**Архитектура системы команд**.

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

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

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

Следующий уровень отражает основную линию разграничения системы, а именно границу между системным программным обеспечением и аппаратурой. Эту идею можно развить и дальше и говорить о распределении функций между отдельными частями физической системы. Например, некоторый интерфейс определяет, какие функции реализуют центральные процессоры, а какие - процессоры ввода/вывода. Архитектура следующего уровня определяет разграничение функций между процессорами ввода/вывода и контроллерами внешних устройств. В свою очередь можно разграничить функции, реализуемые контроллерами и самими устройствами ввода/вывода (терминалами, модемами, накопителями на магнитных дисках и лентах). Архитектура таких уровней часто называется архитектурой физического ввода/вывода.

### Архитектура системы команд. Классификация процессоров (CISC и RISC)

Как уже было отмечено, архитектура набора команд служит границей между аппаратурой и программным обеспечением и представляет ту часть системы, которая видна программисту или разработчику компиляторов.

Двумя основными архитектурами набора команд, используемыми компьютерной промышленностью на современном этапе развития вычислительной техники являются архитектуры CISC и RISC. Основоположником CISC-архитектуры можно считать компанию IBM с ее базовой архитектурой /360, ядро которой используется с1964 года и дошло до наших дней, например, в таких современных мейнфреймах как IBM ES/9000.

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

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

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

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

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

Ко времени завершения университетских проектов (1983-1984 гг.) обозначился также прорыв в технологии изготовления сверхбольших интегральных схем. Простота архитектуры и ее эффективность, подтвержденная этими проектами, вызвали большой интерес в компьютерной индустрии и с 1986 года началась активная промышленная реализация архитектуры RISC. К настоящему времени эта архитектура прочно занимает лидирующие позиции на мировом компьютерном рынке рабочих станций и серверов.

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

Следует отметить, что в последних разработках компании Intel (имеется в виду Pentium P54C и процессор следующего поколения P6), а также ее последователей-конкурентов (AMD R5, Cyrix M1, NexGen Nx586 и др.) широко используются идеи, реализованные в RISC-микропроцессорах, так что многие различия между CISC и RISC стираются. Однако сложность архитектуры и системы команд x86 остается и является главным фактором, ограничивающим производительность процессоров на ее основе.

### Методы адресации и типы данных

#### Методы адресации

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

На рис. 4.1 представлены все основные методы адресации операндов, которые реализованы в компьютерах, рассмотренных в настоящем обзоре. Адресация непосредственных данных и литеральных констант обычно рассматривается как один из методов адресации памяти (хотя значения данных, к которым в этом случае производятся обращения, являются частью самой команды и обрабатываются в общем потоке команд). Адресация регистров, как правило, рассматривается отдельно. В данном разделе методы адресации, связанные со счетчиком команд (адресация относительно счетчика команд) рассматриваются отдельно. Этот вид адресации используется главным образом для определения программных адресов в командах передачи управления.

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

Использование сложных методов адресации позволяет существенно сократить количество команд в программе, но при этом значительно увеличивается сложность аппаратуры. Возникает вопрос, а как часто эти методы адресации используются в реальных программах? На рис. 4.2 представлены результаты измерений частоты использования различных методов адресации на примере трех популярных программ (компилятора с языка Си GCC, текстового редактора TeX и САПР Spice), выполненных на компьютере VAX.

|  |  |  |
| --- | --- | --- |
| **Метод адресации** | **Примеркоманды** | **Смысл командыметодаИспользование**  |
| Регистровая  | Add R4,R3  | R4(R4+R5Требуемое значение в регистре  |
| Непосредственная или литеральная | Add R4,#3 | R4(R4+3 Для задания констант  |
| Базовая со смещением  | Add R4,100(R1)  | R4(R4+M[100+R1]Для обращения клокальным переменным  |
| Косвенная регистровая  | Add R4,(R1)  | R4(R4+M[R1]Для обращения по указателю или вычисленному адресу  |
| Индексная | Add R3,(R1+R2) | R3(R3+M[R1+R2]Иногда полезна при работе с массивами: R1 - база, R3 - индекс  |
| Прямая илиабсолютная | Add R1,(1000)  | R1(R1+M[1000]Иногда полезна для обращения к статическим данным  |
| Косвенная  | Add R1,@(R3) | R1(R1+M[M[R3]]Если R3-адрес указателя p, то выбирается значение по этому указателю  |
| Автоинкрементная | Add R1,(R2)+ | R1(R1+M[R2]R2(R2+dПолезна для прохода в цикле по массиву с шагом: R2 - начало массиваВ каждом цикле R2 получает приращение d  |
| Автодекрементная | Add R1,(R2)- | R2(R2-dR1(R1+M[R2]Аналогична предыдущейОбе могут использоваться для реализации стека  |
| Базовая индексная со смещением и масштабированием | Add R1,100(R2)[R3] | R1(R1+M[100]+R2+R3\*dДля индексации массивов  |

*Рис. 4.1. Методы адресации*

*Рис. 4.2. Частота использования различных методов адресации на программах TeX, Spice, GCC*

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

При этом основной вопрос, который возникает для метода базовой адресации со смещением, связан с длиной (разрядностью) смещения. Выбор длины смещения в конечном счете определяет длину команды. Результаты измерений показали, что в подавляющем большинстве случаев длина смещения не превышает16 разрядов.

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

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

#### Типы команд

Команды традиционного машинного уровня можно разделить на несколько типов, которые показаны на рис. 4.3.

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

*Рис. 4.3. Основные типы команд*

#### Команды управления потоком команд

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

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

При этом примерно 90% команд безусловного перехода выполняются относительно счетчика команд. Для команд перехода адрес перехода должен быть всегда заранее известным. Это не относится к адресам возврата, которые не известны во время компиляции программы и должны определяться во время ее работы. Наиболее простой способ определения адреса перехода заключается в указании его положения относительно текущего значения счетчика команд (с помощью смещения в команде), и такие переходы называются переходами относительно счетчика команд. Преимуществом такого метода адресации является то, что адреса переходов, как правило, расположены недалеко от текущего адреса выполняемой команды и указание относительно текущего значения счетчика команд требует небольшого количества бит в смещении. Кроме того, использование адресации относительно счетчика команд позволяет программе выполняться в любом месте памяти, независимо от того, куда она была загружена. То есть этот метод адресации позволяет автоматически создавать перемещаемые программы.

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

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

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

Второй метод заключается в простом использовании произвольного регистра (возможно одного выделенного) общего назначения. В этом случае выполняется проверка состояния этого регистра, в который предварительно помещается результат операции сравнения. Недостатком этого подхода является необходимость выделения в программе для анализа кодов условий специального регистра.

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

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

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

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

#### Типы и размеры операндов

Имеется два альтернативных метода определения типа операнда. В первом из них тип операнда может задаваться кодом операции в команде. Это наиболее употребительный способ задания типа операнда. Второй метод предполагает указание типа операнда с помощью тега, который хранится вместе с данными и интерпретируется аппаратурой во время выполнения операций над данными. Этот метод использовался, например, в машинах фирмы Burroughs, но в настоящее время он практически не применяется и все современные процессоры пользуются первым методом.

Обычно тип операнда (например, целый, вещественный с одинарной точностью или символ) определяет и его размер. Однако часто процессоры работают с целыми числами длиною 8, 16, 32 или 64 бит. Как правило целые числа представляются в дополнительном коде. Для задания символов (1 байт = 8 бит) в машинах компании IBM используется код EBCDIC, но в машинах других производителей почти повсеместно применяется кодировка ASCII. Еще до сравнительно недавнего времени каждый производитель процессоров пользовался своим собственным представлением вещественных чисел (чисел с плавающей точкой). Однако за последние несколько лет ситуация изменилась. Большинство поставщиков процессоров в настоящее время для представления вещественных чисел с одинарной и двойной точностью придерживаются стандарта IEEE 754.

В некоторых процессорах используются двоично кодированные десятичные числа, которые представляются в в упакованном и неупакованном форматах. Упакованный формат предполагает, что для кодирования цифр 0-9 используются 4 разряда и что две десятичные цифры упаковываются в каждый байт. В неупакованном формате байт содержит одну десятичную цифру, которая обычно изображается в символьном коде ASCII.

В большинстве процессоров, кроме того, реализуются операции над цепочками (строками) бит, байт, слов и двойных слов.