоящее время на рынке, существует огромный разрыв в скорости между процессором и основной памятью. Для передачи кода и данных между основной памятью, регистрами и исполнительными блоками требуется такое огромное количество тактовых циклов процессора, что, если бы не было решения для устранения этого узкого места, это убило бы большую часть прироста производительности, вызванного увеличением процессора. тактовые частоты.

Действительно доступна очень быстрая память, которая могла бы сократить отставание в скорости, но она слишком дорога для широкого использования в основной памяти ПК. На самом деле, как правило, чем быстрее технология памяти, тем больше она стоит на единицу хранения. В результате разработчики компьютерных систем заполняют разрыв в скорости, размещая меньшее количество более быстрой и дорогой памяти, называемой кэш-памятью, между основной памятью и регистрами. Эти кэши, изображенные на рис. 5-2, содержат фрагменты часто используемого кода и данных, поэтому они находятся в пределах легкой досягаемости от внешнего интерфейса процессора.

В большинстве систем существует несколько уровней кэша между основной памятью и регистрами. Кэш уровня 1 (называемый кэшем L1 или просто L1 для краткости) — это самый маленький и самый дорогой бит кэша, поэтому он расположен ближе всего к серверной части процессора. Большинство систем ПК имеют другой уровень кеша, называемый кешем уровня 2 (кэш L2 или просто L2), расположенный между кешем L1 и основной памятью, а некоторые системы даже имеют третий уровень кеша, кэш L3, расположенный между кешем L2 и основной памятью. Память. На самом деле, как показано на рис. 5.2, основная память сама по себе является просто кэшем для жесткого диска.

Когда процессору требуется конкретный фрагмент кода или данных, он сначала проверяет кэш L1, чтобы увидеть, присутствует ли нужный элемент. Если это так (ситуация, называемая попаданием в кэш), он перемещает этот элемент непосредственно либо на стадию выборки (в случае кода), либо в регистровый файл (в случае данных). Если элемент отсутствует — промах кэша — процессор проверяет более медленный, но больший кеш L2. Если элемент присутствует в L2, он копируется в L1 и передается во внешний или задний конец. Если в кэш-памяти L2 происходит промах, процессор проверяет L3 и так далее, пока не произойдет попадание в кэш-память или пока кэш-промах не распространится на всю основную память.

Один из популярных способов размещения кэша L1 — хранить код и данные в отдельных половинах кэша. Половину кеша кода часто называют кешем инструкций или I-кэшем, а половину кеша данных называют кешем данных или D-кэшем. Такой дизайн разделенного кэша имеет определенные преимущества в производительности и используется во всех процессорах, обсуждаемых в этой книге.

ПРИМЕЧАНИЕ. Дизайн разделенного кэша L1 часто называют архитектурой Гарварда в честь Harvard Mark I. Mark I был компьютером на основе ретрансляции, разработанным IBM и поставленным в Гарвард в 1944 г., и это была первая машина, в которой реализована концептуальная явно разделить код и данные на его архитектуру.

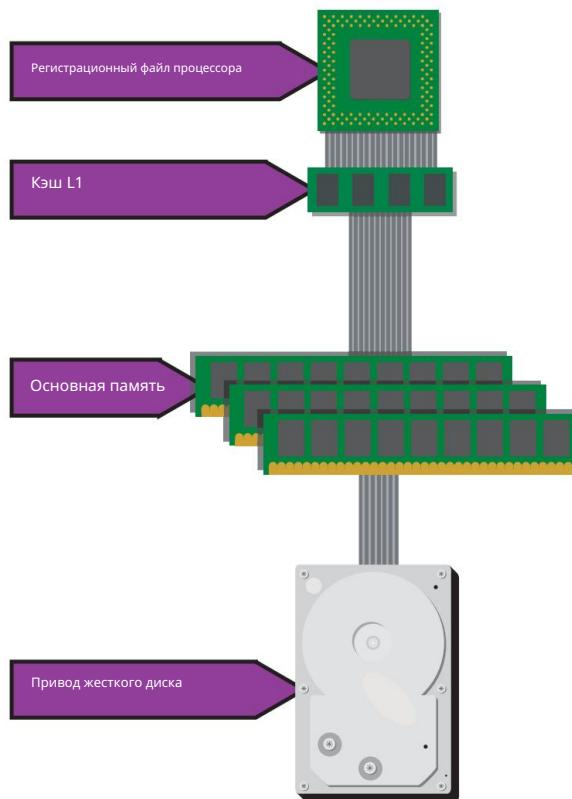


Рисунок 5-2: Иерархия памяти компьютерной системы, от самой маленькой, быстрой и самой дорогой памяти (регистровый файл) до самой большой, самой медленной и самой дешевой (жесткий диск)

Раньше, когда бюджеты транзисторов были намного меньше, чем сегодня, все кэш-памяти располагались где-то на системной шине компьютера между процессором и основной памятью. Однако сегодня кэши L1 и L2 обычно интегрируются в сам кристалл ЦП вместе с остальной схемой ЦП.

Кэш-память на кристалле имеет значительные преимущества в производительности по сравнению с кэшем вне кристалла и необходима для того, чтобы современные высококонвейерные суперскалярные машины были заполнены кодом и данными.

Конвейер Pentium

Как вы, наверное, уже догадались, суперскалярный процессор не имеет только одного конвейера. Поскольку стадия выполнения разделена между несколькими исполнительными модулями, работающими параллельно, можно сказать, что такой процессор, как Pentium, имеет несколько конвейеров — по одному на каждый исполнительный модуль. Рисунок 5-3 иллюстрирует несколько конвейеров Pentium.

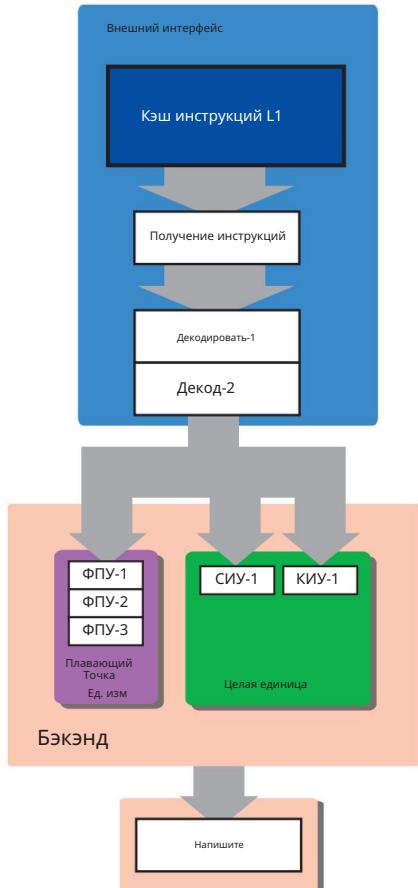


Рисунок 5-3: Конвейеры Pentium

Как видите, каждый из конвейеров Pentium разделяет четыре этапа в общий:

- z Получить
- z Декодировать-1
- z Декодировать-2
- z Написать

Когда инструкция достигает фазы выполнения своего жизненного цикла, она поступает в более специализированный конвейер, специфичный для исполнительного устройства.

Различные исполнительные блоки процессора могут иметь различную глубину конвейера, при этом целочисленный конвейер обычно является самым коротким, а конвейер с плавающей запятой обычно является самым длинным. На рис. 5.3 вы можете видеть, что два целочисленных ALU Pentium имеют одноэтапные конвейеры, а устройство с плавающей запятой имеет трехэтапный конвейер.

Поскольку целочисленный конвейер является самым коротким, его обычно принимают за конвейер по умолчанию при обсуждении микроархитектуры процессора. Поэтому, когда вы видите ссылку в этом тексте или в другом тексте на конвейер суперскалярного процессора или базовый конвейер, вы должны предположить, что он относится к целочисленному конвейеру процессора.

Базовый целочисленный конвейер Pentium состоит из пяти этапов, причем этапы разбито следующим образом:

1. Инструкции Prefetch/Fetch извлекаются из кэша инструкций.
и выровнены в буферах предварительной выборки для декодирования.
2. Инструкции Decode-1 декодируются во внутренний формат инструкций Pentium с использованием быстрого набора аппаратных правил. Прогнозирование переходов также происходит на этом этапе.
3. Декодирование-2 Здесь декодируются инструкции, для которых требуется ПЗУ с микрокодом. Кроме того, на этом этапе происходят адресные вычисления.
4. Выполнение Целочисленное аппаратное АЛУ выполняет команду.
5. Обратная запись . Результаты вычислений записываются обратно в регистрационный файл.

Эти этапы должны быть вам знакомы, хотя первые три этапа немного отличаются от описанных выше простых четырехэтапных конвейеров. Давайте совершим небольшое путешествие по этапам конвейера Pentium, чтобы вы могли изучить каждый из них более подробно.

Стадия предварительной выборки/выборки соответствует фазе выборки стандартного жизненного цикла инструкции. В отличие от простых двухбайтовых инструкций одинакового размера в нашем примере архитектуры DLW, инструкции x86 могут иметь размер от 1 до 17 байтов, хотя средняя длина инструкции составляет немногим менее 3 байтов. Широко варьируемая длина инструкций x86 усложняет стадию выборки Pentium, поскольку инструкции нельзя просто извлечь, а затем передать непосредственно на стадию декодирования. Вместо этого инструкции x86 сначала загружаются в буфер, где определяются и отмечаются границы каждой инструкции. Затем из этого буфера помеченные инструкции выравниваются и отправляются на аппаратное декодирование Pentium.

Фаза декодирования процесса исполнения Pentium разделена на две стадии конвейера, первая из которых наиболее точно соответствует этапу конвейера декодирования, с которым вы уже знакомы. Этап декодирования-1 берет только что полученную команду и декодирует ее во внутренний формат инструкции Pentium, чтобы ее можно было использовать для указания исполнительным блокам процессора, как манипулировать потоком данных. Этап декодирования-1 также включает в себя блок ветвления Pentium, который проверяет текущую инструкцию декодирования, чтобы определить, является ли она ветвью, и если это ветвь, чтобы определить ее тип. Именно в этот момент происходит предсказание ветвления , но мы рассмотрим предсказание ветвления более подробно чуть позже.

Основное различие между пятиэтапным конвейером Pentium и четырехэтапным конвейером, рассмотренным в главе 3, заключается во втором этапе декодирования. ISA RISC, как и примитивная DLW ISA из глав 1 и 2, поддерживают только несколько простых режимов адресации загрузки-сохранения. Напротив, x86 ISA поддерживает

несколько сложных режимов адресации, которые изначально были разработаны для облегчения жизни программистов на ассемблере, но в конечном итоге усложнили жизнь всем. Эти режимы адресации требуют дополнительных вычислений адресов, и эти вычисления передаются на этап декодирования-2, где их обрабатывает специальное аппаратное обеспечение для вычисления адресов перед отправкой инструкции исполнительным устройствам.

На этапе декодирования-2 также вступает в действие ПЗУ микрокода Pentium. Pentium декодирует многие инструкции x86 непосредственно в своем аппаратном декодере, но более длинные инструкции декодируются с помощью ПЗУ микрокода, как описано в разделе «Краткая история ISA» на странице 71.

Как только инструкции декодированы, блок управления Pentium определяет, когда они могут быть отправлены на серверную часть и к какому исполнительному блоку. Таким образом, работа блока управления состоит в том, чтобы координировать перемещение инструкций от внешнего интерфейса процессора к его внутреннему, чтобы они могли войти в фазы выполнения и обратной записи процесса выполнения.

Последние два этапа конвейера — выполнение и обратная запись — должны быть вам уже знакомы. В следующем крупном разделе будут описаны исполнительные устройства Pentium, так что думайте об этом как о более подробном обсуждении стадии выполнения.

[Модуль ветвления и предсказание ветвления](#)

Прежде чем мы более подробно рассмотрим заднюю часть Pentium, давайте более подробно рассмотрим один аспект передней части Pentium: ответвительный блок (BU).

В правой части внешнего интерфейса Pentium, показанного ранее на рис. 5-1, обратите внимание на блок ответвления, присоединенный к этапам конвейера выборки инструкций и декодирования/отправки. Я изобразил модуль ответвления как часть внешнего интерфейса машины, хотя технически он по-прежнему считается модулем доступа к памяти . потому что BU тесно взаимодействует со сборщиком команд, направляя его с помощью счетчика программ на разные участки кодового потока.

Блок ответвления содержит исполнительный блок ответвления (BEU) и ответвление блок предсказания (BPU), и всякий раз, когда декодер переднего плана встречает инструкцию условного перехода, он отправляет ее в BU для выполнения. BU, в свою очередь, обычно должен отправить его одному из других исполнительных блоков для оценки условия перехода инструкции, чтобы BU мог определить, выполнено ли ветвление или нет . Как только BU определяет, что ветвь была принята, он должен получить начальный адрес следующего блока кода, который должен быть выполнен. Этот адрес, цель ветвления, должен быть рассчитан, и передний конец должен быть указан, чтобы начать выборку кода по новому адресу.

В старых процессорах весь процессор просто простоявал и ждал условие перехода, которое должно быть оценено, ожидание, которое может быть довольно долгим, если оценка включает какой-то сложный расчет. Современные процессоры используют метод, называемый спекулятивным выполнением, который включает в себя обоснованное предположение, в каком направлении в конечном итоге пойдет ветвь, а затем начало выполнения в новой цели ветвления до фактической оценки состояния ветвления. Это обоснованное предположение сделано с использованием одного из множества методов прогнозирования ветвлений, о котором я расскажу подробнее чуть позже. Спекулятивное выполнение используется для того, чтобы задержки, связанные с оценкой ветвей, не создавали пузыри в конвейере.

Инструкции, которые спекулятивно выполняются, не могут записать свои результаты обратно в регистровый файл, пока не будет оценено условие ветвления. Если ВРУ правильно предсказал переход, эти спекулятивные инструкции могут быть помечены как неспекулятивные, и их результаты будут записаны обратно, как и обычные инструкции.

Предсказание переходов может иметь неприятные последствия, если процессор неправильно предсказывает переход. Такие неправильные предсказания плохи, потому что, если все те инструкции, которые процессор загрузил в конвейер и начал спекулятивное выполнение, окажутся из неправильной цели перехода, конвейер должен быть очищен от ошибочных, спекулятивных инструкций и их сопутствующих результатов. После сброса этого конвейера внешний интерфейс должен затем получить правильный целевой адрес ветви, чтобы процессор мог начать выполнение в нужном месте в потоке кода.

Как вы узнали из главы 3, очистка конвейера инструкций и последующее его повторное заполнение наносят огромный удар по средней скорости выполнения процессора и общей производительности. Кроме того, существует задержка (и, следовательно, несколько циклов конвейерных пузырей), связанная с вычислением правильной цели перехода и загрузкой нового потока команд во внешний интерфейс. Эта задержка может значительно снизить производительность, особенно в коде с интенсивным переходом. Поэтому крайне важно, чтобы аппаратное обеспечение предсказания переходов процессора было как можно более точным.

Существует два основных типа предсказания переходов: статическое предсказание и динамическое предсказание. Статическое предсказание ветвления простое и основано на предположении, что большинство ветвлений, указывающих назад, происходят в контексте повторяющихся циклов, где инструкция ветвления используется для определения того, следует ли повторять цикл снова. В большинстве случаев условие выхода из цикла будет ложным, что означает, что ветвь цикла будет оценена как выполненная, тем самым предписывая машине повторить код цикла еще раз. В этом случае статическое прогнозирование ветвлений просто предполагает, что все обратные ветвления взяты. Для ветвления, указывающего на блок кода, который появляется позже в программе, статический предиктор предполагает, что ветвь не используется.

Статическое предсказание выполняется очень быстро, потому что оно не требует поиска в таблице или вычислений, но вероятность его успеха сильно зависит от набора инструкций программы. Если программа полна циклов, статическое прогнозирование работает достаточно хорошо; если он не заполнен циклами, статическое прогнозирование ветвлений работает довольно плохо.

Чтобы обойти проблемы, связанные со статическим предсказанием, компьютер архитекторы используют различные алгоритмы для прогнозирования ветвлений на основе прошлого поведения программы. Эти алгоритмы динамического прогнозирования ветвлений обычно включают использование двух типов таблиц — таблицы истории ветвлений (ВНТ) и целевой буфер ветвления (ВТВ) — для записи информации о результатах уже выполненных ветвей. ВНТ создает запись для каждой условной ветви, с которой ВУ столкнулся в своих последних нескольких циклах. Эта запись включает в себя несколько битов, которые указывают вероятность того, что ветвь будет выбрана на основе ее прошлой истории. Когда внешний интерфейс встречает инструкцию ветвления, которая имеет запись в своем ВНТ, предиктор ветвлений использует эту информацию истории ветвлений, чтобы решить, выполнять ли ветвь спекулятивно.

Если предсказатель ветвления решит выполнить ветвь спекулятивно, ему нужно точно знать, куда в памяти указывает ветвь, другими словами, ему нужна цель ветвления. BTB хранит цели ранее выполненных ветвей, поэтому, когда ветвь выполняется, BPU получает предполагаемую цель ветвления от BTB и сообщает внешнему интерфейсу начать выборку инструкций с этого адреса. Надеюсь, BTB содержит запись для ветви, которую вы пытаетесь выполнить, и надеюсь, что эта запись верна. Если цель ветви либо отсутствует, либо неверна, у вас проблема. Я не буду вдаваться в вопросы, связанные с производительностью BTB, но достаточно сказать, что больший BTB обычно лучше, потому что он может хранить больше целевых ветвей и, таким образом, снижает вероятность промаха BTB.

Pentium использует как статические, так и динамические методы прогнозирования переходов, для предотвращения ошибочных прогнозов и задержек ветвления. Если инструкция ветвления не имеет записи в BHT, Pentium использует статическое предсказание, чтобы решить, какой путь выбрать. Если в инструкции есть запись BHT, используется динамическое предсказание. BHT Pentium содержит 256 записей, а это означает, что у него недостаточно места для хранения информации о большинстве ветвей средней программы. Тем не менее, BHT позволяет Pentium предсказывать ответвления с гораздо большей вероятностью успеха, от 75 до 85 процентов, по данным Intel, чем если бы он использовал только статическое предсказание. Pentium также использует BTB для хранения предсказанных целей ветвления. В большей части литературы и диаграмм Intel BTB и BHT объединены под названием front-end BTB и считаются единой структурой.

Бэкэнд Pentium

Суперскалярная серверная часть Pentium довольно проста. Он имеет два пятиступенчатых целочисленных конвейера, которые Intel обозначила как U и V, и один шестиступенчатый конвейер для операций с плавающей запятой. В этом разделе более подробно рассматривается каждое из этих ALU.

Целочисленные ALU

Целочисленные конвейеры Pentium U и V не полностью симметричны. U, как канал по умолчанию, немного более функционален и содержит шифтер, которого нет в V. По этой причине на рис. 5-1 я обозначил простую целочисленную единицу (SIU) для канала U. и V-образный комплексный целочисленный блок (CIU). Большинство схем, которые мы будем изучать в этой книге, имеют асимметричные целочисленные блоки, где один целочисленный блок более сложный и способен обрабатывать больше типов инструкций, чем другой, более простой блок.

Целочисленные единицы Pentium не являются полностью независимыми. Существует ряд ограничений, которые я не буду подробно описывать, налагающих ограничения на то, какие комбинации целочисленных инструкций могут выполняться параллельно. В целом, однако, два целочисленных блока Pentium изначально обеспечивали достаточную производительность для целочисленных вычислений, чтобы быть конкурентоспособными в свое время, особенно для офисных приложений с интенсивным использованием целочисленных вычислений.

Последнее, что стоит отметить в связи с двумя целочисленными ALU Pentium, это то, что они отвечают за многие вычисления адресов процессора.

Процессоры, разработанные недавно, имеют специализированное оборудование для обработки вычислений адресов, связанных с загрузкой и сохранением, но в Pentium эти вычисления выполняются в целочисленных ALU.

[ALU с плавающей запятой](#)

Операции с плавающей запятой обычно сложнее реализовать, чем операции с целыми числами, поэтому конвейеры с плавающей запятой часто имеют больше этапов, чем целочисленные конвейеры. Шестиступенчатый конвейер Pentium для операций с плавающей запятой не является исключением из этого правила. Производительность процессора Pentium с плавающей запятой ограничена двумя основными факторами. Во-первых, процессор может одновременно выполнять операции с плавающей запятой и с целыми числами только в крайне ограничительных условиях. Однако это не так уж плохо, потому что код с плавающей запятой и целочисленный код редко смешиваются. Второй фактор, неудачный дизайн архитектуры x87 с плавающей запятой, более важен.

В отличие от обычного файла регистров с плавающей запятой среднего RISC ISA, файл регистров x87 содержит восемь 80-битных регистров, расположенных в виде стека.

Стек — это простая структура хранения данных, обычно используемая программистами и некоторыми научными калькуляторами для выполнения арифметических операций.

ПРИМЕЧАНИЕ Плоский — это прилагательное, которое программисты используют для описания массива элементов, который логически расположен так, что любой элемент доступен через простой адрес. Например, все файлы регистров, которые мы видели до сих пор, являются плоскими, потому что программисту нужно знать только имя регистра, чтобы получить доступ к этому регистру. Сравните плоский файл со структурой стека, описанной далее, в которой элементы, находящиеся внутри структуры данных, не доступны программисту сразу и напрямую.

Как показано на рис. 5.4, программист записывает данные в стек, помещая на вершину стека с помощью команды `push`. Таким образом, стек растет с каждым новым фрагментом данных, помещаемым на его вершину. Чтобы прочитать данные из стека, программист выдает команду `pop`, которая возвращает самый верхний фрагмент данных и удаляет эти данные из стека, вызывая сжатие стека.

По мере того, как стек увеличивается и уменьшается, переменная `ST`, обозначающая стек, в `top` всегда указывает на верхний элемент стека. В самом базовом типе стека `ST` является единственным элементом стека, к которому программист может получить прямой доступ: он читается с помощью команды `pop` и записывается с помощью команды `push`. В этом случае, если вы хотите прочитать синий элемент из стека на рис. 5.4, вы должны извлечь все элементы над ним, а затем вы должны извлечь сам синий элемент. Точно так же, если вы хотите изменить синий элемент, вам сначала нужно вытолкнуть все элементы над ним. Затем вы извлекаете сам синий элемент, изменяете его, а затем помещаете измененный элемент обратно в стек.

Поскольку первый элемент, который вы помещаете в стек, недоступен до тех пор, пока вы не удалите все элементы над ним, стек часто называют структурой данных `FIFO` (*first in, last out*). Сравните это с традиционной структурой очереди, такой как очередь в кассу супермаркета, которая представляет собой структуру `FIFO` (первым пришел, первым вышел).

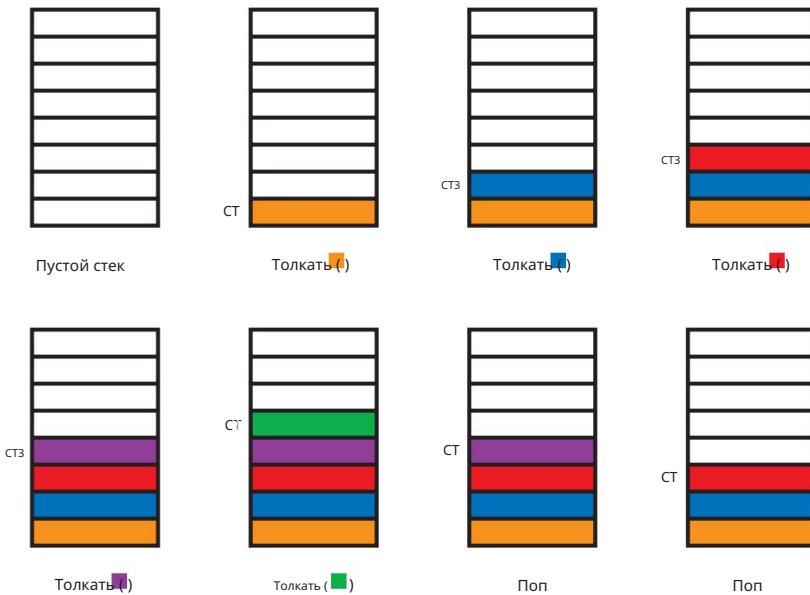


Рисунок 5-4: Отправка и извлечение данных в простой стек

Все эти нажатия и выталкивания звучат как большая работа, и вы можете удивиться, зачем кому-то использовать такую структуру данных. Как оказалось, стек идеально подходит для некоторых специализированных типов приложений, таких как синтаксический анализ естественного языка, отслеживание вызовов вложенных процедур и вычисление постфиксных арифметических выражений. Это была утилита стека для оценки постфиксных арифметических выражений, которая рекомендовала ее разработчикам модуля x87 с плавающей запятой (FPU), поэтому они организовали восемь регистров с плавающей запятой FPU в виде стека.

ПРИМЕЧАНИЕ Обычные арифметические выражения, такие как $5 + 2 - 1 = 6$, называются инфиксными выражениями, поскольку арифметические операторы (+ и -) расположены между числами, над которыми они работают. Постфиксные выражения, напротив, имеют операторы, присоединенные к концу выражения, например, $521-+ = 6$. Вы можете вычислить это выражение слева направо, используя стек, помещая в стек числа 5, 2 и 1 (в указанном порядке), а затем извлекать их обратно (сначала 1, затем 2 и, наконец, 5), когда встречаются операторы в конце выражения. Операторы будут применяться к извлекаемым числам по мере их появления, а текущий результат будет храниться в верхней части стека.

Регистровый файл x87 немного отличается от простого стека, описанного два абзаца назад, потому что ST — не единственная переменная, через которую можно получить доступ к элементам стека. Вместо этого программист может читать и записывать нижние элементы стека, используя ST со значением индекса, обозначающим положение желаемого элемента относительно вершины стека.

Например, на рис. 5.5 стек достигает максимальной высоты, когда в него только что помещено зеленое значение. Доступ к этому зеленому значению осуществляется через переменную ST(0), поскольку оно занимает вершину стека. Доступ к синему значению, поскольку оно находится на три элемента ниже вершины стека, осуществляется через ST(3).

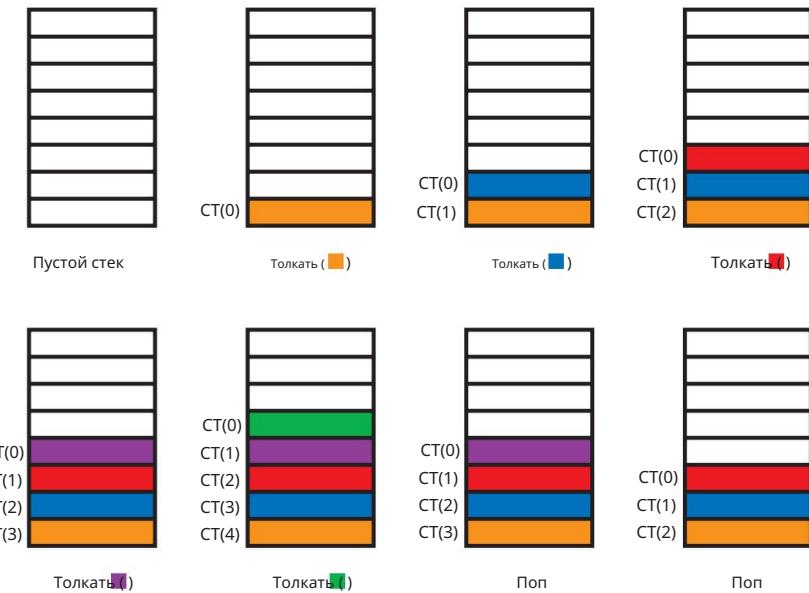


Рисунок 5-5: Загрузка и извлечение данных из стека регистров x87 с плавающей запятой

В общем, для чтения или записи в определенный регистр в стеке вы можете просто использовать форму ST(i), где i — количество регистров с вершиной стека.

Сторонники чистоты программирования могут предположить, что, поскольку вы можете произвольно обращаться к элементам стека, бессмысленно по-прежнему называть регистровый файл x87 стеком. Это было бы так, за исключением одного утова: для каждой арифметической инструкции с плавающей запятой по крайней мере один из операндов должен находиться на вершине стека. Например, если вы хотите добавить два числа с плавающей запятой, одно из чисел должно быть в верхней части стека, а другое может быть в любом из других регистров. Например, инструкция

причуда CT, CT(5)

выполняет операцию

$CT = CT + CT(5)$

Хотя основанный на стеке характер регистра файла x87 с плавающей запятой был первоначально это было благом для программистов на ассемблере, но вскоре оно стало препятствием для производительности операций с плавающей запятой, поскольку компиляторы получили более широкое распространение. Компилятору проще управлять плоским файлом регистров, а в более новых RISC ISA использовались не только большие плоские файлы регистров, но и инструкции с тремя операндами с плавающей запятой.

Хотя трюков компилятора, возможно, достаточно, чтобы компенсировать ограничение x87 на два операнда в большинстве случаев, они не вполне способны преодолеть ограничение как на два операнда, так и на стек. Таким образом, одни только трюки с компилятором не устранит потери производительности, связанные с обоими этими факторами.

странные вместе взятые. Файл регистра на основе стека настолько плох, что требуется микроархитектурный взлом, чтобы имитировать плоский файл регистра и, таким образом, не допустить снижения производительности вычислений с плавающей запятой при разработке архитектуры x87.

Этот микроархитектурный хак включает в себя турбонадув одной инструкции: FXCH. Инструкция fxch — это обычная инструкция x87, позволяющая поменять местами любой элемент стека с вершиной стека. Например, если вы хотите вычислить $ST(2) = ST(2) + ST(6)$, вы можете выполнить код, показанный в программе 5-1:

Номер строки	Код	Комментарии
1	FXCH ST(2)	Поместите содержимое ST(2) в ST , а содержимое ST в ST(2).
2	fadd ST, ST(6)	Добавить содержимое ST в ST(6).
3	FXCH ST(2)	Поместите содержимое ST(2) в ST , а содержимое ST в ST(2).

Программа 5-1: Использование инструкции fxch

А теперь в дело вступает микроархитектурный хак. На всех современных процессорах x86, от оригинального Pentium до Pentium 4, но не включая Pentium 4, инструкция fxch может выполняться за нулевые циклы. Это означает, что во всех смыслах и целях fxch является «бесплатным» и поэтому может использоваться при необходимости без снижения производительности. (Обратите внимание, однако, что инструкция fxch по-прежнему использует пропускную способность декодирования, поэтому даже когда она «бесплатна», она не совсем «бесплатна».) Если вы остановитесь и подумаете о том, что перед выполнением любой инструкции с плавающей запятой (которая чтобы задействовать вершину стека), вы можете мгновенно поменять местами ST с любым другим регистром, вы поймете, что инструкция fxch с нулевым циклом дает программистам функциональный эквивалент файла плоского регистра.

Возвращаясь к предыдущему примеру, тот факт, что первая инструкция в Программа 5-1 выполняется «мгновенно», так сказать, означает, что последовательность операций фактически выглядит следующим образом:

причуда CT(2), CT(6)

На самом деле существуют некоторые ограничения на использование «бесплатной» инструкции fxch , но общий результат заключается в том, что, используя этот трюк, и Pentium, и его преемники получают эффективные преимущества файла с плоским регистром, но с вышеупомянутым наложением, чтобы декодировать пропускную способность.

Накладные расходы x86 на Pentium

Есть ряд мест, таких как этап декодирования-2 Pentium, где устаревшая поддержка x86 значительно увеличивает нагрузку на структуру Pentium. По оценкам Intel, колоссальные 30 процентов транзисторов Pentium предназначены исключительно для поддержки устаревшей архитектуры x86. Если принять во внимание тот факт, что RISC-конкуренты Pentium с сопоставимым количеством транзисторов могли потратить эти транзисторы на повышающую производительность аппаратное обеспечение, такое как исполнительные блоки и кэш-память, неудивительно, что Pentium отставал от некоторых своих современников, когда он был впервые представлен.

Большая часть транзисторов Pentium, поддерживающих наследие, съедена его ПЗУ с микрокодом. В главе 4 объяснялось, что одно из больших преимуществ процессоров RISC заключается в том, что им не нужны ПЗУ микрокода, которые требуются конструкциям CISC для декодирования больших и сложных инструкций. (Подробнее о x86 в качестве CISC ISA см. в разделе «CISC, RISC и трансляция набора инструкций» на стр. 103.)

Внешний интерфейс Pentium также страдает от раздувания, связанного с x86, поскольку его логика предварительной выборки должна учитывать тот факт, что инструкции x86 не имеют одинакового размера и, следовательно, могут охватывать строки кэша. Логика декодирования Pentium также должна поддерживать модель сегментированной памяти x86, что означает проверку и соблюдение ограничений на сегменты кода; такая проверка требует собственного специального оборудования для вычисления адресов в дополнение к другому оборудованию для адресов Pentium.

[Резюме: Pentium в историческом контексте](#)

Основным фактором, сдерживающим производительность Pentium по сравнению с его RISC-конкурентами, был тот факт, что весь его внешний интерфейс был раздут аппаратным обеспечением, предназначенным исключительно для поддержки функций x86, которые даже во время появления процессора быстро выходили из употребления. При таком ограниченном бюджете транзисторов, каким он был в 1993 году, каждый из этих дополнительных сумматоров адресов и буферов предварительной выборки, не говоря уже о ПЗУ микрокода, представлял собой болезненную трату скучных ресурсов, которые никак не улучшали производительность Pentium.

К счастью для Intel, проблемы с устаревшей поддержкой Pentium не закончились. Было несколько фактов и тенденций, которые работали в пользу Intel и x86 ISA. Если мы на мгновение забудем о расширениях ISA, таких как MMX, SSE и т. д., и о нечетной горстке инструкций специального назначения, таких как инструкция идентификатора процессора Intel, которые время от времени добавляются к x86 ISA, базовая устаревшая x86 ISA исправлена . в размерах и не увеличивается с годами.

Точно так же, за одним исключением (P6, о котором пойдет речь ниже), количество оборудования, необходимого для поддержки таких инструкций, также не имеет тенденции к увеличению.

Транзисторы, с другой стороны, быстро сократились с момента появления Pentium. Если сложить эти два факта вместе, это означает, что относительная стоимость (в транзисторах) поддержки x86, стоимость, которая в основном сосредоточена на внешнем интерфейсе процессора x86, снизилась по мере увеличения количества транзисторов процессора.

На поддержку x86 приходится менее 10 процентов транзисторов Pentium 4, и этот процент еще меньше для самых последних процессоров Intel. Это устойчивое и резкое снижение относительной стоимости устаревшей поддержки в значительной степени способствовало тому, что аппаратное обеспечение x86 смогло догнать и даже превзойти своих конкурентов RISC по производительности как для целых чисел, так и для вычислений с плавающей запятой. Другими словами, Кривые Мура были чрезвычайно добры к x86 ISA.

Несмотря на высокую цену, которую пришлось заплатить за поддержку x86, Pentium имел коммерческий успех и способствовал доминированию Intel на рынке x86, изобретенном самой компанией. Но для того, чтобы Intel подняла производительность x86 на новый уровень, ей нужно было предпринять ряд радикальных шагов, выпустив Pentium Pro вслед за Pentium.

ПРИМЕЧАНИЕ . Здесь и далее в этой книге я использую термин «кривые Мура» вместо более популярная фраза Закон Мура. Подробное объяснение явления, на которое ссылаются оба этих термина, см. в моей статье на Ars Technica под названием «Понимание закона Мура» (<http://arstechnica.com/paedia/m/moore/moore-1.html>).

Микроархитектура Intel P6: Pentium Pro

Микроархитектура P6 от Intel, впервые реализованная в Pentium Pro, по любым разумным меркам имела оглушительный успех. Его производительность была значительно выше, чем у Pentium, и рынок щедро вознаградил Intel за это. Микроархитектура также оказалась чрезвычайно масштабируемой, обеспечив Intel добрых полдесятилетия господства настольных компьютеров и проложив путь системам x86 для конкуренции с RISC на рынках рабочих станций и серверов.

Таблица 5-2 суммирует эволюцию функций микроархитектуры P6.

Таблица 5-2: Эволюция P6

	Pentium Pro Vitals	Пентиум II Виталс	Пентиум III Виталс
Дата введения	1 ноября 1995 г. 0,60/0,35	7 мая 1997 г.	26 февраля 1999 г.
Процесс	МКМ 5,5 млн.	0,35 мкм	0,25 мкм
Количество транзисторов		7,5 миллионов	9,5 миллионов
Тактовая частота при вводе 150, 166, 180 и 200 МГц	233, 266 и 300 МГц	450 и 500 МГц	
Размер кэша L1	8 КБ инструкции, 8 КБ данных	инструкция 16КБ, 16 КБ данных	инструкция 16КБ, 16 КБ данных
Размер кэша L2	256 КБ или 512 КБ (на кристалле)	512 КБ (вне кристалла)	512 КБ (на кристалле)
Расширения x86 ISA		MMX	SSE добавлен в 1999 г.

В чем был секрет P6 и как он обеспечил такой качественный скачок в производительности? Ответ сложен и включает в себя вклад многочисленных технологий и методов, наиболее важные из которых уже были представлены в мире x86 более мелкими конкурентами Intel x86 (в первую очередь, AMD K5): разделение функций выборки и декодирования внешнего интерфейса из исполнительной функции серверной части с помощью окна инструкций.

Рисунок 5-6 иллюстрирует базовую микроархитектуру P6. Как видите, у этой микроархитектуры есть немало выдающихся особенностей, которые коренным образом отличают ее от схем, которые мы изучали до сих пор.

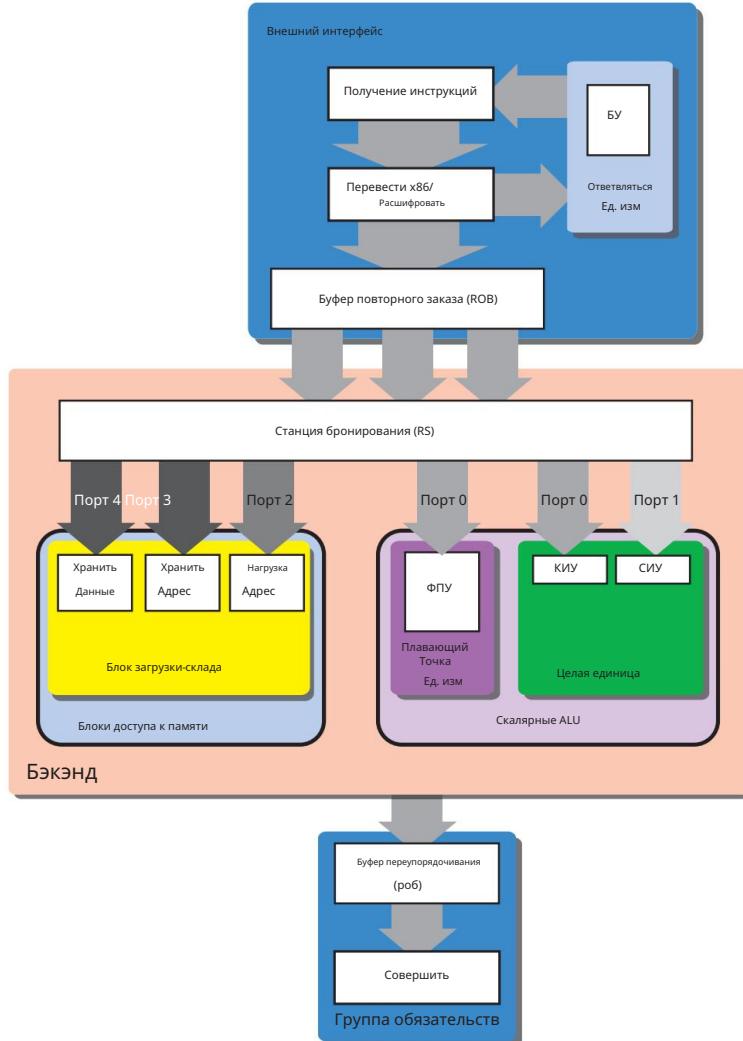


Рисунок 5-6: Pentium Pro

Отделение внешнего интерфейса от внутреннего

В Pentium и его предшественниках инструкции передаются непосредственно от аппаратуры декодирования к аппаратуре исполнения, как показано на рис. 5-7. В этом простом процессоре инструкции статически планируются логикой диспетчера для выполнения двумя АЛУ. Сначала извлекаются и декодируются инструкции. Затем логика диспетчера блока управления проверяет пару инструкций, используя набор запрограммированных правил, чтобы определить, могут ли они выполняться параллельно. Если две инструкции могут выполняться параллельно, блок управления отправляет их на два АЛУ, где они одновременно выполняются в одном и том же тактовом цикле. Когда две инструкции завершили свое выполнение

фазе (т. е. их результаты доступны на шине данных), они возвращаются в программный порядок, и их результаты записываются обратно в регистровый файл в правильной последовательности.

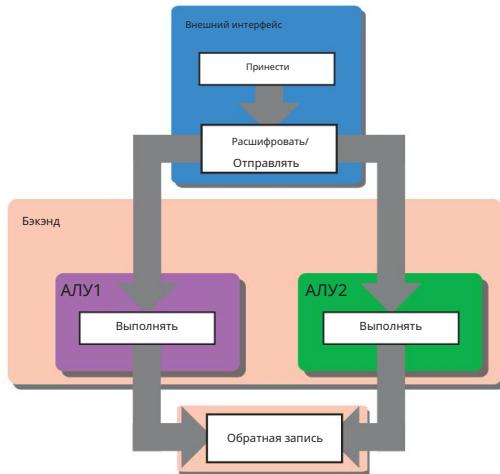


Рисунок 5-7: Статическое планирование в оригинальном Pentium

Этот статический, основанный на правилах подход к диспетчеризации инструкций является жестким и упрощенным, и у него есть два основных недостатка, оба из которых связаны с тем фактом, что, хотя поток кода по своей сути является последовательным, суперскалярный процессор пытается выполнять его части параллельно. В частности, статическое планирование

- z плохо адаптируется к динамичному и постоянно меняющемуся кодовому потоку;
- z плохо использует более широкое суперскалярное оборудование.

Поскольку Pentium может отправлять не более двух операций одновременно от своего оборудования для декодирования к своему оборудованию для выполнения за каждый такт, его правила отправки рассматривают только две инструкции за раз, чтобы определить, могут ли они быть отправлены одновременно. Если бы в Pentium было добавлено больше исполнительного оборудования, а ширина диспетчера была бы увеличена до трех инструкций за такт (как в P6), правила для определения того, куда идут инструкции, должны были бы учитывать различные возможные комбинации две и три инструкции одновременно, чтобы передать эти инструкции нужному исполнительному устройству в нужное время. Кроме того, программистам неизбежно будет трудно оптимизировать такие правила, и если бы они не были чрезмерно сложными, обязательно существовало бы много общих последовательностей инструкций, которые работали бы неоптимально при наборе правил по умолчанию.

Проще говоря, структура потока кода будет меняться от приложения к приложению и от момента к моменту, но правила, отвечающие за планирование выполнения потока кода на серверной части Pentium, останутся навсегда фиксированными.

Фаза выпуска

Решение дилеммы, возникающей при статическом выполнении, заключается в отправке вновь декодированных инструкций в специальный буфер, который находится между внешним интерфейсом и исполнительными модулями. Как только этот буфер соберет несколько инструкций, ожидающих выполнения, логика динамического планирования процессора может проверить инструкции и, приняв во внимание состояние процессора и ресурсы, доступные в настоящее время для выполнения, выдать инструкции из буфера для выполнения единиц в самое подходящее время и в оптимальном порядке. Логика динамического планирования имеет некоторую свободу переупорядочивать поток кода так, чтобы инструкции выполнялись оптимально, даже если это означает, что две (или более) инструкции должны выполняться не только параллельно, но и в обратном порядке. При динамическом планировании текущий контекст, в котором выполняется конкретная инструкция, может иметь гораздо большее влияние на то, когда и как она выполняется. Заменив блок управления Pentium комбинацией буфера и динамического планировщика, микроархитектура P6 заменяет фиксированные правила гибкостью.

Конечно, инструкции, которые были отправлены из буфера в исполнительные блоки не в порядке программы, должны быть возвращены в порядок программы после того, как они завершили свою фазу выполнения, поэтому необходим еще один буфер для захвата инструкций, которые завершили выполнение, и для их обработки. верните их в программный порядок. Мы обсудим этот второй буфер чуть позже.

На рис. 5-8 показаны два новых буфера, оба из которых работают вместе для отделить фазу выполнения от остальной части жизненного цикла инструкции.

В процессоре, изображенном на рис. 5.8, инструкции передаются в порядке программы от этапа декодирования в первый буфер, буфер выдачи, где они находятся до тех пор, пока динамический планировщик процессора не определит, что они готовы к выполнению. Когда инструкции готовы к выполнению, они передаются из буфера задач в исполнительный блок. Этот шаг, когда инструкции перемещаются из буфера задач, где они запланированы для оптимального выполнения, в сами исполнительные блоки, называется выдачей.

Существует ряд факторов, которые могут помешать выполнению инструкции не по порядку, как описано ранее. Ввод инструкции может зависеть от результатов еще не выполненной инструкции, или она может ожидать загрузки данных из памяти, или она может ожидать освобождения занятого исполнительного блока, или любой из может потребоваться выполнение ряда других условий, прежде чем декодированная инструкция будет готова к отправке соответствующему исполнительному устройству. Но как только инструкция готова, планировщик видит, что она передана исполнительному блоку, где она будет выполняться.

Этот новый поворот в стандартном жизненном цикле инструкции называется выполнением не по порядку или динамическим выполнением, и он требует добавления двух новых фаз в жизненный цикл нашей инструкции, как показано в Таблице 5-3. Первая новая фаза — это фаза выдачи, и она включает в себя буферизацию и переупорядочивание потока кода, которые я только что описал.

Фаза выдачи реализована по-разному на разных процессорах. Это может занять несколько этапов конвейера и может включать использование нескольких буферов, расположенных в разных конфигурациях. Что общего у всех различных реализаций, так это то, что инструкции входят в задачу

фазы, а затем ждать там неопределенное количество времени, пока не наступит подходящий момент для их выполнения. Когда они выполняются, они могут делать это не по порядку программы.

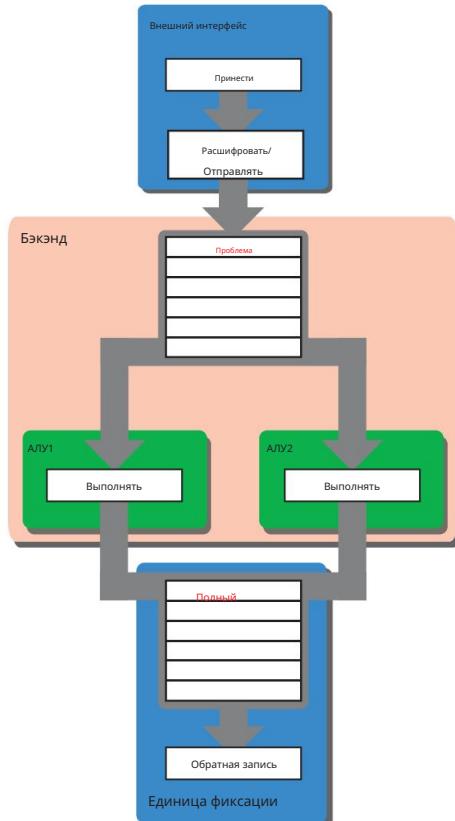


Рисунок 5-8: Динамическое планирование с использованием буферов

Помимо использования в динамическом планировании, еще одной важной функцией буфера задач является то, что он позволяет процессору «выдавливать» пузыри из конвейера до фазы выполнения. Буфер представляет собой очередь, и инструкции, поступающие в нее, выпадают в самую нижнюю доступную запись.

Таблица 5-3: Этапы динамического планирования

Жизненный цикл инструкции

1	Принести	В целях
2	Декодирование/отправка	
3	Выпуск	Изменение порядка
4	Выполнить	Не работает
5	Завершить	Изменение порядка
6	Обратная запись (фиксация)	В целях

Таким образом, если инструкции предшествует облачко конвейера, когда она входит в выдача буфера, он упадет в пустое пространство непосредственно за инструкцией перед ним, тем самым устранив пузыри.

Конечно, способность буфера задач выдавливать пузыри конвейера зависит от способности клиентской части производить больше инструкций за цикл, чем может потреблять серверная часть. Если серверная часть и передняя часть перемещаются на шаге блокировки, пузыри конвейера будут распространяться через очереди задач в заднюю часть.

Фаза завершения

Вторая фаза, которую неупорядоченное выполнение добавляет к жизненному циклу инструкции, — это фаза завершения. На этом этапе инструкции, которые завершили выполнение или завершили выполнение, ожидают во втором буфере, чтобы их результаты были записаны обратно в регистровый файл в порядке программы. Когда результаты выполнения инструкции записываются обратно в регистровый файл и видимое программисту состояние машины постоянно изменяется, говорят, что эта инструкция фиксируется. Инструкции должны выполняться в порядке выполнения программы, чтобы сохранить иллюзию последовательного выполнения. Это означает, что никакая инструкция не может быть зафиксирована до тех пор, пока не будут зафиксированы все инструкции, которые изначально находились перед ней в потоке кода.

Требование того, что все инструкции должны выполняться в исходном программном порядке, делает необходимым второй буфер, показанный на рис. 5-8.

Процессору нужно место для сбора инструкций по мере того, как они завершают фазу неупорядоченного выполнения своего жизненного цикла, чтобы их можно было вернуть в исходный порядок перед отправкой на финальную стадию записи, где они фиксируются. Подобно буферу задач, описанному ранее, этот буфер завершения может принимать различные формы. Вскоре мы рассмотрим форму, которую этот буфер принимает в Р6.

Ранее я говорил, что инструкция находится в буфере фазы завершения, который я сейчас назову буфером завершения, и ожидает, пока ее результат будет записан обратно в регистровый файл. Но где ждать результата инструкции в течение этого промежуточного периода? Когда инструкция выполняется не по порядку, ее результат помещается в специальный регистр переименования, выделенный специально для использования этой инструкцией. Обратите внимание, что этот регистр переименования является частью внутреннего бухгалтерского аппарата процессора, что означает, что он не является частью модели программирования и поэтому не виден программисту. Результат ожидает в этом скрытом регистре переименования до тех пор, пока инструкция не зафиксируется, после чего результат записывается из регистра переименования в видимый для программиста файл архитектурного регистра. После фиксации результата инструкции регистр переименования возвращается в пул доступных регистров переименования, где он может быть назначен другой инструкции в более позднем цикле.

Фаза выпуска Р6: Станция бронирования

Микроархитектура Р6 помещает каждую вновь декодированную инструкцию в буфер, называемый станцией резервирования (RS), где она ожидает, пока не будут выполнены все требования к ее выполнению. После того, как они выполнены, инструкция затем перемещается из станции резервирования в исполнительный блок (т. е. выдается), где она выполняется.

Глядя на диаграмму Р6 (рис. 5-6), видно, что до трех инструкций за цикл может быть отправлено от декодеров на станцию резервирования. И, как вы скоро увидите, до пяти инструкций за цикл может быть отправлено со станции резервирования на исполнительные устройства. Таким образом, первоначальная суперскалярная конструкция Pentium, в которой две инструкции за такт могли отправляться из декодеров непосредственно в серверную часть, была заменена буферизованной схемой, в которой три инструкции могут отправляться в буфер, а пять инструкций могут выдаваться из него в любое время. заданный цикл.

Это действие буферизации и отделение функций выборки/пропускная способность декодирования от пропускной способности серверной части, которую он обеспечивает, лежат в основе прироста производительности Р6.

Фаза завершения Р6: буфер повторного заказа

Поскольку микроархитектура Р6 должна фиксировать свои инструкции по порядку, ей нужно место для отслеживания исходного порядка выполнения каждой инструкции, поступающей на станцию резервирования. Следовательно, после декодирования инструкций они должны пройти через буфер переупорядочивания (ROB), прежде чем попасть на станцию резервирования. ROB подобен большому журналу, в котором Р6 может записывать всю важную информацию о каждой инструкции, поступающей в неупорядоченный сервер. Основная функция ROB состоит в том, чтобы гарантировать, что инструкции выходят с другой стороны неупорядоченной серверной части в том же порядке, в котором они поступили в нее. Другими словами, работа станции резервирования заключается в том, чтобы следить за тем, чтобы инструкции выполнялись в наиболее оптимальном порядке, даже если это означает их выполнение вне порядка программы. Задача буфера переупорядочивания состоит в том, чтобы гарантировать, что завершенные инструкции будут возвращены в программный порядок и что их результаты будут записаны в файл архитектурного регистра в правильной последовательности. С этой целью ROB хранит данные о статусе каждой инструкции, операндах, необходимых регистрах, исходном месте в программе и так далее.

Таким образом, недавно декодированные инструкции поступают в ROB, где их соответствующая информация регистрируется в одной из 40 доступных записей. Оттуда они проходят на станцию бронирования, а затем на заднюю часть. После завершения выполнения они ждут в ROB, пока не будут готовы к фиксации.

Роль, которую я только что описал для буфера переупорядочивания, должна быть знакома вы в этот момент. Буфер переупорядочивания соответствует структуре, которую я ранее назвал буфером завершения, но с некоторыми дополнительными обязанностями, возложенными на него.

Если вы посмотрите на мою диаграмму микроархитектуры Р6, вы заметите, что буфер переупорядочивания изображен в двух местах: передний конец и модуль фиксации. Это связано с тем, что ROB активен на обеих фазах жизненного цикла инструкции. На ROB возложена задача отслеживать инструкции по мере их прохождения через фазы своего жизненного цикла и возвращать инструкции в программный порядок в конце их жизненного цикла. Таким образом, вновь декодированные инструкции должны иметь отслеживаемую запись в ROB и иметь временный регистр переименования, выделенный для их личного использования. Точно так же вновь выполняемые инструкции должны ожидать в ROB, прежде чем они смогут зафиксироваться, поскольку содержимое временного регистра переименования, в котором хранится их результат, постоянно записывается в файл архитектурного регистра.

Как следует из предыдущего предложения, ROB P6 действует не только как буфер завершения и средство отслеживания инструкций, но также обрабатывает переименование регистров. Каждая из 40 записей ROB микроархитектуры P6 имеет поле данных, который содержит данные программы так же, как регистр x86. Эти поля предоставляют серверной части P6 40 микроархитектурных регистров переименования для работы, и они используются в сочетании с таблицей распределения регистров P6 (RAT) для реализации переименования регистров в микроархитектуре P6.

Окно инструкций

Станция резервирования и буфер переупорядочивания вместе составляют основу серверной части процессора P6, работающего в неупорядоченном режиме, и они объясняют его резкое преимущество в производительности такта за такт по сравнению с исходным Pentium. Эти два буфера — один для перетасовки и оптимизации кодового потока (RS), а другой для расстановки и переупорядочивания кодового потока (ROB) — позволяют процессору P6 динамически и разумно адаптировать свою работу в соответствии с потребностями постоянно меняющегося кодового потока. .

Распространенной метафорой для размышлений и разговоров о комбинации RS + ROB в P6 или аналогичных структурах в других процессорах является окно инструкций. ROB P6 может отслеживать до 40 инструкций на различных этапах выполнения, а его станция резервирования может хранить и анализировать до 20 инструкций, чтобы определить оптимальное время их выполнения. Думайте о буфере станции резервирования с 20 инструкциями как об окне, которое перемещается по последовательно упорядоченному кодовому потоку; в любом заданном цикле P6 смотрит через это окно на этот видимый сегмент потока кода и думает о том, как его аппаратное обеспечение может оптимально выполнить 20 или около того инструкций, которые он там видит.

Хорошей аналогией для этого является игра тетрис, где небольшое превью окна показывает вам следующий кусок, который встретится вам, пока вы решаете, как лучше всего разместить падающий в данный момент кусок. Таким образом, в любой момент вы можете видеть всего две части тетриса и думать о том, как эти две должны сочетаться с фигурами, которые были раньше, и с теми, которые могут появиться после.

Работа микроархитектуры P6 немного сложнее, чем средний Tetris игрок, потому что он должен маневрировать и оптимально размещать до трех падающих фигур за раз; следовательно, он должен иметь возможность заглянуть дальше в будущее, чтобы принимать наилучшие решения о том, что размещать, где и когда. Более широкое окно команд P6 позволяет процессору смотреть вперед в потоке кода и жонглировать своими инструкциями так, чтобы они оптимальным образом соответствовали доступным в настоящее время ресурсам выполнения.

Конвейер P6

P6 имеет 12-этапный конвейер, который значительно длиннее, чем пятиэтапный конвейер Pentium. Я не буду перечислять и описывать все 12 стадий по отдельности, а дам общий обзор фаз, через которые проходит конвейер P6.

[Доступ к ВТВ и получение инструкций](#)

Первые три с половиной этапа конвейера посвящены доступу к целевому буферу ветвления и выборке следующей инструкции. Двухтактная фаза выборки инструкций Р6 длиннее, чем однотактная фаза выборки Pentium, но она позволяет задержке доступа к кэшу L1 не сдерживать тактовую частоту процессора в целом.

[Расшифровать](#)

Следующие два с половиной этапа посвящены декодированию инструкций x86 и разбиению их на внутренний, RISC-подобный формат инструкций Р6. Вскоре мы более подробно обсудим эту трансляцию набора инструкций, которая имеет место во всех современных процессорах x86 и даже в некоторых процессорах RISC.

[Зарегистрировать переименовать](#)

На этом этапе выполняются инструкции по переименованию регистров и протоколированию в ROB.

[Пишите в PC](#)

Запись инструкций из ROB в RS занимает один цикл и происходит на этом этапе.

[Читать с PC](#)

На данный момент выполняется этап выпуска жизненного цикла инструкции.

Инструкции могут находиться в RS в течение неопределенного количества циклов, прежде чем будут считаны из RS. Даже если оничитываются из RS сразу после входа в нее, требуется один цикл для перемещения инструкций из RS через порты выдачи в исполнительные устройства.

[Выполнять](#)

Выполнение инструкции может занимать один цикл, как в случае простых целочисленных инструкций, или несколько циклов, как в случае инструкций с плавающей запятой.

[Совершить](#)

Эти два последних цикла посвящены записи результатов выполнения инструкции обратно в ROB, а затем фиксации инструкций путем записи их результатов из ROB в файл архитектурного регистра.

Удлинение конвейера Р6, как описано в этой главе, имеет два основных полезных эффекта. Во-первых, это позволяет Intel увеличить тактовую частоту процессора, поскольку каждый из этапов короче и проще и может быть выполнен быстрее.

Второй эффект немногого более тонкий и менее широко оцененный.

Более длинный конвейер Р6 в сочетании с его буферизованной связкой пропускной способности выборки/декодирования от пропускной способности выполнения позволяет процессору скрывать сбои на этапах выборки и декодирования. Короче говоря, девять этапов конвейера, предшествующих этапу выполнения, объединяются с RS, образуя глубокий буфер для инструкций. Этот буфер может скрывать пробелы и зависания в потоке инструкций примерно так же, как большой резервуар с водой может скрывать перебои с подачей воды на объект.

Но с другой стороны (продолжая пример с резервуаром для воды), когда одно мертвое животное замечают плавающим в резервуаре, все это приходится смыть. Это своего рода случай с Р6 и неправильным предсказанием ветвления.

[Предсказание переходов на Р6](#)

Разработчики Р6 потратили значительно больше ресурсов, чем его предшественник, на прогнозирование ветвлений, и им удалось повысить точность динамического прогнозирования ветвлений с примерно 75 процентов Pentium до более чем 90 процентов. Р6 имеет 512 записей ВНТ + ВТВ и использует четыре бита для записи информации об истории переходов (по сравнению с двухбитным предиктором Pentium). Четырехбитная схема прогнозирования позволяет Pentium хранить больше истории каждого перехода, тем самым повышая его способность правильно прогнозировать результаты перехода.

Как вы узнали из главы 2, предсказание ветвлений становится все более важным по мере того, как конвейеры становятся длиннее, потому что сброс конвейера из-за неправильного предсказания означает большее количество потерянных циклов и более длительное время восстановления пропускной способности процессора и скорости выполнения.

Рассмотрим случай условного перехода, результат которого зависит от результата целочисленного вычисления. В оригинальном Pentium расчет происходит на четвертом этапе конвейера, и если блок прогнозирования ветвлений (BPU) ошибся, при сбросе конвейера будут потеряны только три рабочих цикла. Однако на Р6 условное вычисление не выполняется до этапа 10, что означает, что 10 циклов работы сбрасываются, если BPU делает неверный выбор.

Когда процессор с динамическим планированием выполняет инструкции спекулятивно, эти спекулятивные инструкции и их результаты сохраняются в ROB точно так же, как и неспекулятивные инструкции. Однако записи ROB для спекулятивных инструкций помечаются как спекулятивные и не могут быть зафиксированы до тех пор, пока не будет оценено условие ветвления. После оценки условия ветвления, если BPU угадал правильно, записи ROB спекулятивных инструкций помечаются как неспекулятивные, и инструкции фиксируются по порядку. Если BPU угадал неправильно, спекулятивные инструкции и их результаты удаляются из ROB без фиксации.

[Серверная часть Р6](#)

Задняя часть Р6 (показана на рис. 5-9) значительно шире, чем у Pentium. Как и Pentium, он содержит два асимметричных целочисленных ALU и отдельный блок с плавающей запятой, но его возможности загрузки-сохранения были расширены за счет включения трех исполнительных блоков, предназначенных исключительно для доступа к памяти: блока адреса загрузки, блока адреса сохранения и блока адреса хранения, единица хранения данных. Блоки адреса загрузки и адреса хранения содержат по паре сумматоров с четырьмя входами для вычисления адресов и проверки пределов сегментов; это сумматоры, которые появляются на этапе декодирования-1 оригинального Pentium.

Асимметричные целочисленные ALU на Р6 имеют пропускную способность за один цикл. и задержка для большинства операций, при этом умножение имеет пропускную способность в один цикл, но задержку в четыре цикла. Таким образом, инструкции умножения выполняются быстрее на Р6, чем на Pentium.

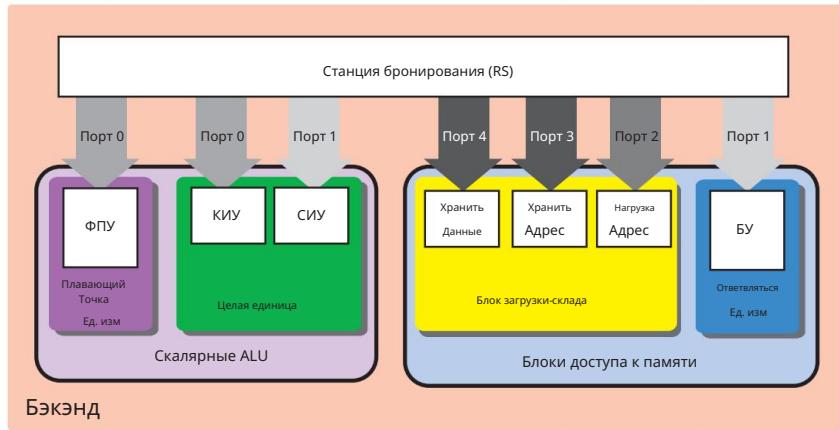


Рисунок 5-9: Серверная часть Р6

Блок вычислений с плавающей запятой в Р6 выполняет большинство операций с одинарной и двойной точностью за три такта, при этом пять циклов необходимы для инструкций умножения. FPU полностью конвейерен для большинства инструкций, так что большинство инструкций выполняются с пропускной способностью в один такт. Некоторые инструкции, такие как деление с плавающей запятой и квадратный корень, не являются конвейерными и занимают от 18 до 38 и от 29 до 69 циклов соответственно.

С точки зрения настоящего обзора наиболее примечательной особенностью Внутренней частью Р6 является то, что его исполнительные блоки подключены к станции резервирования через пять портов выдачи, как показано на рис. 5-9.

Это означает, что за такт от станции резервирования через порты выдачи в исполнительные блоки может пройти до пяти инструкций. Эти пять-

Структура порта выдачи — одна из самых узнаваемых черт Р6, и когда более поздние разработки (например, PII) добавляли исполнительные блоки к микроархитектуре (например, блоки MMX), их приходилось добавлять к существующим портам выдачи.

Если вы внимательно посмотрели на схему Pentium Pro, то наверняка заметили что в исходном Pentium Pro уже было два модуля, которые совместно использовали один порт: модуль простых целых чисел и модуль с плавающей запятой. Это означает, что существуют некоторые ограничения на выполнение второго целочисленного вычисления и вычисления с плавающей запятой, но эти ограничения редко влияют на производительность.

CISC, RISC и трансляция набора инструкций

Как и оригинальный Pentium, Р6 тратит дополнительное время на фазу декодирования, но на этот раз дополнительные полтора цикла идут не на вычисления адреса, а на преобразование набора команд. Преобразование ISA — важный метод, используемый во многих современных процессорах, но прежде чем вы сможете понять, как он работает, вы должны сначала познакомиться с двумя терминами, которые часто встречаются в обсуждениях компьютерной архитектуры: RISC и CISC.

Одним из наиболее важных отличий x86 ISA от PowerPC ISA (описанного в следующей главе) и гипотетической DLW ISA, представленной в главе 2, является то, что она поддерживает операции «регистр-в-память» и «память-в-память». форматирование арифметических инструкций. В архитектуре DLW

операнды источника и назначения для каждой арифметической инструкции должны были быть либо регистрами, либо непосредственными значениями, и программист отвечал за включение в программу инструкций загрузки и сохранения, необходимых для обеспечения того, чтобы регистры источника арифметических инструкций были заполнены правильными значениями из памяти, а их результаты записывались обратно в память.

В архитектуре x86 программист может добровольно передать контроль над большей частью трафика загрузки-сохранения процессору, используя исходные и/или целевые операнды, которые являются ячейками памяти. Если бы архитектура DLW поддерживала такие операции, они могли бы выглядеть как программа 5-2:

Номер строки	Код	Комментарии
1	add #12, #13, A	Сложите содержимое ячеек памяти #12 и #13 и поместите результат в регистр A.
2	sub A, #15, #16	Вычтите содержимое регистра A из содержимого ячейки памяти #15 и сохраните результат в ячейке памяти #16.
3	sub A, #B, #100	Вычтесь содержимое регистра A из содержимого ячейки памяти, на которую указывает #B, и сохранить результат в ячейке памяти №100.

Программа 5-2. Арифметические инструкции с использованием форматов память-память и память-регистр

Добавление содержимого двух ячеек памяти, как в строке 1 программы 5-2, по-прежнему требует от процессора загрузки необходимых значений в регистры и сохранения результатов. Однако в инструкциях формата «память-регистр» и «память-память» эти операции загрузки и сохранения являются неявными в инструкции.

Процессор должен просмотреть инструкцию и выяснить, что ему нужно выполнить необходимые обращения к памяти; то он должен выполнить их до и/или после выполнения арифметической части инструкции. Таким образом, для сложения в строке 1 программы 5-2 процессор должен будет выполнить две загрузки перед выполнением сложения. Точно так же для вычитаний в строках 2 и 3 процессор должен будет выполнить одну загрузку перед выполнением вычитания и одно сохранение после него.

Использование таких инструкций формата «регистр-в-память» и «память-в-память» переносит бремя планирования трафика памяти с программиста на процессор, позволяя программисту сосредоточиться на других аспектах кодирования. Это также приводит к уменьшению количества инструкций, которые программист должен написать (или плотности кода) для выполнения большинства задач. В те дни, когда программисты программировали в основном на ассемблере, компиляторы для языков высокого уровня (HLL), таких как С и FORTRAN, были примитивными, а оперативная память была маленькой и дорогой, такие качества ISA, как простота использования программиста и высокая плотность кода, были очень важны . привлекательный.

Еще один метод, используемый ISA, такими как x86, для снижения нагрузки на программистов и увеличения плотности кода, заключается в включении на уровне ISA поддержки сложных типов данных, таких как строки. Стока — это просто последовательность, или «строка», непрерывных ячеек памяти определенной длины. Строки часто используются для хранения текста ASCII, поэтому в короткой строке может храниться слово, а в более длинной строке может храниться целое предложение. Если ISA включает в себя инструкции по работе со строками

x86 ISA умеет — программисты на ассемблере могут писать такие программы, как текстовые редакторы и терминальные приложения, за гораздо более короткий промежуток времени и со значительно меньшим количеством инструкций, чем они могли бы, если бы ISA не имела такой поддержки.

Сложные инструкции, такие как инструкции по работе со строками x86, выполняют сложные многошаговые задачи и, следовательно, заменяют то, что в противном случае представляло бы собой множество строк кода на ассемблере RISC. Однако у этих типов инструкций есть серьезные недостатки, когда речь идет о динамическом планировании и неупорядоченном выполнении, описанном ранее в этой главе. Строковые инструкции, например, имеют задержки, которые могут варьироваться в зависимости от длины обрабатываемой строки: чем длиннее строка, тем больше циклов требуется для выполнения инструкции. Поскольку их задержки непредсказуемы, процессору сложно оптимально планировать их с помощью механизмов динамического планирования, описанных ранее.

Наконец, сложные инструкции часто различаются по количеству байтов, необходимых им для отображения на машинном языке. Такие инструкции переменной длины труднее получить и декодировать, а после декодирования их сложнее запланировать.

Из-за использования нескольких форматов инструкций (регистр в память и память в память) и сложных инструкций переменной длины, x86 является примером подхода к процессору и дизайну ISA, называемого сложным вычислением набора инструкций (CISC). И DLW, и PowerPC, напротив, представляют подход, называемый вычислениями с сокращенным набором команд (RISC), в котором все инструкции машинного языка имеют одинаковую длину, поддерживается меньшее количество форматов инструкций, а сложные инструкции полностью исключаются.

Для ISA RISC труднее программировать на языке ассемблера, поэтому они предполагают существование и широкое использование языков высокого уровня и сложных компиляторов. Для RISC-программистов, использующих язык высокого уровня, такой как C, бремя планирования трафика памяти и обработки сложных типов данных перекладывается с процессора на компилятор. Перекладывая бремя планирования доступа к памяти и других типов кода на компилятор, процессоры, реализующие RISC ISA, могут быть менее сложными, а поскольку они менее сложные, они могут быть быстрее и эффективнее.

Было бы неплохо, если бы x86, самая популярная в мире ISA, была RISC, но это не так. x86 ISA является хрестоматийным примером CISC ISA, а это означает, что процессоры, реализующие x86, требуют более сложных микроархитектур. В какой-то момент разработчики процессоров x86 поняли, что для того, чтобы использовать новейшие методы динамического планирования, ориентированные на RISC, для ускорения архитектуры на основе x86 без выхода из-под контроля сложности процессора, им придется ограничить дополнительную сложность внешнего интерфейса путем перевод операций x86 CISC в более мелкие, быстрые и более единообразные RISC-подобные операции для использования в серверной части. AMD K6 и Intel P6 были двумя ранними разработками x86, которые использовали этот тип преобразования набора инструкций с большим преимуществом. Этот метод оказался настолько успешным, что все последующие процессоры x86 от Intel и AMD использовали трансляцию набора команд, как и некоторые процессоры RISC, такие как IBM PowerPC 970.

Блок декодирования инструкций микроархитектуры Р6

Микроархитектура Р6 разбивает сложные инструкции x86 переменной длины на одну или несколько меньших микроопераций фиксированной длины (также называемых микрооперациями, микрооперациями или моопами) с использованием блока декодирования, состоящего из трех отдельных декодеров, изображенных на рис. 5-10: два простых/быстрых декодера, которые обрабатывают простые инструкции x86 и могут производить одну декодированную микрооперацию за цикл; и один сложный/медленный декодер, который обрабатывает более сложные инструкции x86 и может производить до четырех декодированных микроопераций за цикл.

Группы инструкций x86 по 16 байт извлекаются из I-cache в 32-байтовую очередь инструкций внешнего интерфейса, где логика предварительного декодирования сначала определяет границы и тип каждой инструкции, прежде чем выравнивать инструкции для входа в аппаратное обеспечение декодирования. Затем до трех инструкций x86 за цикл могут перемещаться из очереди инструкций в декодеры, где они преобразуются в микрооперации и передаются в очередь микроопераций перед отправкой в ROB. Вместе три декодера Р6 способны производить до шести декодированных микроопераций за цикл (один медленный декодер плюс по одному от каждого из двух простых/быстрых декодеров) для потребления очередью микроопераций. Очередь микроопераций, в свою очередь, способна передавать до трех микроопераций за цикл в окно инструкций Р6.

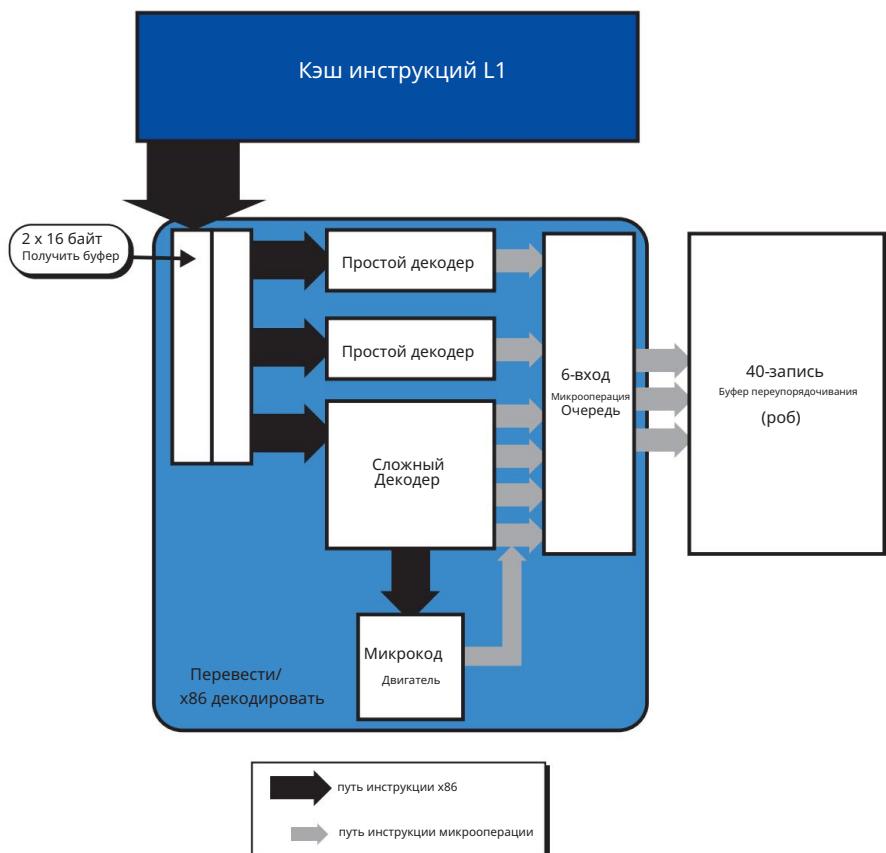


Рисунок 5-10: Аппаратное обеспечение декодирования микроархитектуры Р6

Простые инструкции x86, которые можно декодировать очень быстро и которые разбиваются только на одну или две микрооперации, на сегодняшний день являются наиболее распространенным типом инструкций, встречающихся в средней программе x86. Таким образом, P6 посвящает большую часть своего оборудования для декодирования этим типам инструкций. Более сложные инструкции x86, такие как инструкции по работе со строками, менее распространены и требуют больше времени для декодирования. Сложный/медленный декодер P6 работает в сочетании с ПЗУ микрокода для обработки действительно сложных устаревших инструкций, которые преобразуются в последовательности микроопераций, которые считаются непосредственно из ПЗУ.

[Стоимость устаревшей поддержки x86 на P6](#)

Все это аппаратное обеспечение для декодирования и трансляции занимает много транзисторов. По оценкам MDR, почти 40 процентов бюджета транзисторов P6 тратится на поддержку устаревшей архитектуры x86. Если это так, то это даже выше, чем поразительная оценка в 30% для исходного Pentium, и даже если это неверно, это все равно говорит о том, что стоимость поддержки устаревших процессоров довольно высока.

В этот момент вы, вероятно, вспомните заключение первой части этой главы, в котором я предположил, что относительная стоимость поддержки x86 снижалась с последующими поколениями Pentium. Это по-прежнему верно, но эта тенденция не сохранилась для первого экземпляра микроархитектуры P6: оригинального 133-мегагерцового Pentium Pro. Кэш L1 Pentium Pro составлял скромные 16 КБ, что было мало даже по меркам 1995 года. Разработчикам чипа пришлось экономить на кэш-памяти на кристалле, потому что они потратили большую часть бюджета транзистора на оборудование для декодирования и трансляции. Сопоставимые RISC-процессоры имели в два-четыре раза больше кэш-памяти, потому что меньшая часть кристалла занимала интерфейсную логику, поэтому они могли использовать пространство для кэш-памяти.

Когда микроархитектура P6 впервые была запущена в ее воплощении Pentium Pro, количество транзисторов по сегодняшним меркам все еще было относительно небольшим.

Но по мере того, как Кривые Мура продвигались вперед, разработчики микропроцессоров перестали думать: «Как нам втиснуть все аппаратное обеспечение, которое мы хотели бы поставить на чип, в наш транзисторный бюджет?» до «Теперь наш щедрый бюджет на транзисторы позволит нам делать действительно хорошие вещи!» на «Как заставить такое количество транзисторов выполнять полезную работу, повышающую производительность?»

Что действительно привело к снижению затрат последующих поколений на поддержку x86, так это увеличение размеров кэша L1 и перемещение кэша L2 на кристалл, потому что ответ на этот последний вопрос — до недавнего времени — был «Давайте добавим кэш».

[Резюме: микроархитектура P6 в историческом контексте](#)

В этом заключительном разделе представлен обзор микроархитектуры P6 в ее различных воплощениях. Основное внимание здесь уделяется тому, чтобы собрать все вместе и дать вам представление о том, как развивался P6. Историческое повествование, изложенное в этом разделе, кажется в ретроспективе разворачивающимся в течение гораздо более длительного периода времени, чем семь лет, которые фактически потребовались для перехода от Pentium Pro к Pentium 4, но семь лет — это вечность в компьютерном времени. .

Пентиум Про

Процессор, описанный в предыдущем разделе под названием Р6 , представляет собой оригинальный Pentium Pro с тактовой частотой 133 МГц. Как видно из сравнения процессоров в Таблице 5-2, у Pentium Pro было относительно мало транзисторов, мало кэш-памяти и мало возможностей. На самом деле, исходный Pentium в конце концов получилrudиментарную поддержку SIMD-вычислений в виде Intel MMX (Multimedia Extensions), но в Pentium Pro не было достаточно места для этого, поэтому от SIMD отказались в пользу всей этой причудливой логики декодирования, описанной ранее. .

Однако, несмотря на все свои недостатки, Pentium Pro удалось значительно поднять планку производительности x86. Механизм выполнения вне очереди, двойные целочисленные конвейеры и усовершенствованный блок вычислений с плавающей запятой придали ему достаточно сил, чтобы вывести x86 ISA на рынок массовых серверов.

Пентиум II

MMX не возвращался в линейку продуктов Intel до Pentium II.

Представленное в 1997 году, это следующее поколение микроархитектуры Р6 дебютировало на скоростях в диапазоне от 233 до 300 МГц и имело ряд улучшений, повышающих производительность по сравнению с его предшественником.

Первым среди этих улучшений был раздельный кэш L1 на кристалле, размер которого был удвоен до 32 КБ. Этот более крупный L1 помог повысить производительность по всем направлениям, сохраняя длинный конвейер PII полным кода и данных.

Базовый конвейер Р6 остался прежним в PII, но Intel расширила его. серверной части, как показано на рис. 5-11, путем добавления вышеупомянутой поддержки MMX в виде двух новых исполнительных модулей MMX: одного на порту 0, а другого на порту 1. Однако MMX обеспечивает поддержку векторов только для целых чисел. Только после появления Streaming SIMD Extensions (SSE) с PIII микроархитектура Р6 получила поддержку векторной обработки с плавающей запятой.

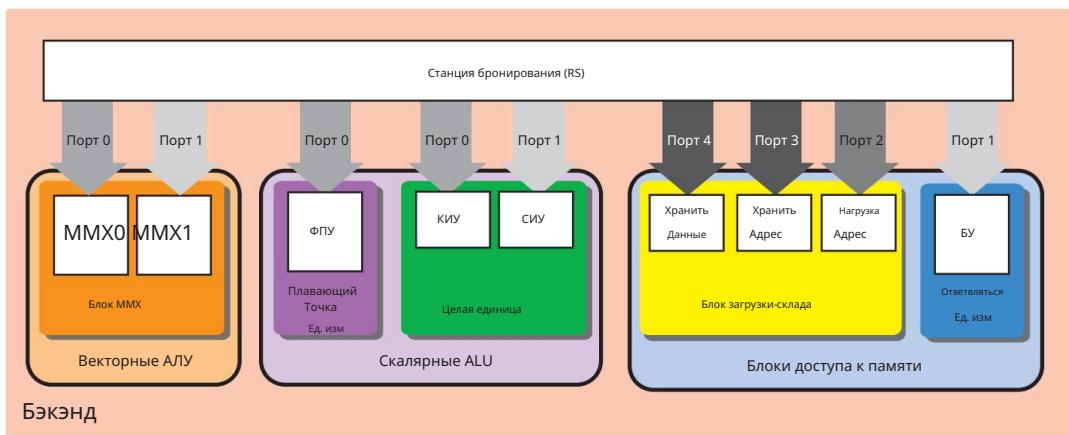


Рисунок 5-11: Внутренний интерфейс Pentium II

Производительность Pentium II для целочисленных операций и вычислений с плавающей запятой была относительно невысокой. хорошо по сравнению с его конкурентами CISC, и это способствовало развитию тенденции, начатой Pentium Pro, миграции массового оборудования x86 в области серверов и рабочих станций. Однако PII по-прежнему не мог конкурировать с конструкциями RISC, построенными на том же процессе с аналогичным количеством транзисторов. Его главное преимущество заключалось в эффективности, в то время как более дорогие RISC-чипы специализировались на чистой эффективности.

Пентиум III

Intel представила свою следующую производную Р6, Pentium III (PIII), в 1999 году с тактовой частотой 450 МГц и производственным процессом 0,25 микрона. Первая версия Pentium III под кодовым названием Katmai имела кэш-память второго уровня объемом 512 КБ, которая разделяла небольшую часть печатной платы (называемую дочерней платой) с PIII. Хотя эта конструкция обеспечивала неплохую производительность, PIII действительно не рос с точки зрения производительности до тех пор, пока в начале 2000 года не была представлена следующая версия PIII под кодовым названием Coppermine .

Coppermine производился по 0,18-микронному техпроцессу, а это означает, что Intel могла разместить больше транзисторов на кристалле процессора.

Intel воспользовалась этой возможностью, уменьшив размер кэш-памяти второго уровня PIII до 256 КБ и переместив кэш-память на сам кристалл ЦП. Наличие L2 на том же кристалле, что и ЦП, и кэш L1, значительно сократило время доступа к кэшу L2, что более чем компенсировало уменьшение размера кэша. Производительность Coppermine хорошо масштабировалась с увеличением тактовой частоты, и в конечном итоге он преодолел отметку в 1 ГГц вскоре после AMD Athlon.

Процессор Pentium III представил два значительных дополнения к x86 ISA, наиболее важным из которых был набор расширений SIMD с плавающей запятой для архитектуры x86, называемых Streaming SIMD Extensions (SSE). С добавлением 70 новых инструкций SSE архитектура x86 дополннила гораздо больше того, чего не хватало в ее поддержке векторных вычислений, что сделало ее более привлекательной для таких приложений, как игры и цифровая обработка сигналов. Я расскажу о расширениях MMX и SSE более подробно в главе 8, а пока необходимо сказать несколько слов о том, как эти расширения были реализованы на аппаратном уровне.

Разработчики Pentium III добавили большую часть нового оборудования SSE к порту 1 выпуска (см. заднюю часть на рис. 5-12). Новые модули SSE, подключенные к порту 1, выполняют векторное сложение SIMD, перемешивание и обратные арифметические функции. Intel также модифицировала FPU на порту 0 для обработки умножения SSE.

Таким образом, основной функциональный блок FPU Pentium III отвечает как за скалярные, так и за векторные операции.

PIII также представил печально известный серийный номер процессора (PSN) вместе с новыми инструкциями x86, предназначенными для чтения номера. PSN был уникальным серийным номером, которым обозначался каждый процессор, и он предназначался для использования в целях защиты коммерческих транзакций в Интернете. Однако из-за опасений защитников конфиденциальности PSN в конечном итоге была исключена из линейки Pentium.

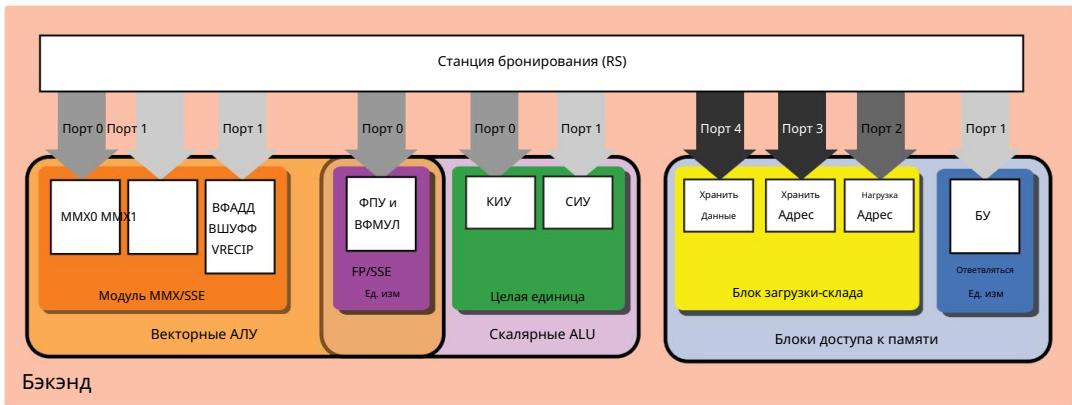


Рисунок 5-12: Внутренний интерфейс Pentium III

Вывод

Pentium, возможно, не превзошел своих RISC-современников, но он был достаточно превосходен по сравнению с конкурентами на базе архитектуры x86, чтобы удерживать Intel на рынке массовых ПК. Действительно, до того, как компания Advanced Micro Devices (AMD) стала серьезным конкурентом, Intel могла позволить себе роскошь задавать темпы прогресса в области ПК с архитектурой x86. Продукты были выпущены, когда Intel была готова их выпустить, и тактовые частоты выросли, когда Intel была готова к этому. Конкурентам Intel приходилось реагировать на то, что делал более крупный производитель микросхем, поскольку их собственные продукты x86 всегда значительно отставали от продуктов Intel по производительности и популярности.

Athlon от AMD был первым процессором x86, который представлял какую-либо угрозу техническому доминированию Intel, и к тому времени, когда PIII дебютировал в 1999 году, стало ясно, что Intel и AMD сошлись в «гонке гигагерц», чтобы выяснить, кто окажется первым. Первым представил процессор с тактовой частотой 1 ГГц. Микроархитектура Р6 в ее воплощении PIII была лошадью Intel в этой гонке, и эта базовая конструкция в конечном итоге достигла отметки в 1 ГГц вскоре после AMD Athlon. Таким образом, микроархитектура, которая начиналась с частоты 150 МГц, в конечном итоге вывела x86 за пределы 1 ГГц и на прибыльные рынки серверов и рабочих станций, где традиционно доминировали архитектуры RISC.

Гонка гигагерцов оказала глубокое влияние не только на массовые ПК. рынке, но и на самой линейке Pentium, поскольку следующий чип, носящий имя Pentium — Pentium 4 — нес на себе печать гигагерцовой гонки в самой своей архитектуре. Если Intel чему-то и научилась за последние несколько лет жизни Р6, так это тому, что тактовая частота продается, и она помнила об этом прежде всего, когда проектировала микроархитектуру NetBurst для Pentium 4. (Подробнее о Pentium 4 см. в главах 7 и 8.)

6

ПРОЦЕССОРЫ POWERPC: СЕРИЯ 600, СЕРИЯ 700, И 7400

Теперь, когда вы познакомились с первой половиной линейки Intel Pentium в предыдущей главе, в этой главе основное внимание будет уделено происхождению и развитию другого популярного семейства микропроцессоров: линейки процессоров PowerPC (или PPC), созданной совместными усилиями Apple, IBM и Motorola.

Поскольку семейство процессоров PowerPC чрезвычайно велико и может использоваться во множестве приложений, начиная от мэйнфреймов и настольных ПК и заканчивая маршрутизаторами и игровыми консолями, описание PowerPC в этой главе представляет собой лишь небольшую и ограниченную выборку процессоров, реализующих PowerPC ISA. В частности, эта глава будет посвящена исключительно подмножеству чипов PowerPC, поставляемых в продуктах Apple, поскольку эти чипы наиболее непосредственно сопоставимы с линейкой Pentium в том смысле, что они нацелены на рынок «персональных компьютеров».

Краткая история PowerPC

Архитектура PowerPC уходит своими корнями в две отдельные архитектуры. Первая из них — это архитектура под названием POWER (оптимизация производительности с расширенным RISC), RISC-архитектура IBM, разработанная для использования в мейнфреймах и серверах. Второй — процессор Motorola 68000 (он же 68K), который до PowerPC составлял основу линейки настольных компьютеров Apple.

Короче говоря, IBM нужен был способ превратить POWER в более широкий спектр вычислительных продуктов для использования за пределами серверного шкафа, Motorola нуждался в высокопроизводительном микропроцессоре RISC, чтобы конкурировать на рынке рабочих станций RISC, а Apple нуждался в процессор для своих персональных компьютеров, который был бы одновременно передовым и обратно совместимым с 68K.

Так родился альянс AIM (Apple, IBM, Motorola), а вместе с ним и подмножество архитектуры POWER, получившее название PowerPC. Процессоры PowerPC должны были быть совместно разработаны и произведены IBM и Motorola при участии Apple и должны были использоваться в компьютерах Apple и на рынке встраиваемых систем. Альянс AIM с тех пор ушел в историю, но PowerPC живет не только в компьютерах Apple, но и во множестве различных продуктов, использующих чипы на базе PowerPC от Motorola и IBM.

PowerPC 601

В 1993 году AIM положила начало вечеринке PowerPC, выпустив 32-разрядный PowerPC 601 с начальной частотой 66 МГц. Модель 601, основанная на старом процессоре IBM RISC Single Chip (RSC) и изначально предназначенная для использования в качестве «моста» между POWER и PowerPC, объединяет части архитектуры IBM POWER с шиной 60x, разработанной Motorola для использования с их 88000. В качестве моста 601 поддерживает объединение наборов инструкций POWER и PowerPC, и это позволило первым авторам приложений PowerPC легко перейти от старой ISA к новой.

ПРИМЕЧАНИЕ Термин «32-разрядный» может быть вам незнаком. Если вам интересно, что это значит, вы можете пропустить главу 9, посвященную 64-битным вычислениям.

В Табл. 6-1 приведены характеристики PowerPC 601.

Табл. 6-1: Характеристики PowerPC 601

Дата введения	14 марта 1994 г.
Процесс	0,60 мкм
Количество транзисторов	2,8 миллиона
Размер штампа	121 мм ²
Тактовая частота при введении	60–80 МГц
Размеры кэша	32 КБ унифицированный L1
Впервые появился в	Мощность Макинтош 6100/60

Несмотря на то, что у совместной команды IBM-Motorola в Остине, штат Техас, было всего 12 месяцев, чтобы запустить этот чип, это был очень хороший и полнофункциональный RISC-проект для своего времени.

[Трубопровод и передняя часть 601-го](#)

В предыдущей главе вы узнали, насколько сложными обычно бывают внешние интерфейсы и конвейеры различных процессоров Pentium. Ничего этого нет в 601, который имеет классический четырехэтапный целочисленный конвейер RISC:

1. Получить
2. Декодировать/отправить
3. Выполнить
4. Обратная запись

Тот факт, что все RISC-инструкции PowerPC имеют одинаковый размер, означает, что логика выборки инструкций процессора 601 не имеет проблем с выравниванием инструкций, характерных для архитектуры x86, и поэтому аппаратное обеспечение выборки проще и быстрее. В те времена, когда бюджеты транзисторов были ограничены, такие вещи могли иметь большое значение в производительности, энергопотреблении и стоимости.

[Очередь инструкций PowerPC](#)

Как вы можете видеть на рис. 6-1, до восьми инструкций за цикл могут быть выбраны непосредственно в очередь инструкций с восемью входами (IQ), где они декодируются перед отправкой на серверную часть. Привыкайте видеть очередь инструкций, потому что она присутствует в той или иной форме в каждой отдельной модели PPC, которую мы будем обсуждать в этой книге, вплоть до PPC 970.

Очередь инструкций используется в основном для обнаружения и обработки ветвей. Модуль ветвления 601 сканирует четыре нижние записи очереди, идентифицируя инструкции ветвления и определяя их тип (условный, безусловный и т. д.). В случаях, когда модуль перехода имеет достаточно информации для немедленного разрешения перехода (например, в случае безусловного перехода или условного перехода, условие которого зависит от информации, которая уже находится в регистре условий), инструкция перехода просто удаляется из очереди инструкций и заменяется инструкцией, расположенной в цели перехода.

ПРИМЕЧАНИЕ. Регистр состояния PowerPC является аналогом слова состояния процессора Pentium.

Мы обсудим регистр условий более подробно в главе 10.

Эта техника устранения ветвей, называемая складыванием ветвей, ускоряет работу двумя способами. Во-первых, он исключает инструкцию (ветвь) из потока кода, что освобождает пропускную способность диспетчера для других инструкций.

Во-вторых, он устраниет пузыри конвейера с одним циклом, которые обычно возникают сразу после ответвления. Все процессоры PowerPC, описанные в этой главе, выполняют свертывание ветвей.

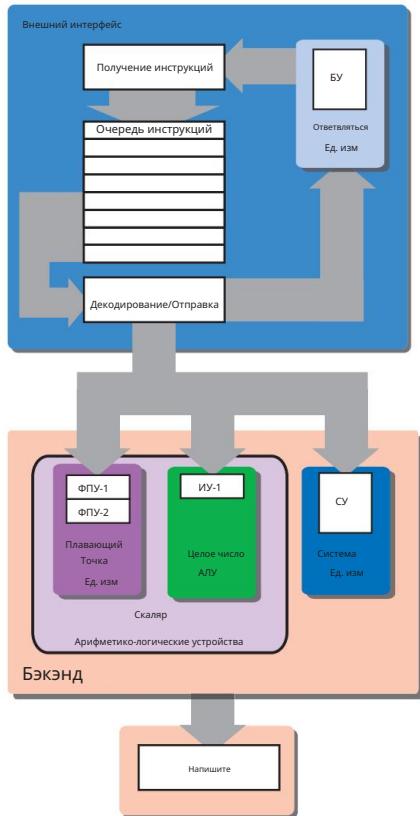


Рисунок 6-1: Микроархитектура PowerPC 601

Если модуль ветвления определяет, что ветвь не принята, он позволяет ветвям перейти в конец очереди, где логика диспетчера просто удаляет ее из потока кода. Действие, позволяющее неиспользованным ветвям выпадать из очереди инструкций, называется прохождением, и оно происходит на всех процессорах PowerPC, рассматриваемых в этой книге.

Инструкции без ветвления и инструкции ветвления, которые не свернуты, находятся в очереди инструкций, в то время как логика отправки проверяет четыре самые нижние записи, чтобы увидеть, какие три из них она может отправить на сервер в следующем цикле. Логика отправки может отправлять до трех инструкций за цикл не по порядку из четырех нижних элементов очереди с некоторыми ограничениями, одно из которых является наиболее важным для наших непосредственных целей: Целочисленные инструкции могут быть отправлены только из самого нижнего элемента очереди.

Планирование инструкций на 601

Обратите внимание, что процессор 601 не имеет аналога буфера переупорядочивания Pentium Pro (ROB) для отслеживания исходного порядка выполнения программы. Вместо этого инструкции помечаются тем, что составляет метаданные, чтобы логика обратной записи могла зафиксировать результаты в регистровом файле в порядке программы. Этот метод пометки инструкций метаданными порядка выполнения программы прекрасно работает для простой конструкции со статическим планированием, такой как 601, с очень небольшим количеством выполняемых операций.

инструкции. Но позже для динамически запланированных проектов PPC потребуются специальные структуры для отслеживания большего количества выполняемых инструкций и обеспечения того, чтобы они фиксировали свои результаты по порядку.

Задняя часть 601-го

Со стадии отправки инструкции поступают в серверную часть 601, где они выполняются каждым из трех различных исполнительных блоков: целочисленным блоком, блоком с плавающей запятой или блоком ветвления. Давайте рассмотрим каждый из этих блоков по очереди.

Целочисленная единица

32-битный целочисленный блок 601 представляет собой простое ALU с фиксированной запятой, которое отвечает за все целочисленные вычисления, включая вычисление адресов.

на чипе. В то время как конструкции x86, как и исходный Pentium, нуждаются в дополнительных сумматорах адресов, чтобы все вычисления адресов, связанные с множественностью режимов адресации x86, не связывали целочисленное аппаратное обеспечение серверной части, RISC-модель процессора 601 с загрузкой-хранением означает, что он реально может обрабатывать трафик памяти и обычный трафик ALU с помощью одного целочисленного исполнительного блока.

Таким образом, целочисленный блок 601 обрабатывает следующие связанные с памятью функции, большинство из которых в последующем перемещаются в выделенный блок загрузки-сохранения.

Конструкции КПП:

- z вычисления адресов загрузки с целыми числами и числами с плавающей запятой
- z вычисление адресов хранения с целыми числами и числами с плавающей запятой
- z операции загрузки данных с целыми числами и с плавающей запятой
- z целочисленные операции хранения данных

Впихивание всех этих функций загрузки-сохранения в одно целочисленное ALU процессора 601 не совсем улучшает целочисленную производительность чипа, но этого достаточно, чтобы не отставать от Pentium в этой области, даже несмотря на то, что Pentium имеет два целочисленных ALU. Большая часть этого целочисленного паритета производительности, вероятно, обусловлена огромным 32-килобайтным унифицированным кэшем L1 процессора 601 (сравните его с 8-килобайтным разделенным кэшем L1 Pentium) — роскошью, доступной для 601 благодаря относительной простоте аппаратного декодирования внешнего интерфейса.

Последний момент, который стоит отметить в отношении целочисленных единиц 601, заключается в том, что многоцикловые целочисленные инструкции (например, целочисленные умножения и деления) не полностью конвейеризованы. Когда инструкция, выполнение которой занимает, скажем, пять циклов, поступает в IU, она связывает весь IU на все пять циклов. К счастью, наиболее распространенными целочисленными инструкциями являются инструкции с одним циклом.

Модуль с плавающей запятой

Благодаря одному блоку вычислений с плавающей запятой, который выполняет все вычисления с плавающей запятой и операции сохранения адресов, модель 601 показала очень высокие результаты при первом запуске.

Конвейер операций с плавающей запятой процессора 601 состоит из шести этапов и включает в себя четыре основных этапа, описанных ранее в этой главе, но с дополнительным этапом декодирования и дополнительным этапом выполнения. Что на самом деле задает плавающую точку чипа

Помимо аппаратного обеспечения по сравнению с его современниками, является тот факт, что не только почти все операции с одинарной точностью полностью конвейеризированы, но и большинство операций с плавающей запятой с двойной точностью (64-бит). Это означает, что для операций с одинарной точностью (за исключением деления) и большинства операций с двойной точностью аппаратное обеспечение процессора 601 с плавающей запятой может выполнять одну инструкцию за цикл с задержкой в два цикла.

Еще одна замечательная особенность FPU процессора 601 — его способность выполнять инструкции плавного умножения-сложения одинарной точности (fmadd) с пропускной способностью в один цикл. fmadd — является базовой функцией цифровой обработки сигналов (DSP) и научных вычислений, поэтому быстродействующие возможности 601 fmadd делают его хорошо подходящим для приложений такого типа. Эта возможность fmadd с одним циклом на самом деле является важной особенностью всей линейки вычислений PowerPC, начиная с 601 и заканчивая сегодняшним днем, и это одна из причин, по которой эти процессоры так популярны для средств массовой информации и научных приложений.

Еще одним фактором доминирования процессора 601 в операциях с плавающей запятой является то, что его целочисленный блок обрабатывает весь трафик памяти (при этом FPU предоставляет данные для хранения данных с плавающей запятой). Это означает, что во время длинных отрезков операций с плавающей запятой только код, целочисленная единица действует как выделенная единица загрузки-хранения (LSU), единственной целью которой является подача данных на FPU.

Такая комбинация FPU + LSU работает хорошо по двум причинам: во-первых, целочисленный код и код с плавающей запятой редко смешиваются, поэтому для производительности не имеет значения, связана ли целочисленная единица с трафиком памяти, связанным с плавающей запятой. Во-вторых, код с плавающей запятой часто требует больших объемов данных, с большим количеством загрузок и сохранений и, следовательно, с высоким уровнем трафика памяти, чтобы держать выделенный LSU занятым.

Когда вы объединяете оба этих фактора с здоровенным 32-килобайтным кэшем L1 601 и его способностью выполнять плавные умножения-сложения за один цикл со скоростью один за тakt, вы получаете силу с плавающей запятой, с которой нужно считаться в терминах 1994 года.

[Исполнительный отдел филиала](#)

Блок ветвлений (BU) процессора 601 работает в сочетании со сборщиком инструкций и очередью инструкций, чтобы направлять переднюю часть процессора через поток кода, выполняя инструкции ветвлений и прогнозируя ветвления.

Что касается последней функции, BU 601 использует простой статический предсказатель ветвлений для предсказания условных ветвлений. Я немного подробнее расскажу о прогнозировании ветвлений и спекулятивном выполнении при рассмотрении 603e в разделе «PowerPC 603 и 603e» на стр. 118.

[Блок секвенсора](#)

601 содержит своеобразный переключок IBM RSC, называемый блоком секвенсора. Блок секвенсора, который, признаюсь, для меня немного загадка, представляет собой небольшой процессор типа CISC с собственным 18-битным набором инструкций, 32-словным ОЗУ, ПЗУ с микрокодом, регистровым файлом и исполнительным механизмом. блок, все из которых встроены в 601. Его цель - выполнить некоторые устаревшие инструкции, характерные для более старого RSC; заботиться о домашних делах, таких как функции самопроверки, сброса и инициализации; и для обработки исключений, прерываний и ошибок.

Включение блока секвенсора в 601 — вполне очевидный результат.

о нехватке времени, с которой столкнулась команда 601 при выводе на рынок первого чипа PowerPC; IBM признала это в своем официальном документе 601. Команда начала с IBM RSC в качестве основы и начала переделывать его для реализации PowerPC ISA. Вместо того, чтобы выбросить секвенсор, компонент, который играл важную роль в функционировании оригинального RSC, IBM просто уменьшила его размер и функциональность для использования в 601.

У меня нет точных цифр, но я думаю, что можно с уверенностью сказать, что этот встроенный подпроцессор занимал приличное количество места на кристалле 601 и что команда разработчиков выбросила бы его, если бы у нее было больше времени. Последующие процессоры PowerPC, которым не нужно было беспокоиться о устаревшей поддержке RSC, реализовали все (не связанные с RSC) функции секвенсора 601, распределив их по другим функциональным блокам.

[Еще раз о задержке и пропускной способности](#)

В суперскалярных процессорах, таких как 601 и его более современные аналоги, разные инструкции проходят через процессор разное количество циклов. Разные исполнительные блоки часто имеют разную глубину конвейера, и даже в одном исполнительном блоке разные инструкции иногда занимают разное количество циклов. Что касается последнего случая, одна инструкция может проходить через ALU дольше, чем другая инструкция, либо потому, что инструкция имеет обязательную остановку на определенном этапе, либо потому, что конкретный подблок, который обрабатывает инструкцию, имеет более длинный конвейер, чем другие подблоки, которые вместе составляют ALU. В этом случае нам больше не имеет смысла упрощенно трактовать задержку команд как свойство процессора в целом. Скорее, задержка инструкции на самом деле является вопросом индивидуальной инструкции, поэтому наше обсуждение будет отражать это с этого момента.

Ранее мы определили задержку инструкции как минимальное количество циклов, которое инструкция должна пройти в фазе выполнения. Вот задержки некоторых часто используемых инструкций PowerPC на PowerPC G4, процессоре, который мы обсудим в разделе «PowerPC 7400 (он же G4)» на стр. 133:

Мнемонические циклы для выполнения	
деление	1
а также	1
стр	1
раздел	19
мул	6

Обратите внимание, что для выполнения большинства инструкций требуется только один цикл, а для некоторых, таких как деление и умножение, требуется больше. Целочисленное деление полного слова, например, занимает 19 тактов, а 32-битное умножение занимает 6 тактов. Это означает, что инструкция деления находится в целочисленном конвейере IU1 в течение 19 тактов, и в течение этого времени никакая другая инструкция не может выполняться в IU1.

Теперь давайте посмотрим на задержки инструкций с плавающей запятой для G4:

Мнемонические циклы для выполнения	
пересечение	1-1-1
приходит	1-1-1
fdiv	32
фмадд	1-1-1
фмуль	1-1-1
фсуб	1-1-1

Эти задержки перечислены немного иначе, чем целочисленные задержки команд. Числа, разделенные тире, показывают, сколько времени инструкция проводит на каждом из трех этапов конвейера FPU. Большинство перечисленных инструкций занимают один цикл на каждом этапе, всего три цикла в конвейере FPU G4, поэтому, если программа использует только эти инструкции, FPU может запускать и заканчивать одну инструкцию в каждом цикле.

Некоторые инструкции, такие как деление с плавающей запятой, имеют только одно число в столбце задержки. Это связано с тем, что при выполнении fdiv связывает весь конвейер операций с плавающей запятой. Пока fdiv отрабатывает свои 32 цикла в FPU, никакие другие инструкции не могут выполняться вместе с ним в конвейере с плавающей запятой. Это означает, что любые инструкции с плавающей запятой, которые идут сразу после fdiv в потоке кода, должны ждать в очереди инструкций, потому что они не могут быть отправлены во время выполнения fdiv .

Резюме: 601 в историческом контексте

601-й мог потратить тонну транзисторов (по крайней мере, тонну для того времени) на кэш-память 32 КБ, потому что его интерфейс был намного проще, чем у его аналога x86, Intel Pentium. В то время это было значительным преимуществом перед использованием RISC. Чип дебютировал в PowerMac 6100 и получил хорошие отзывы, что вывело Apple на первое место по производительности по сравнению с конкурентами x86. Модель 601 окончательно утвердила культ Apple как производителя компьютеров высокого класса.

Тем не менее, модель 601 оставляла место для совершенствования. Секвенсор, унаследованный от своего предка, майнфрейма, занимал ценное место на кристалле, которое можно было бы использовать с большей пользой. Если бы у него было немного больше времени, чтобы настроить его, 601 мог бы быть ближе к идеалу. Но для достижения почти совершенства нужно было дождаться двух ударов 603e и 604.

PowerPC 603 и 603e

В то время как одна команда наносила последние штрихи на 601-й, другая команда в Центре дизайна IBM в Сомерсете в Остине уже начала работу над преемником 601-го — 603-м. эволюционный сдвиг, чем это был совершенено другой процессор. В Табл. 6-2 приведены характеристики PowerPC 603 и 603e.

Табл. 6-2: Характеристики PowerPC 603 и 603e

	PowerPC 603 Виталс	PowerPC 603e Основные характеристики
Дата введение	1 мая 1995 г.	16 октября 1995 г.
Процесс	0,50 мкм	0,50 мкм
Количество транзисторов	1,6 миллиона	2,6 миллиона
размер штампа	81 мм ²	98 мм ²
Тактовая частота при введении 75 МГц		100 МГц
Размер кэша L1	16 КБ разделенного L1	32 КБ разделенного L1
Впервые появился в	Macintosh Performa 5200CD Macintosh Performa 6300CD	

Модель 603 была разработана для работы с очень небольшим энергопотреблением, потому что Apple нуждался в чипе для своей линейки портативных компьютеров PowerBook. В результате у процессора было очень хорошее соотношение производительности на ватт на родном коде PowerPC, и фактически он был в состоянии соответствовать 601 по тактовой производительности, даже несмотря на то, что в нем было примерно вдвое меньше транзисторов, чем в старом процессоре. Но меньший 16 КБ разделенный кэш L1 603 означал, что он довольно плохо эмулировал устаревший код 68 КБ, который составлял большую часть ОС Apple и базы приложений.

В результате 603 был отнесен к самому низкому уровню продуктовой линейки Apple (Performas, начиная с 6200, и многофункциональные устройства, разработанные для рынка образования, начиная с 5200), до измененной версии (был выпущен 603e) с увеличенным разделенным кешем на 32 КБ. 603e работал лучше на эмулированном коде 68K, поэтому он нашел широкое применение в линейке PowerBook.

В этом разделе будет кратко рассмотрена микроархитектура 603e, показанная на рис. 6-2, поскольку именно эта версия 603 получила наиболее широкое распространение.

ПРИМЕЧАНИЕ. Модель 604 также была выпущена одновременно с оригинальной моделью 603. Модель 604, предназначенная для высококлассных продуктов Apple, точно так же, как 603e предназначалась для недорогих продуктов, была еще одним совершенно новым дизайном. Мы рассмотрим 604 в разделе «PowerPC 604» на стр. 123.

Задняя часть 603e

Как и 601, 603e оснащен классическим четырехступенчатым конвейером RISC. Но в отличие от 601, который может декодировать и отправлять до трех инструкций за цикл на любой из своих исполнительных блоков, включая блок ответвлений, у 603e есть одно важное ограничение, ограничивающее использование пропускной способности диспетчеризации, равной трем инструкциям за цикл.

На 603e и на всех производных от него процессорах (750 и 7400/7410) ветки, которые не свернуты или не проваливаются, отправляются из очереди инструкций в блок ветвления пошине диспетчеризации, которая не т подключен к любому другому исполнительному устройству. Таким образом, инструкции ветвления не занимают никакой доступной пропускной способности диспетчеризации, которая питает основную часть серверной части. 603e и его производные могут отправлять одну команду ветвления за цикл в модуль ветвления по этой конкретнойшине.

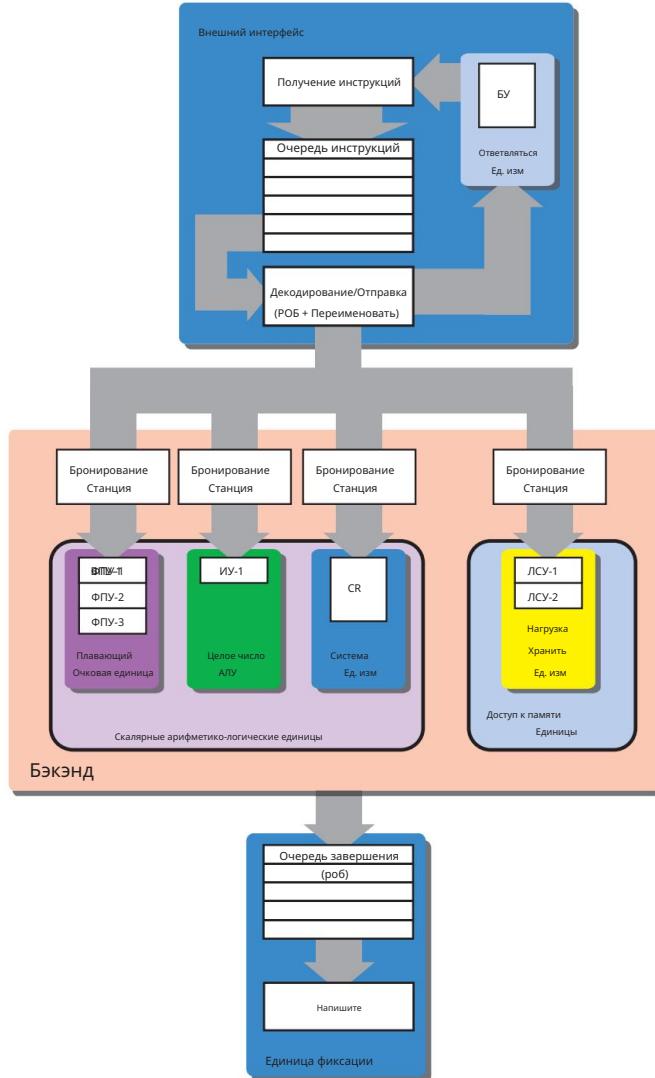


Рисунок 6-2: Микроархитектура PowerPC 603e

Инструкции без ветвления могут отправлять на серверную часть со скоростью до двух инструкций за цикл, что означает, что максимальная скорость отправки 603 составляет три инструкции за цикл (две инструкции без ветвления + одна ветвь). Однако, поскольку за цикл могут выполняться две инструкции без ветвления, инструкции ветвления часто игнорируются при обсуждении скорости отправки 603 и его последователей. Поэтому часто говорят, что эти процессоры имеют скорость диспетчериизации до двух инструкций за такт, даже если скорость диспетчериизации технически невелика для трех инструкций за цикл.

Логика диспетчеризации 603е берет максимум две инструкции без ветвления за цикл из нижней части очереди инструкций и передает их на сервер, где они выполняются одним из пяти исполнительных блоков:

- z Целая единица
- z Модуль с плавающей запятой
- z Филиал
- z Блок загрузки-накопителя
- z Системный блок

Обратите внимание, что этот список содержит на два блока больше, чем аналогичный список для 601: блок загрузки-сохранения (LSU) и системный блок. Блок загрузки-сохранения 603е берет на себя всю работу по вычислению адресов, которую старый 601 возлагал на его единственное целочисленное ALU. Поскольку 603е имеет выделенный LSU для выполнения вычислений адресов и выполнения операций сохранения данных, его целочисленный блок освобождается от необходимости обрабатывать трафик памяти и, следовательно, может сосредоточиться исключительно на целочисленной арифметике. Это помогает повысить производительность 603е при работе с целочисленным кодом.

Выделенный системный блок 603е также берет на себя некоторые функции целочисленного блока 601, поскольку он обрабатывает обновления регистра условий PowerPC. Подробнее о регистре условий мы поговорим в главе 10, так что не волнуйтесь, если вы не знаете, что это такое. Системный блок 603е также содержит ограниченный целочисленный сумматор, который может снять часть нагрузки с целочисленного ALU, выполняя определенные типы сложения. (В оригинальном системном блоке 603 эта функция отсутствовала.)

Основной конвейер вычислений с плавающей запятой в 603е отличается от конвейера в 601 тем, что у него на один этап выполнения больше и на один этап декодирования меньше. Большинство инструкций с плавающей запятой имеют задержку в три такта (и пропускную способность в один цикл) на 603е по сравнению с задержкой в два цикла на 601. Эта конструкция с задержкой в три цикла и пропускной способностью в один цикл была бы неплохой для всех, если бы не одна серьезная проблема: на самом быстром уровне FPU 603е может выполнять только три инструкции каждые четыре цикла. Другими словами, после каждой третьей одноцикловой инструкции с плавающей запятой обязательно появляется пузырь конвейера. Я не буду вдаваться в причины этого, но FPU 603е подвергся нетривиальному удару по производительности для этой схемы с тремя инструкциями и четырьмя циклами.

Другим, возможно, более серьезным недостатком FPU 603е является то, что он не полностью конвейеризирован для операций с несколькими операциями. Умножения с двойной точностью, включая fmadd с двойной точностью, занимают два цикла на этапе выполнения, а это означает, что FPU процессора 603е может выполнять только одно умножение с двойной точностью каждые два цикла.

Однако модуль 603е с плавающей запятой не так уж и плох. Он по-прежнему имеет стандартную для PPC способность выполнять операции fmadd с одинарной точностью, с задержкой в четыре цикла и пропускной способностью в один цикл. Эта способность к быстрому формированию помогла архитектуре сохранить большую часть своей полезности для DSP, научных и медийных приложений, несмотря на вышеупомянутые недостатки.

[Интерфейс пользователя 603e, окно инструкций и предсказание переходов](#)

До двух инструкций за цикл может быть загружено в очередь команд 603 с шестью записями. Оттуда максимум две инструкции за цикл (на одну меньше, чем у 601) могут быть отправлены из двух нижних записей в IQ на станции резервирования в задней части 603e.

Мало того, что 603e отправляет на серверную часть на одну инструкцию меньше за такт, чем 601, но его общий подход к суперскалярному и неупорядоченному выполнению отличается от подхода 601 и другим способом. В 603e используется выделенный модуль фиксации, который содержит очередь завершения с пятью элементами (аналог ROB P6) для отслеживания порядка выполнения инструкций в программе. Когда инструкции выполняются не по порядку, блок фиксации обращается к информации, хранящейся в очереди завершения, и возвращает инструкции в программный порядок перед их фиксацией.

Если использовать термин, который фигурировал в нашем обсуждении Pentium, можно сказать, что 603 — это первый процессор PowerPC с динамическим планированием через полнофункциональное окно команд, в комплекте с ROB и станциями резервирования.

Мы поговорим подробнее о концепции окна инструкций и о составляющих его структурах (ROB и станции резервирования) в следующем разделе, посвященном модели 604. А пока достаточно сказать, что окно инструкций модели 603 довольно небольшой по сравнению с его преемниками - три из четырех его станций бронирования только с одним входом, а одна - с двойным входом (та, которая прикреплена к блоку загрузки-накопления). Поскольку окно инструкций процессора 603 очень маленькое, ему требуется относительно небольшое количество регистров переименования для временного хранения результатов выполнения перед фиксацией. 603 имеет пять регистров переименования общего назначения, четыре регистра переименования с плавающей запятой и по одному регистру переименования для регистра условий (CR), регистра связи (LR) и регистра счетчика (CTR).

603 и 603e следуют за 601 в их способности выполнять спекулятивное выполнение с помощью простого статического предсказателя перехода. Как и статический предсказатель на 601, предиктор 603e помечает прямые переходы как неиспользованные, а обратные - как выполненные. Этот статический предсказатель ветвления прост и быстр, но его эффективность невелика по сравнению даже со слабо разработанным динамическим предсказателем ветвления. Если пользователям PPC в эпоху 603e/604 требовалось динамическое прогнозирование переходов, им приходилось переходить на 604.

[Резюме: 603 и 603e в историческом контексте](#)

С его звездным соотношением производительности на ватт, 603 был отличным маленьким процессором, и он также мог бы стать хорошим процессором для настольных ПК с низким и средним уровнем, если бы не устаревшая кодовая база Apple 68K. Настройки 603e и больший размер кэш-памяти несколько помогли решить устаревшие проблемы, но обновленный чип по-прежнему играл вторую скрипку в линейке продуктов Apple после более крупного и гораздо более мощного 604.

Однако вы еще не видели последний из 603e. Дизайн 603e лег в основу того, что в конечном итоге стало Motorola PowerPC 7400, также известным как G4, о котором мы расскажем в разделе «PowerPC 7400 (он же G4)» на стр. 133.

PowerPC 604

В то же время, когда модель 603 продвигалась к рынку, в разработке находилась и модель 604. 604 должен был стать высокопроизводительным процессором Apple для настольных ПК, поэтому его мощность и бюджеты на транзисторы были намного выше, чем у 603. Рисунок 6-3) показывает несколько очевидных отличий от младшего брата. Например, во внешнем интерфейсе длина очереди инструкций увеличена на две записи. В серверной части добавлены еще две целочисленные единицы, а логическая единица CR удалена. Эти изменения отражают некоторые важные различия в общем подходе к модели 604, которые вскоре будут рассмотрены более подробно.

Табл. 6-3: Характеристики PowerPC 604 и 604e

	PowerPC 604	PowerPC 604e
Дата введение	1 мая 1995 г.	19 июля 1996 г.
Процесс	0,50 мкм	0,35 мкм
Количество транзисторов	3,6 миллиона	5,1 миллиона
Размер штампа	197 мм ²	148 мм ²
Тактовая частота при введении 120 МГц		180–200 МГц
Размер кэша L1	32 КБ, разделенный L1	64 КБ, разделенный L1
Впервые появился в	PowerMac 9500/120	PowerComputing PowerTower Pro 200 (PowerMac 9500/180, 7 августа 1996 г.)

Конвейер 604 и серверная часть

Конвейер 604 глубже, чем у 601 и 603, и состоит из следующих шести этапов:

Четыре фазы стандартного конвейера RISC Шесть этапов конвейера 604	
Принести	1. Получить
Декодирование/отправка	2. Расшифровать
Выполнять	3. Диспетчеризация (РОВ и переименование) 4. Выполнить
Обратная запись	5. Завершить 6. Обратная запись

В модели 604 стандартная фаза декодирования/отправки RISC разделена на две стадии, как и фаза обратной записи. Я объясню, как работают эти два новых этапа конвейера, в разделе, посвященном окну инструкций, а пока все, что вам нужно понять, это то, что этот удлиненный конвейер позволяет процессору 604 достигать более высоких тактовых частот, чем его предшественники. Поскольку каждый этап конвейера проще, для его завершения требуется меньше времени, а это означает, что время тактового цикла ЦП может быть сокращено.

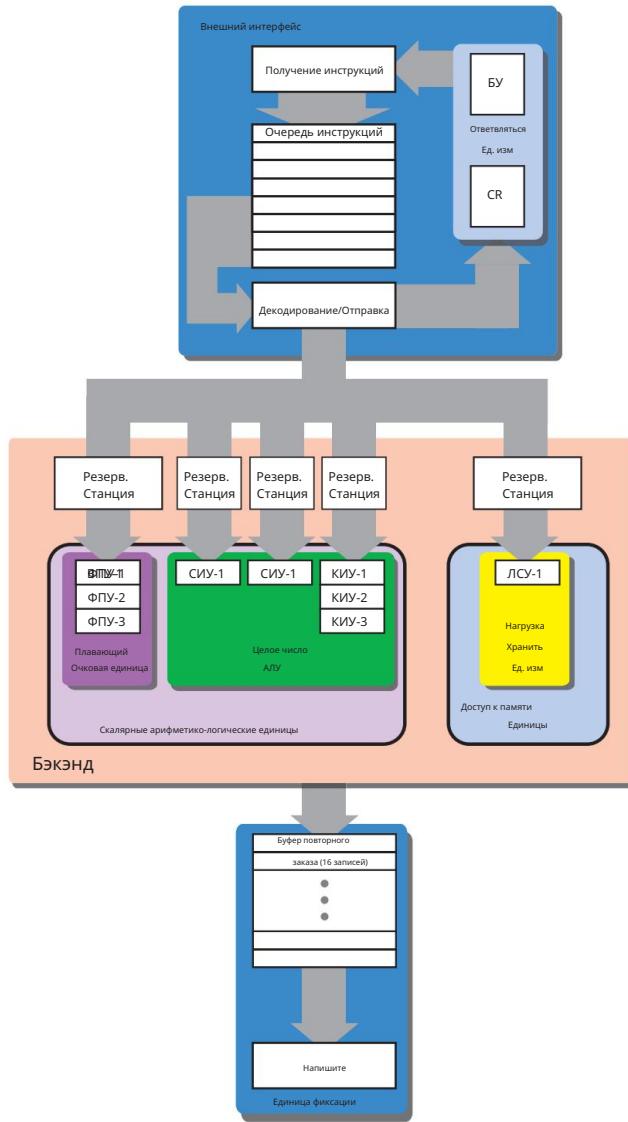


Рисунок 6-3: Микроархитектура PowerPC 604

Помимо более длинного конвейера, еще одним фактором, который действительно отличает 604 от других обсуждавшихся до сих пор конструкций PPC серии 600, является его более широкая задняя часть. 604 может выполнять до шести инструкций за такт в следующих шести исполнительных блоках:

- z Модуль ветвления (BU)/модуль регистра условий (CRU)
- z Модуль загрузки-сохранения (LSU)
- z Модуль с плавающей запятой (FPU)

[z Три целых единицы \(ME\)](#)

[z Две простые целочисленные единицы \(SIU\)](#)

[z Одна комплексная целочисленная единица \(CIU\)](#)

В отличие от других процессоров серии 600, 604 имеет несколько целочисленных блоков. Это разделение труда, при котором несколько быстрых целочисленных блоков выполняют простые целочисленные инструкции, а один более медленный целочисленный блок выполняет сложные целочисленные инструкции, будет более подробно обсуждаться в главе 8. Любая целочисленная инструкция, для выполнения которой требуется только один цикл, может пройти через один из двух CIU. С другой стороны, целочисленные инструкции, выполнение которых занимает несколько циклов, например целочисленное деление, должны проходить через более медленный CIU.

Как и 603e, 604 имеет переименование регистров, метод, который облегчается файлом переименования регистра с 12 записями, прикрепленным к файлу регистра общего назначения с 32 записями. Эти буферы переименования предоставляют дополнительным блокам 604 больше возможностей для предотвращения ложных зависимостей и остановок, связанных с регистрами.

Блок вычислений с плавающей запятой 604 выполняет большинство операций с одинарной и двойной точностью с задержкой в три цикла, как и 603e. Однако, в отличие от 603e, модуль 604 с плавающей запятой полностью конвейеризирован для умножения с двойной точностью. Деление с плавающей запятой и две другие инструкции занимают от 18 до 33 циклов на 604, как и на 603e. Наконец, файл регистра с плавающей запятой 604 с 32 записями присоединен к буферу регистра переименования с 8 записями с плавающей запятой.

Блок загрузки-накопления (LSU) 604 также аналогичен блоку 603e. Как и LSU 603e, он содержит сумматор для вычисления адресов и обрабатывает весь трафик загрузки-сохранения, но, в отличие от 603e, он подключен к более глубоким очередям загрузки и сохранения и обеспечивает немного большую гибкость для оптимального переупорядочивания операций с памятью.

Модуль ветвления 604 также имеет схему динамического предсказания ветвления, которая является значительным улучшением по сравнению со статическим предсказателем ветвления 603e. 604 имеет большую таблицу истории ветвлений (BHT) с 512 записями, по два бита на запись для отслеживания ветвей, в сочетании с кешем целевого адреса ветвления (BTAC) с 64 записями, который является эквивалентом BTB Pentium.

Как всегда, чем больше транзисторов вы тратите на прогнозирование ветвления, тем выше производительность, поэтому более продвинутый блок ветвления в 604 немного помогает. Тем не менее, в случае неправильного прогноза более длинный конвейер 604 должен платить более высокую цену, чем его предшественники с более коротким конвейером с точки зрения производительности. Конечно, большая потеря производительности, связанная с неправильным предсказанием, также является причиной того, что 604 должен тратить эти дополнительные ресурсы на предсказание переходов.

Обратите внимание, что в списке дополнительных устройств на стр. 124 отсутствует модуль, присутствующий в 603e: системный модуль. Системный блок 603e обрабатывал обновления в регистре условий PowerPC, функция, которая выполнялась целочисленным дополнительным блоком в более старых 601. 604 перекладывает ответственность за работу с регистром условий на ветвящийся блок. Таким образом, модуль ответвления 604 содержит отдельный дополнительный модуль, который обрабатывает все логические операции, связанные с регистром условий PowerPC. Этот блок регистра условий (CRU) совместно использует

диспетчерская шина и некоторые другие ресурсы с исполнительным блоком ветвления, так что это не полностью независимый исполнительный блок, такой как системный блок 603e. Как эта комбинация BU/CRU влияет на производительность? Это, вероятно, не имеет большого влияния, но какое бы влияние оно ни оказалось, оно достаточно существенно для того, чтобы непосредственный преемник 604 — 604e — добавил независимый исполнительный блок к серверной части для логических операций CR.

Внешний вид модели 604 и окно инструкций

Передняя часть 604-го и окно инструкций выглядят как комбинация лучших характеристики моделей 601 и 603e. Как и у 601, у 604 очередь инструкций имеет глубину восемь записей. Инструкции извлекаются из кэша L1 в очередь инструкций, где они декодируются перед отправкой на серверную часть. Ветви, которые могут быть свернуты, свернуты, и логика диспетчеризации 604 может отправлять до четырех инструкций за цикл (по сравнению с двумя на 603e и тремя на 601) из четырех нижних записей очереди инструкций на серверные исполнительные устройства.

На этапе отправки 604 регистры переименования и запись буфера переупорядочивания назначаются каждой инструкции отправки. Когда инструкция готова к отправке, она отправляется либо непосредственно в исполнительный модуль, либо на станцию резервирования исполнительного модуля, в зависимости от того, доступны ли ее операнды во время отправки. Обратите внимание, что 604 может отправить не более одной инструкции каждому исполнительному блоку, и существуют определенные правила, определяющие, когда логика отправки может отправлять инструкцию серверной части. Мы рассмотрим эти правила более подробно чуть позже, а сейчас вам нужно знать об одном из них: инструкция не может быть отправлена, если требуемый исполнительный модуль недоступен.

Фаза выпуска: станции бронирования 604-го

На рис. 6.3 вы, вероятно, заметили, что к каждому исполнительному блоку 604 подключена станция резервирования; это включает в себя каждую станцию резервирования (не показана) для блоков выполнения ответвления и регистра условий, которые составляют блок ответвлений. Станции резервирования 604 относительно небольшие, с двумя входами (станцией резервирования CIU является однократная), работающие в порядке поступления (FIFO), но они составляют основу окна команд 604, потому что они позволяют выполнять инструкции, назначается одному исполнительному устройству для выполнения не в порядке программы по отношению к инструкциям, назначенным другим исполнительным устройствам.

Это работает следующим образом: этап отправки отправляет инструкции на станции резервирования (т. е. этап выдачи) в порядке программы, и, за одним важным исключением (описанным в следующем абзаце), инструкции проходят через соответствующие станции резервирования по порядку. Инструкция поступает в верхнюю часть станции резервирования, и по мере выдачи инструкций, предшествующих ей, она перемещается вниз по очереди, пока в конце концов не выйдет через нижнюю часть (т. е. не выдаст).

Следовательно, мы можем сказать, что каждая инструкция выдается по порядку относительно других инструкций на той же станции резервирования. Однако различные станции резервирования могут выдавать инструкции в разное время, в результате чего инструкции выдаются не по порядку с точки зрения общего потока программы.

Простые целочисленные блоки функционируют немного иначе, чем описано ранее, потому что они позволяют командам выдаваться из их станций резервирования с двумя входами не по порядку по отношению к другим инструкциям в их собственном исполнительном блоке.

Таким образом, в отличие от других типов инструкций, описанных ранее, целочисленные инструкции могут перемещаться по соответствующим станциям резервирования и конвейерам вне порядка программы не только в отношении общего потока программы, но и в отношении других инструкций в своей собственной станции резервирования.

Станции бронирования в 604 и его архитектурных преемниках существуют.

Чтобы инструкции, которым не хватает входных данных операнда, но которые в остальном готовы к отправке, не связывали очередь инструкций. Если инструкция соответствует всем остальным требованиям отправки (см. «Четыре правила отправки инструкций») и если назначенный ей исполнительный блок доступен, но он просто еще не имеет доступа к той части потока данных, которая ему нужна, он отправляет на станцию резервирования соответствующей исполнительной единицы, чтобы инструкции, стоящие за ним в очереди инструкций, могли двигаться вверх и быть отправлены.

Небольшой размер станций резервирования 604 по сравнению с аналогичными структурами на Р6 связан с тем, что конвейер 604 относительно короткий.

Задержки конвейера не так разрушительны для производительности на машине с 6-ступенчатым конвейером, как на машине с 12-ступенчатым конвейером, поэтому 604 не нужно такое большое окно инструкций, как его супер-конвейер. аналоги.

[Четыре правила отправки инструкций](#)

Вот четыре наиболее важных правила, регулирующих отправку инструкций на 604:

[Правило упорядоченной отправки](#)

Прежде чем инструкция может быть отправлена, все инструкции, предшествующие этой инструкции, должны быть отправлены. Другими словами, инструкции отправляются из очереди инструкций в программном порядке. Только после того, как инструкции поступят на станции резервирования, где они могут выдаваться исполнительным блокам не по порядку, первоначальный порядок программы нарушается.

[Правило доступности буфера задач/единицы выполнения](#)

Прежде чем логика диспетчеризации сможет отправить инструкцию на станцию резервирования исполнительного устройства, эта станция резервирования должна иметь доступную запись. Если инструкции не нужно отправлять на станцию резервирования, потому что ее входы доступны во время отправки, требуемый исполнительный модуль должен иметь доступный слот конвейера, а станция резервирования модуля должна быть пустой (т. инструкции, ожидающие выполнения) до того, как инструкция может быть отправлена исполнительному блоку. (Это правило изменено для PowerPC 7450 — также известного как G4e — и мы рассмотрим эту модификацию в разделе «PowerPC 7400 (также известный как G4)» на стр. 133.)

Правило доступности буфера завершения

Для отправки инструкции в очереди завершения должно быть свободное место, чтобы можно было создать новую запись для инструкции.

Помните, что очередь завершения (или ROB) отслеживает порядок выполнения каждой выполняемой инструкции, поэтому любая инструкция, поступающая в неупорядоченную серверную часть, должна быть сначала зарегистрирована в очереди завершения.

Правило доступности реестра переименования

Должно быть достаточно доступных регистров переименования для временного хранения результатов для каждого регистра, который будет изменять инструкция.

Если отправленная инструкция соответствует требованиям, налагаемым этими правилами, и если она соответствует другим, более специфичным для инструкции правилам отправки, не перечисленным здесь, она может быть отправлена из очереди инструкций на серверную часть.

Все рассмотренные в этой главе процессоры PowerPC, имеющие станции резервирования, подчиняются (как минимум) этим четырем правилам диспетчеризации, так что помните об этих правилах, когда мы будем говорить о диспетчеризации инструкций в оставшейся части этой главы. Обратите внимание, что все процессоры, включая 604, имеют дополнительные правила, управляющие отправкой определенных типов инструкций, но эти четыре общих правила отправки являются наиболее важными.

Фаза завершения: буфер повторного заказа модели 604

Как и в микроархитектуре Р6, станции резервирования — не единственные структуры, составляющие окно команд 604. 604 имеет буфер переупорядочивания (ROB) с 16 элементами, который выполняет ту же функцию, что и гораздо более крупный ROB с 40 элементами микроархитектуры Р6.

ROB соответствует более простой очереди завершения на старых процессорах PPC. На этапе отправки не только инструкции отправляются на серверные станции резервирования, но и записи для отправленных инструкций выделяются записью в ROB и набором регистров переименования. На этапе завершения инструкции возвращаются в программном порядке, чтобы их результаты можно было записать обратно в регистровый файл на последующем этапе обратной записи. Стадия завершения соответствует тому, что я назвал фазой завершения жизненного цикла инструкции, а стадия обратной записи соответствует тому, что я назвал фазой фиксации.

ROB 604 намного меньше, чем ROB Р6, по той же причине, по которой у 604 меньше станций резервирования: у 604 гораздо более мелкий конвейер, а это означает, что ему требуется гораздо меньшее окно команд для отслеживания меньшего количества инструкций в полете, чтобы достичь той же производительности.

Компромиссом за отсутствие сложности и меньшую глубину конвейера является более низкая тактовая частота. 6-этапный процессор 604 дебютировал в мае 1995 г. на частоте 120 МГц, а 12-этапный Pentium Pro дебютировал позже в том же году (ноябрь 1995 г.) на частоте от 150 до 200 МГц.

Резюме: 604 в историческом контексте

С 32-килобайтным разделенным кэшем L1 у 604-й модели был намного больший кэш, чем у его предшественников, который был необходим для поддержания более глубокого конвейера. Кэш большего размера, более высокая скорость отправки и выпуска, более широкая серверная часть и более глубокий конвейер сделали надежного исполнителя RISC, который легко мог идти в ногу со своими конкурентами.

Тем не менее, Pentium Pro не был неудачником, и его производительность хорошо масштабировалась с улучшением технологий производства процессоров. Apple требовалось больше мощности от AIM, чтобы идти в ногу со временем, и они получили больше мощности с небольшой микроархитектурной доработкой, которая стала называться 604e.

PowerPC 604e

604e построен на преимуществах, достигнутых 604, с некоторыми основными изменениями, которые включали удвоение размера кэша L1 (до 32 КБ инструкций / 32 КБ данных) и добавление нового независимого исполнительного блока: блока регистра условий (CRU).

В предыдущих процессорах серии 600 ответственность за обработку логических операций регистра условий была перенесена между различными модулями (целочисленный модуль в 601, системный модуль в 603/603e и модуль ветвлений в 604). Теперь, с появлением модели 604e, эти операции получили собственное исполнительное устройство. 604e имел функциональный блок в своей серверной части, который был предназначен для обработки логических операций регистра условий, а это означало, что эти нередкие операции не связывали другие исполнительные блоки, такие как целочисленный блок или блок ветвлений, которые выполняли более серьезную работу. сделать.

Блок ответвлений 604e, теперь, когда он был свободен от обработки логических операций CR, получил несколько расширенных возможностей, которые я не буду здесь подробно описывать. Кэши 604e, в дополнение к увеличению, также получили дополнительные буферы обратного копирования и несколько других улучшений.

В конечном итоге 604e смог масштабироваться до 350 МГц после перехода с 0,35 на 0,25 микрон производственного процесса, что сделало его успешной частью многообещающей линейки мультимедийных рабочих станций Apple RISC.

PowerPC 750 (он же G3)

PowerPC 750, известный пользователям Apple как G3, представляет собой конструкцию, в значительной степени основанную на 603/603e. Его четырехэтапный конвейер такой же, как у 603/603e, и многие особенности его внешнего интерфейса и внутреннего интерфейса будут вам знакомы из нашего обсуждения более старого процессора. Тем не менее, 750 содержит несколько очень важных улучшений по сравнению с 603e, которые делают его быстрее, чем даже 604e, как вы можете видеть в Таблице 6-4.

Табл. 6-4: Характеристики PowerPC 750

Дата введения	сентябрь 1997 г.
Процесс	0,25 мкм
Количество транзисторов	6,35 миллиона
Размер штампа	67 мм ²
Тактовая частота при введении 200–300 МГц	
Размеры кэша	64 КБ разделенного L1, 1 МБ L2
Впервые появился в	Мощность Macintosh G3

Значительное улучшение производительности 750 по сравнению с 603/603e является результатом ряда факторов, не последним из которых являются усовершенствования, которые IBM внесла в 750 в целочисленных возможностях и возможностях с плавающей запятой.

Беглый взгляд на компоновку 750 (см. рис. 6-4) показывает, что его задняя часть шире, чем у 603. Точнее, если у 603 есть один целочисленный блок, то у 750 их два — простой целочисленный блок (SIU).) и комплексной целочисленной единицы (CIU). Сложный целочисленный блок 750 обрабатывает все целочисленные инструкции, в то время как простой целочисленный блок обрабатывает все целочисленные инструкции, кроме умножения и деления. Большинство целочисленных инструкций, которые выполняются в SIU, являются инструкциями с одним циклом.

Как и в 603 (и 604), блок операций с плавающей запятой в 750 может выполнять все операции с плавающей запятой одинарной точности, включая умножение, с задержкой в три цикла. И, как и в случае с 603, ранние версии 750 должны были вставлять пузырек конвейера после каждой третьей инструкции с плавающей запятой в своем конвейере; это исправлено в более поздних версиях IBM 750. Операции с плавающей запятой двойной точности, за исключением операций, включающих умножение, также занимают три цикла на 750. Операции умножения двойной точности и умножения-сложения занимают четыре цикла., потому что у 750 нет полноценного FPU с двойной точностью.

Блок загрузки-хранения и блок системных регистров модели 750 выполняют те же функции, которые описаны в предыдущем разделе для модели 603, поэтому они не заслуживают дальнейшего комментария.

[Внешний интерфейс модели 750, окно инструкций и инструкция ответвления](#)

750 извлекает до четырех инструкций за цикл в свою очередь инструкций с шестью записями и отправляет до двух инструкций без ветвления за цикл из двух нижних записей IQ. Логика отправки следует четырем правилам отправки, описанным ранее, при принятии решения о том, когда инструкция имеет право на отправку, и каждой отправленной инструкции назначается запись в ROB с шестью записями 750 (сравните ROB с пятью записями 603).

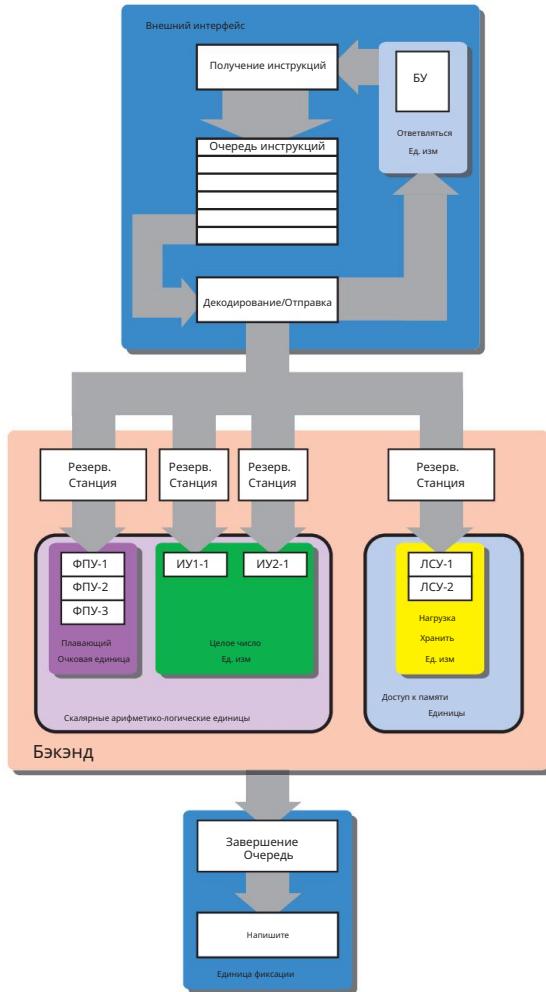


Рисунок 6-4: Микроархитектура PowerPC 750

Как и в случае 603 и 604, вновь отправленные инструкции поступают на станцию резервирования исполнительного блока, в который они были отправлены, где они ждут, пока их операнды станут доступными, чтобы их можно было выдать. Конфигурация станции резервирования модели 750 аналогична конфигурации станции резервирования модели 603 в том смысле, что, за исключением станции резервирования с двумя входами, присоединенной к LSU модели 750, все исполнительные устройства имеют станции резервирования с одним входом. Как и в модели 603, в 750-м филиале нет станции резервирования.

Поскольку окно команд 750 настолько маленькое, у него вдвое меньше регистров переименования, чем у 604. Тем не менее, шесть регистров переименования общего назначения и шесть регистров переименования с плавающей запятой у 750 по-прежнему опережают 603 по количеству регистров переименования (пять GPR и четыре FPR). Как и 603, 750 имеет по одному регистру переименования для CR, LR и CTR.

Вы могли бы подумать, что у 750-го места для бронирования меньше и короче.

РОВ поставил бы его в невыгодное положение по сравнению с 604, у которого окно команд больше. Но конвейер 750 короче, чем у 604, поэтому ему требуется меньше буферов для отслеживания меньшего количества инструкций в процессе выполнения. Что еще более важно, у 750 есть один очень хитрый трюк в рукаве, который он использует, чтобы не заполнять свой конвейер.

Напомним, что стандартные схемы динамического прогнозирования ветвлений обычно используют таблицу истории ветвлений (BHT) в сочетании с целевым буфером ветвлений (BTB), чтобы спекулировать на результате инструкций ветвления и перенаправлять внешний интерфейс процессора в другую точку в кодовом потоке на основе на этом домысле. BHT хранит информацию о прошлом поведении (использовано или не выполнено) самых последних выполненных инструкций ветвления, чтобы процессор мог определить, следует ли ему выполнять эти ветвления, если он встретит их снова. Целевые адреса недавно выполненных ветвей хранятся в BTB, поэтому, когда аппаратное обеспечение предсказания ветвлений решает спекулятивно взять ветвь, оно имеет немедленный доступ к целевому адресу этой ветви без необходимости его пересчета. Целевой адрес спекулятивно принятой ветви загружается из BTB в регистр инструкций, так что в следующем цикле выборки процессор может начать выборку и спекулятивное выполнение инструкций с целевого адреса.

Модель 750 очень умно улучшает эту стандартную схему. Вместо того, чтобы хранить только целевые адреса недавно выполненных ветвей в BTB, в кэше целевых инструкций ветвления (BTIC) 750 с 64 записями хранится инструкция, расположенная по целевому адресу ветвления. Когда блок прогнозирования ветвления 750 проверяет BHT с 512 записями и решает спекулятивно выбрать переход, ему не нужно обращаться к хранилищу кода, чтобы получить первую инструкцию с целевого адреса этого перехода. Вместо этого BPU загружает целевую инструкцию перехода непосредственно из BTIC в очередь инструкций, что означает, что процессору не нужно ждать, пока логика выборки выйдет и извлечет целевую инструкцию из хранилища кода. Эта схема экономит ценные циклы и помогает не допустить появления пузырей, убивающих производительность, в конвейере 750.

[Резюме: PowerPC 750 в историческом контексте](#)

Несмотря на короткий конвейер и маленькое окно с инструкциями, 750-й был весьма хорош. Ему удалось превзойти 604 частично из-за выделенного интерфейса кэш-памяти L2 на задней стороне, который позволил разгрузить трафик L2 с передней шины. Он был настолько успешным, что от производной 604 отказались в пользу создания только 750. 750 и его непосредственные преемники, все из которых вышли под названием G3, в конечном итоге нашли широкое применение как в качестве встроенных устройств, так и во всей линейке продуктов Apple., от портативных компьютеров до рабочих станций.

Однако G3 не хватало одной важной функции, которая отличала его от конкурентов x86: возможности векторных вычислений. В то время как сопоставимые процессоры ПК поддерживали SIMD в виде вектора Intel и AMD.

расширения набора инструкций x86, G3 застрял в мире скалярных вычислений. Поэтому, когда Motorola решила превратить G3 в еще более функциональный чип для встроенных и мультимедийных рабочих станций, этот недостаток был первым, что она устранила.

PowerPC 7400 (он же G4)

Motorola MPC7400 (он же G4) был разработан как центр обработки мультимедиа для настольных и портативных компьютеров. Apple Computer использовала 7400 в качестве ЦП в первой версии своей линейки рабочих станций G4, и этот процессор позже был заменен версией с меньшим энергопотреблением — 7410 — до того, как был представлен 7450 (он же G4+ или G4e). Сегодня преемники 7400/7410 получили широкое распространение в качестве встроенных процессоров, а это означает, что они используются в маршрутизаторах и других устройствах, отличных от ПК, которым нужен микропроцессор с низким энергопотреблением и мощными возможностями DSP. В Табл. 6-5 перечислены функции PowerPC 7400.

Табл. 6-5: Характеристики PowerPC 7400

Дата введения	сентябрь 1999 г.
Процесс	0,20 мкм
Количество транзисторов	10,5 миллионов
Размер штампа	83 мм ²
Тактовая частота при введении 400–600 МГц	
Размеры кэша	64 КБ разделенного L1, 2 МБ L2 поддерживается с помощью встроенных тегов
Впервые появился в	Мощность Macintosh G4

На рис. 6-5 показана микроархитектура PowerPC 7400.

За исключением добавления возможностей SIMD, которые мы обсудим в следующей главе, G4 по сути такой же, как 750. В техническом обзоре G4 от Motorola говорится о G4 по сравнению с 750:

Философия дизайна MPC7410 (и MPC7400) состоит в том, чтобы изменить базовый MPC750 только там, где это необходимо для получения убедительной мультимедийной и многопроцессорной производительности. Ядро MPC7410 по существу такое же, как и MPC750, за исключением того, что в то время как MPC750 имеет очередь завершения из 6 записей и более медленную производительность при выполнении некоторых операций двойной точности с плавающей запятой, MPC7410 имеет очередь завершения из 8 записей и полную двойную прецизионный ФПУ. MPC7410 также добавляет набор инструкций AltiVec, имеет новую подсистему памяти и может взаимодействовать с улучшенной шиной MPX.

— Техническое резюме микропроцессора MPC7410 RISC, раздел 3.11.

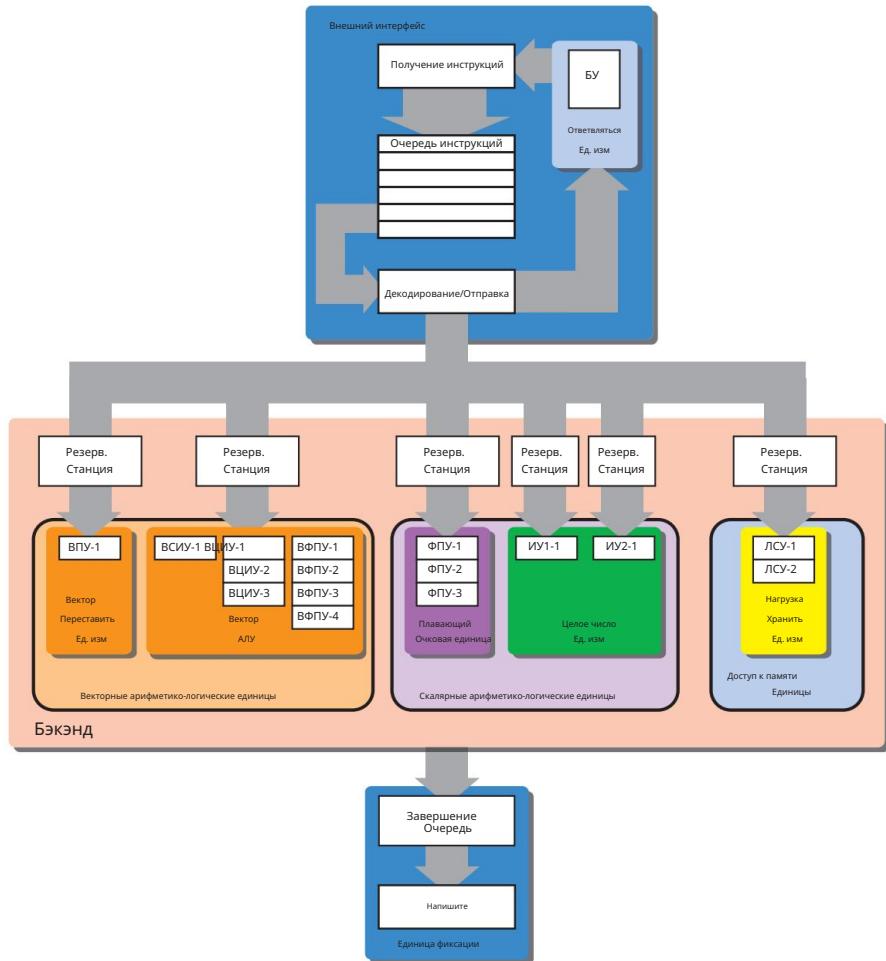


Рисунок 6-5: Микроархитектура PowerPC 7400

Помимо векторного исполнительного блока, самое важное различие в задней части двух блоков заключается в улучшенном FPU G4. FPU G4 — это полноценный FPU двойной точности, выполняющий операции с плавающей запятой одинарной и двойной точности, включая умножение и умножение-сложение, за три полностью конвейерных цикла.

Что касается окна команд, G4 имеет то же количество и конфигурацию станций резервирования, что и 750. (Обратите внимание, что два векторных исполнительных блока G4, которых не было в 750, имеют по одной станции резервирования с одним входом.) Единственное отличие состоит в том, что очередь инструкций G4 была увеличена до восьми записей по сравнению с исходными шестью в 750, чтобы уменьшить узкие места в диспетчериизации.

Векторный блок G4

В конце 1990-х Apple, Motorola и IBM совместно разработали набор расширений SIMD для набора инструкций PowerPC для использования в серии процессоров PowerPC. Эти расширения SIMD носили разные имена: IBM называла их VMX, а Motorola — AltiVec. В этой книге эти расширения будут обозначаться маркировкой Motorola AltiVec.

Новые инструкции AltiVec, которые я подробно рассмотрю в главе 8, были впервые представлены в G4. G4 выполняет эти инструкции в своем векторном блоке, который состоит из двух векторных исполнительных блоков: векторного АЛУ (VALU) и векторного блока перестановки (VPU). VALU выполняет векторные арифметические и логические операции, а VPU выполняет операции перестановки и сдвига векторов.

Для поддержки инструкций AltiVec, которые могут обрабатывать до 128 бит данных одновременно, в PowerPC ISA были добавлены 32 новых 128-битных векторных регистра. В G4 эти 32 архитектурных регистра сопровождаются 6 векторными регистрами переименования.

Резюме: PowerPC G4 в историческом контексте

Набор инструкций AltiVec G4 стал хитом, и он начал широко использоваться Apple и клиентами встроенных систем Motorola. Но в реализации AltiVec для G4 оставалось еще много возможностей для улучшения. В частности, единственному VALU векторного модуля было поручено обрабатывать все операции с целыми числами и векторами с плавающей запятой. Точно так же, как скалярный код выигрывает от наличия нескольких специализированных скалярных ALU, производительность векторов можно улучшить, разделив нагрузку на векторные вычисления между несколькими специализированными VALU, работающими параллельно. Такое улучшение должно было отложиться до появления преемника G4 — G4e.

Основная проблема с G4 заключалась в том, что его короткий четырехэтапный конвейер сильно ограничивал возможность увеличения его тактовой частоты. В то время как Intel и AMD застряли в гонке за гигагерцы, Motorola G4 довольно долго застряла на отметке в 500 МГц. В результате конкуренты Apple x86 вскоре превзошли ее как по тактовой частоте, так и по производительности, оставив некогда самую мощную линейку RISC-рабочих станций с серьезными проблемами на рынке.

Вывод

В серии 600 линейка PPC прошла путь от новичка до зрелой альтернативы RISC, которая вывела рабочую станцию Apple PowerMac на передний план производительности персональных компьютеров. В то время как у первоначального 601 было несколько проблем с зубьями, линейка была в отличной форме после того, как 603e и 604e вышли на рынок. 603e был превосходным мобильным чипом, который хорошо работал в ноутбуках Apple, и хотя у него была более ограниченная пропускная способность отправки/фиксации инструкций и меньший кэш, чем у 601, ему все же удалось превзойти своего предшественника благодаря более эффективному использованию транзисторов. .

604 удвоил пропускную способность диспетчеризации инструкций и фиксации по сравнению с 603, а также имел более широкую серверную часть и большее окно инструкций, что позволяло серверной части выполнять больше инструкций за такт. Кроме того, его конвейер был расширен, чтобы увеличить количество инструкций за такт и обеспечить лучшее масштабирование тактовой частоты. Конечным результатом стало то, что 604 оказался достаточно мощным чипом для настольных ПК, чтобы PowerMac мог комфортно играть в игре с производительностью.

Тем не менее, важно помнить, что серия 600 царила в то время, когда бюджеты транзисторов были относительно небольшими по сегодняшним меркам, поэтому RISC-характер архитектуры PowerPC давал ей определенное преимущество в стоимости, производительности и энергопотреблении по сравнению с конкурентами x86. Это не означает, что серия 600 всегда лидировала по производительности; это не так. За этот период корона перформанса несколько раз переходила из рук в руки.

В период расцвета серии 600 и на заре эры G3 тот факт, что PowerPC был RISC ISA, был сильным знаком в пользу платформы.

Но по мере того, как Кривые Мура увеличивали количество транзисторов и количество МГц, относительная стоимость поддержки устаревшей архитектуры x86 начала снижаться, а преимущество PowerPC ISA в RISC стало ослабевать. К тому времени, когда 7400 появился на рынке, процессоры x86 от Intel и AMD уже догоняли его по производительности, а к тому времени, когда гонка гигагерц закончилась, флагманская линейка рабочих станций Apple оказалась в беде. Тактовая частота и производительность 7400 слишком долго оставались на прежнем уровне в период, когда Intel и AMD вели ожесточенную конкуренцию цена/производительность.

Временным решением этой проблемы от Apple стало обращение к симметричной многопроцессорной обработке (SMP), чтобы повысить производительность своей линейки настольных компьютеров. (См. главу 12 для более подробного обсуждения SMP.) Предлагая компьютеры, в которых два процессора G4 работали вместе для выполнения кода и обработки данных, Apple надеялась наделить свои компьютеры большей вычислительной мощностью таким образом, чтобы не полагаться на Motorola. увеличить тактовые частоты. Двойной G4 имел неоднозначный успех на рынке, и только после дебюта значительно переработанного PowerPC 7450 (также известного как G4+ или G4e) Apple увидела улучшение производительности своих рабочих станций на процессор. Внедрение G4e в свою линейку рабочих станций позволило Apple восстановить некоторые позиции в гонке со своим основным конкурентом в области ПК — системами на базе Intel Pentium 4.

7

INTEL PENTIUM 4 ПРОТИВ. MOTOROLA G4E: ПОДХОДЫ И КОНСТРУКТИВНАЯ ФИЛОСОФИЯ

Теперь, когда мы рассмотрели не только основы микропроцессоров, но и разработку двух популярных линеек процессоров x86 и PowerPC, у вас есть все необходимое, чтобы сравнить и понять два процессора, которые были одними из самых популярных примеров этих двух линеек. : Intel Pentium 4 и Motorola G4e.

Когда Pentium 4 появился на рынке в ноябре 2000 года, это была первая крупная новая микроархитектура x86 от Intel с момента появления Pentium Pro в 1995 году. В годы, предшествовавшие выпуску Pentium 4, ядро Pentium Pro P6 доминировало на рынке в своих воплощениях как Pentium II и Pentium III, и любой, кто обращал на это внимание, усвоил по крайней мере один важный урок: тактовая частота продается. В Intel определенно обращали на это внимание, и, пока члены команды Уилламетта работали в Хиллсборо, штат Орегон, они держали в уме МГц в первую очередь. Эта исключительная направленность очевидна во всем, начиная с рекламной и технической литературы Intel Pentium 4 и заканчив-

самая последняя деталь конструкции процессора. Как будет показано в этой главе, преемником самой успешной микроархитектуры x86 всех времен была машина, созданная с нуля для достижения заоблачной тактовой частоты.

ПРИМЕЧАНИЕ. Willamette было кодовым названием Intel для Pentium 4, пока проект находился в разработке.

Проекты Intel обычно носят кодовые названия рек в штате Орегон. Многие компании используют кодовые имена, которые следуют определенному соглашению, например Apple использует имена больших кошек для версий OS X.

Motorola представила MPC7450 в январе 2001 года, и Apple быстро приняла его под названием G4. Поскольку модель 7450 представляла собой значительный отход от модели 7400, модель 7450 часто называли G4e или G4+, поэтому в этой главе мы будем называть ее G4e. У нового процессора был немного более глубокий конвейер, что позволяло ему масштабироваться до более высоких тактовых частот, а его внешний и внутренний интерфейсы имели целый ряд улучшений, которые отличали его от оригинального G4. Он также продолжил отличную производительность /

коэффициент энергопотребления своих предшественников. Сочетание этих функций сделало его превосходным чипом для портативных компьютеров, и Apple использовала производные от этой базовой архитектуры под названием G4 в серии инновационных корпусов для настольных ПК и портативных устройств. G4e также обеспечил улучшенную производительность векторных вычислений, что сделало его отличной платформой для DSP и медиа-приложений.

В этой главе будут рассмотрены компромиссы и конструктивные решения, которые приняли архитекторы Pentium 4, стремясь создать мегагерцового монстра, уделяя особое внимание инновационным функциям, которыми обладал Pentium 4, и тому, как эти функции сочетаются с общей конструкцией процессора. философия и целевая область применения. Мы рассмотрим сверхглубокий конвейер Pentium 4, его кэш трассировки, его ALU с двойной накачкой и множество других аспектов его конструкции с учетом их влияния на производительность. Для сравнения мы также рассмотрим микроархитектуру Motorola G4e. Изучая два микропроцессора рядом друг с другом, вы получите более глубокое понимание того, как концепции, изложенные в предыдущих главах, проявляются в паре популярных, реальных проектов.

Пристрастие Pentium 4 к скорости

В Таблице 7-1 перечислены функции Pentium 4.

Таблица 7-1: Возможности Pentium 4

Дата введения	23 апреля 2001 г.
Процесс	0,18 мкм
Количество транзисторов	42 миллиона
Тактовая частота при введении 1,7 ГГц	
Размеры кэша	L1: около 16 КБ инструкций, 16 КБ данных
Функции	В 2003 г. добавлена одновременная многопоточность (SMT, также известная как «гиперпоточность»). В 2004 г. добавлена поддержка 64-разрядных систем (EM64T) и SSE3. В 2005 г. добавлена технология виртуализации (VT).