# Современные RISC-архитектуры

## CISC vs RISC [1,2]

Традиционно, когда речь заходит об архитектуре процессоров вплывает несколько понятий, являющихся причиной споров, разногласий, а в академическо-студенческой среде еще и причиной микротрагедий). Многие эти термины слышали, или так или иначе с ними встречались.

Если говорить о «глобальной» архитектуре вычислительной машины (ЭВМ), то здесь фигурируют понятия «Фон-Неймановская архитектура (или Пристонская)» и «Гарвардская архитектура» (иногда называемая «Новой Гарвардской архитектурой», справедливо порождающей вопрос-на-засыпку бедному студенты – а какова же тогда была «старая Гарвардская архитектура»).

Формально - **архитектура фон Неймана (Принстонская архитектура)** характеризуется использованием общей оперативной памяти для хранения программ, данных, а также для организации стека. Для обращения к этой памяти используется общая системная шина, по которой в процессор поступают и команды, и данные.

Недостатком такой архитектуры является то, что и данные и команды из памяти должны передаваться в процессорному ядру и обратно. Даже тот факт, что современные процессоры имеют несколько уровней кэша непосредственно в ядре процессора, не решает проблему.

**Гарвардская архитектура** была разработана с целью увеличить скорость выполнения вычислительных операций и оптимизировать работу памяти. Она характеризуется физическим разделением памяти программ и памяти данных. Каждая память соединяется с процессором отдельной шиной, что позволяет одновременно с чтением-записью данных при выполнении текущей команды производить выборку и декодирование следующей команды. Также иногда выделяют отдельный стек для хранения адресов возврата из функций и подпрограмм.

За счет разделения потоков команд и данных реализуется потенциально более высокая производительность, чем при использовании архитектуры Фон-Неймана.

Недостатки Гарвардской архитектуры связаны с необходимостью проведения большего числа шин, а также с фиксированным объемом памяти, выделенной для команд и данных, назначение которой не может оперативно перераспределяться в соответствии с требованиями решаемой задачи.

При разделении каналов передачи команд и данных на кристалле процессора последний должен иметь почти вдвое больше интерфейсных выводов, так как шина адреса и шина данных составляют основную часть выводов микропроцессора. Способом разрешения этой проблемы стала идея использовать общие шину данных и шину адреса для всех внешних данных, а внутри процессора использовать шину данных, шину команд и две шины адреса. Такую концепцию стали называть **модифицированной гарвардской архитектурой**.

Подобная схемотехника применяется в современных сигнальных процессорах. Ещё дальше по пути уменьшения стоимости пошли при создании однокристальных микроЭВМ — микроконтроллеров. В них одна шина команд и данных применяется и внутри кристалла.

Часто требуется выбрать три составляющие: два операнда и инструкцию (в алгоритмах цифровой обработки сигналов это наиболее распространенная задача в БПФ, КИХ и БИХ-фильтрах). Для этого существует кэш-память. В ней может храниться инструкция — следовательно, обе шины остаются свободными и появляется возможность передать два операнда одновременно. Использование кэш-памяти вместе с разделёнными шинами получило название **«Super Harvard Architecture» («SHARC») — расширенная гарвардская архитектура**.

Архитектурные особенности процессора раскрываются понятиями CISC, RISC, MISC – характеризующие общие черты системы команд и общей организации процессора.

**CISC (Complete Instruction Set Computing)** - тип архитектуры процессора с полным набором команд. При этом подходе выполнение любой сколь угодно сложной команды из системы команд процессора реализовывается аппаратно внутри самого процессора.

Основную идею CISC-архитектуры отражает ее название - «полный набор команд». В данной архитектуре стремятся иметь отдельную машинную команду для каждого возможного (типового) действия по обработке данных. ((написанного преимущественно вручную на ассемблере) видели постоянно повторяющиеся паттерны инструкций, они стремились в следующей итерации архитектуры внести такие паттерны в архитектуру в виде новых инструкций).

В общие черты организации CISC-процессоров:

- большое количество различных машинных команд (сотни), каждая из которых выполняется за несколько тактов центрального процессора;

- устройство управления с программируемой логикой;

- небольшое количество регистров общего назначения;

- различные форматы команд с разной длиной;

- преобладание двухадресной адресации;

- развитый механизм адресации операндов, включающий различные методы косвенной адресации.

**RISC (Restricted (Reduced) Instruction Set Computer** - компьютер с сокращенным набором команд) - архитектура процессора, в которой быстродействие увеличивается за счет упрощения инструкций, чтобы их декодирование было более простым, а время выполнения - короче. В процессорах с RISC-архитектурой используется ограниченный набор быстрых команд. Каждая команда RISC-процессора должна выполняться за один машинный такт.

Для технологии RISC характерна сравнительно простая структура устройства управления. Площадь, выделяемая на кристалле микросхемы для его реализации, существенно меньше. Как следствие, появляется возможность разместить на кристалле большое число регистров ЦП. Кроме того, остается больше места для других узлов ЦП и для дополнительных устройств: кэш-памяти, блока арифметики с плавающей запятой, части основной памяти, блока управления памятью, портов ввода/вывода.

Унификация набора команд, ориентация на конвейерную обработку, унификация размера команд и длительности их выполнения, устранение периодов ожидания в конвейере - все эти факторы положительно сказываются на общем быстродействии.

Недостатки RISC прямо связаны с некоторыми преимуществами этой архитектуры. Принципиальный недостаток - сокращенное число команд: на выполнение ряда функций приходится тратить несколько команд вместо одной в CISC. Это удлиняет код программы, увеличивает загрузку памяти и трафик команд между памятью и ЦП. Исследования показали, что RISC-программа в среднем на 30% длиннее CISC-программы, реализующей те же функции.

**MISC-архитектура (Minimal Instruction Set Computer)** является процессором с минимальным набором команд. Она отличается ещё большей простотой и используется для ещё большего снижения энергопотребления и итоговой стоимости процессора. MISC-архитектура применяется в IoT-сегменте и компьютерах малой стоимости вроде роутеров.

MISC-архитектура для проектирования процессора, которая отличается наилучшей эффективностью и простотой в сравнении с CISC и RISC. Может содержать в себе блок RISC, обрабатывающий в себе от 10 базовых команд (+, —, /, \*, if, else & etc), из которых формируются более сложные операции над значениями, методом ветвления полученных результатов в ПЗУ. С точки зрения быстродействия, время выполнения инструкции, скорость записи и передачи данных в память, сократилось бы в разы, так как не нужно было бы ожидать, пока заполнится и очистится конвейер, а выполнять всё «потоково» без задержек.

Исторически первыми в массовом доступном коммерческом сегменте были CISC-процессоры. CISC-подход позволяет получить достаточно высокую плотность кода, что для того времени было одним из критических факторов.

**Из истории[2]**

К началу восьмидесятых однокристальные микропроцессоры освоились в 16-разрядной нише и наметился переход к 32 разрядам. Это уже была область компьютеров среднего класса мощности со своими лидерами, традициями, инструментами, опытом, … но также тупиками и накопленными ошибками.

В то же самое время компьютеры среднего класса стремительно дешевели, так, Interdata представила в 1973 г. первый 32-разрядный мини-компьютер (IBM 360 совместимый) дешевле 10 000 долларов.

Предпосылки к массовому внедрению RISC-подхода.

Во-первых. На тот момент уже существовала свободно распространяемая (~ по цене магнитной ленты) переносимая операционная система (UNIX) с переносимым системным языком С. Причем, они еще не успели обрасти мышцами и накопить жирка, так что добавление новой архитектуры или учет её изменений не были слишком дорогостоящими.

Во-вторых, произошла тихая революция в компиляторах. Условной точкой отсчета можно считать 1981 год, когда математиком Грегори Хайтином (Gregory Chaitin) был предложен [3,4] способ распределения регистров.

Речь идет о той стадии компиляции [5], когда программа преобразована во внутреннее представление в виде трёхадресного кода и до стадии кодогенерации. Трёхадресный код - фактически код для процессора с бесконечным количеством регистров. Но в конечной архитектуре количество регистров ограничено и требуется решить, какие значения можно оставить в регистрах, а какие следует переместить в память (грубо).

До того момента эта стадия компиляции выполнялась эмпирически, с помощью набора разнообразных рецептов/приёмов, которые следовало применять в тех или иных ситуациях, качество такого кода зачастую оказывалось сомнительным. Грегори Хайтин пришел к выводу, что задача распределения регистров сводится к задаче раскраски графа. Про эту задачу известно, что она NP-полная, т.е. стоимость ее решения экспоненциально зависит от числа вершин (исходных регистров, которые надо распределить). К счастью, в данном случае это не приговор, поскольку была также предложена приемлемая эвристика.

На первый взгляд, сколь-нибудь существенным элементом новизны в RISC процессорах было лишь увеличение числа регистров общего назначения. Однако это увеличение имеет и побочные эффекты помимо ускорения доступа к данным. В частности, использование трёхадресного кода. Не очень удобно использовать три регистра в одной инструкции, когда этих регистров всего восемь. Сравните с тридцатью двумя.

В RISC архитектуре компилятор при прочих равных имеет возможность задействовать больше регистров, чтобы уменьшить количество конфликтов конвейера.

Не только количество регистров, а и двухоперандность (переиспользование регистра как для чтения так и для записи результата), флаги результатов после всех операций (порождает больше конфликтов), одновременный доступ к памяти с арифметикой в одной инструкции, а также увеличение сложности при обработке исключений и спекулятивном выполнении (в том случае если мы превращаем cisc инструкцию в микроинструкции, так как исключения относятся к целой инструкции).

При равной производительности CISC сложнее, выделяют больше энергии, в них приходится раньше прибегать к трюкам вроде переименования регистров ... Тем не менее, одновременно существовали ветки RISC и CISC архитектур с близкой производительностью.

Технологическое преимущество RISC - регистры было нивелировано внутри-кристальным кэшем, обе стороны наращивали его объем, тактовую частоту и сложность конвейеров, интегрировали сопроцессоры с плавающей точкой … В какой-то момент микропроцессоры (в том числе и RISC) научились переименовывать регистры, переставлять микро-инструкции, спекулятивно исполнять код … и тут на сцену вышли суперскалярные микропроцессоры, которым стало (по большому счету) всё равно, какого типа - RISC или CISC у них набор внешних инструкций. С этого момента понятие RISC - не более, чем название раскрученного брэнда, которое используется преимущественно по инерции (либо маркетологами).

**Архитектура[6]**

Существует *фиксированное* количество команд, которые понимает каждый конкретный процессор.

Каждый конкретный процессор имеет определенный набор внутренних элементов для выполнения операций, хранения переменных и элементов внутреннего состояния.

**Архитектура набора команд** (instruction set architecture, **ISA**) — часть архитектуры компьютера, определяющая программируемую часть ядра микропроцессора.

На этом уровне определяются:

* архитектура памяти;
* взаимодействие с внешними устройствами ввода/вывода;
* режимы адресации;
* регистры;
* машинные команды;
* различные типы внутренних данных (например, с плавающей запятой, целочисленные типы и т. д.);
* обработчики прерываний и исключительных состояний.

**Микроархитектура**

Описывает модель, топологию и реализацию ISA на микросхеме микропроцессора. Фактически – каждый разработчик может предлагать свою микроархитетуру, свое видение реализации ISA, оптимизируя те или иные аспекты архитектуры, или предлагая решение для

На этом уровне определяется:

- конструкция и взаимосвязь основных блоков ЦП,

- структура ядер, исполнительных устройств, АЛУ, а также их взаимодействия,

- блоков предсказания переходов,

- организация конвейеров,

- организация кэш-памяти,

- взаимодействие с внешними устройствами.

В рамках одного семейства микропроцессоров микроархитектура со временем расширяется путём добавления новых усовершенствований и оптимизации существующих команд с целью повышения производительности, энергосбережения и функциональных возможностей микропроцессора. При этом сохраняется совместимость с ISA.

## MIPS [6-8]

MIPS (сокращение от названия соответствующего проекта Стэнфордского университета англ. Microprocessor without Interlocked Pipeline Stages, без блокировок в конвейере) — система команд и микропроцессорных архитектур, разработанных компанией MIPS Computer Systems (в настоящее время Wave Computing[2]) в соответствии с концепцией проектирования процессоров RISC (то есть для процессоров с упрощенным набором команд). Ранние модели процессора имели 32-битное машинное слово, позднее появились его 64-битные версии. Существует множество модификаций процессора, включая MIPS I, MIPS II, MIPS III, MIPS IV, MIPS V, MIPS32 и MIPS64, из них действующими являются MIPS32 (для 32-битной реализации) и MIPS64 (для 64-битной реализации). MIPS32 и MIPS64 определяют как набор регистров управления, так и набор команд.

Существует несколько наборов команд: MIPS32, MIPS64, microMIPS и nanoMIPS. Помимо этого, доступны дополненные наборы инструкций и модели процессоров, например, MIPS-3D, включающий в себя набор SIMD-команд для обработки чисел с плавающей запятой, предназначенный для решения простых 3D-задач, MDMX (MaDMaX) — с ещё более широкими возможностями — набором SIMD-команд и использующий 64-битные регистры с плавающей запятой для работы с целыми числами, MIPS16e, который сжимает поток команд, чтобы уменьшить объём памяти, занимаемый программами, а также MIPS MT, обеспечивающий многопоточный режим обработки.

Одним из главных препятствий в использовании конвейера был тот факт, что некоторые команды, такие, как деление, выполняются намного дольше, и, вследствие этого, центральному процессору приходится ждать, прежде, чем передать на конвейер следующую команду. Единственное решение этой проблемы — использовать серию блокировок, позволяющих определенным стадиям конвейера показать, что они заняты и, в этом случае, приостанавливать вышестоящие в потоке команды. Главным аспектом устройства MIPS было согласовать каждую подфазу каждой команды, в том числе кэширование, в один цикл, таким образом избегая необходимости в блокировках и пропуская на конвейер только один цикл.

Чтобы сэкономить биты в коде команды, в RISC было уменьшено количество инструкций для кодирования. В MIPS из 32 битов слова всего 6 используются для основного кода, а остальные могут содержать либо единственный 26-битный адрес перехода, либо до 5 полей, устанавливающих от 1 до 3 регистров + длина сдвига регистра. Существует и ряд других форматов, например, когда 2 регистра задаются непосредственно выделенным 16-битным полем и т. д. Такое распределение позволило процессору загружать команду и необходимые ей данные за один машинный такт, в то время как в более старых архитектурах (не являвшихся RISC), например, таких, как MOS Technology 6502, требовались отдельные такты для загрузки основного кода и данных.

Это было одним из главных усовершенствований, наращивающих производительность, предлагаемых RISC. Однако другие архитектуры всё же достигли подобной скорости, но другими средствами (такими, как очереди в ЦП).

Типичный конвейер MIPS (рис.1):

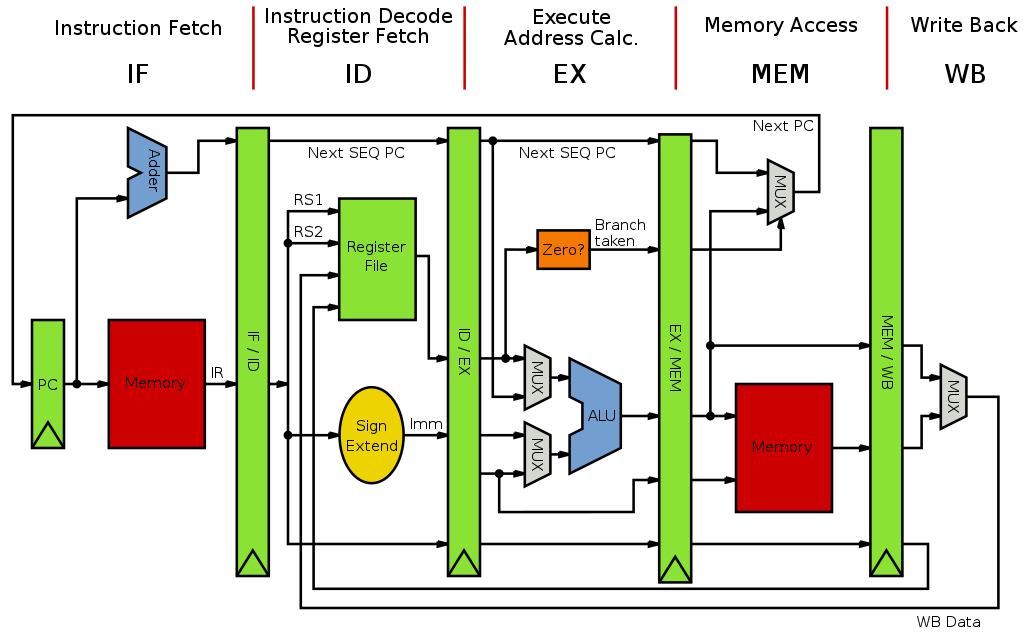


Рис. 1 Типичная структура конвейера архитектуры MIPS [7].

Архитектура MIPS определяет 32 целочисленных регистра и 32 регистра с плавающей точкой [6,8].

Инструкции делятся на три типа: R, I и J. Каждая инструкция начинается с 6-битного кода. В дополнение к коду, инструкции R-типа определяют три регистра, область размера сдвига регистра, и область функции; инструкции I-типа определяют два регистра и непосредственное значение; инструкции J-типа состоят из кода операции и 26-битного адреса перехода.

Целочисленные регистры (указаны типовое применения регистров по соглашениям):

**$zero ($0)** – регистр, всегда содержащий значение 0 и доступный только для чтения;

**$at ($1)** – временный регистр процессора;

**$v0-$v1 ($2-$3)** – для результатов, возвращаемых функциями;

**$a0-$a3 ($4-$7)** – для аргументов функций;

**$t0-$t9 ($8-$15, $24-$25)** – для временных данных, можно использовать как угодно;

**$s0-$s8 ($16-$23, $30)** – для постоянных данных, можно использовать как угодно;

**$k0-$k1 ($26-$27)** – зарезервировано для ядра операционной системы;

**$gp ($28)** – указатель глобальных переменных, практически не используется;

**$sp ($29)** – указатель стека, его значение всегда равно верхнему адресу стека;

**$ra ($31)** – адрес инструкции, из которой была вызвана функция.

Регистры **$k0** и **$k1** зарезервированы для использования операционной системой и не должны использоваться в пользовательских программах. Регистр **$at** используется ассемблером и не должен использоваться в пользовательских программах. Регистр **$gp** используется для указания глобальных данных (по мере необходимости) и не должен использоваться в пользовательских программах.

В дополнение к ранее перечисленным регистрам, существуют специализированные регистры.

Регистр **$pc** или program counter указывает на следующую команду, которая должна быть выполнена, и автоматически обновляется центральным процессором после выполнения команды. Обычно пользовательские программы не обращаются непосредственно к этому регистру.

Регистр **$status** или состояния является регистром состояния процессора и обновляется после каждой команды центрального процессора. Обычно пользовательские программы не получают прямого доступа к этому регистру.

Регистр **$caus**e или exception cause используется центральным процессором в случае исключения или неожиданного прерывания потока управления программой. Примеры исключений включают деление на 0, попытка доступа к недопустимому адресу памяти или попытка выполнить недопустимую инструкцию (например, попытка выполнить элемент данных вместо кода).

Регистры **$hi** и **$lo** используются некоторыми специализированными инструкциями умножения и деления. Например, число, кратное двум 32-разрядным значениям, может генерировать 64-разрядные результаты, которые хранятся в **$hi** и **$lo** (по 32 бита каждое или всего 64 бита).

Операции с плавающей точкой реализуется математическим сопроцессором.

Регистры с плавающей точкой.

**$f0** – для результатов, возвращаемых функциями, с плавающей запятой;

**$f4, $f6, $f8, $f10, $f16, $f18** – для временных данных с плавающей запятой;

**$f12, $f14** – для параметров функций с плавающей запятой.

Таблица 1

Форматы MIPS-инструкции

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Тип** | **Формат инструкций** | | | | | |
| **31..26** | **25..21** | **20..16** | **15..11** | **10..6** | **5..0** |
| **R** | Код (*6 бит*) | rs (*5 бит*) | rt (*5 бит*) | rd (*5 бит*) | степень сдвига (shamt, от [англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB.) *shift amount*) (*5 бит*) | функция (*6 бит*) |
| **I** | Код (*6 бит*) | rs (*5 бит*) | rt (*5 бит*) | Постоянное или [непосредственное](https://ru.wikipedia.org/wiki/%D0%97%D0%BD%D0%B0%D1%87%D0%B5%D0%BD%D0%B8%D0%B5_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)#%D0%9D%D0%B5%D0%BF%D0%BE%D1%81%D1%80%D0%B5%D0%B4%D1%81%D1%82%D0%B2%D0%B5%D0%BD%D0%BD%D0%BE%D0%B5_%D0%B7%D0%BD%D0%B0%D1%87%D0%B5%D0%BD%D0%B8%D0%B5) значение ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB.) *immediate value*) (*16 бит*) | | |
| **J** | Код (*6 бит*) | Адрес (*26 бит*) | | | | |

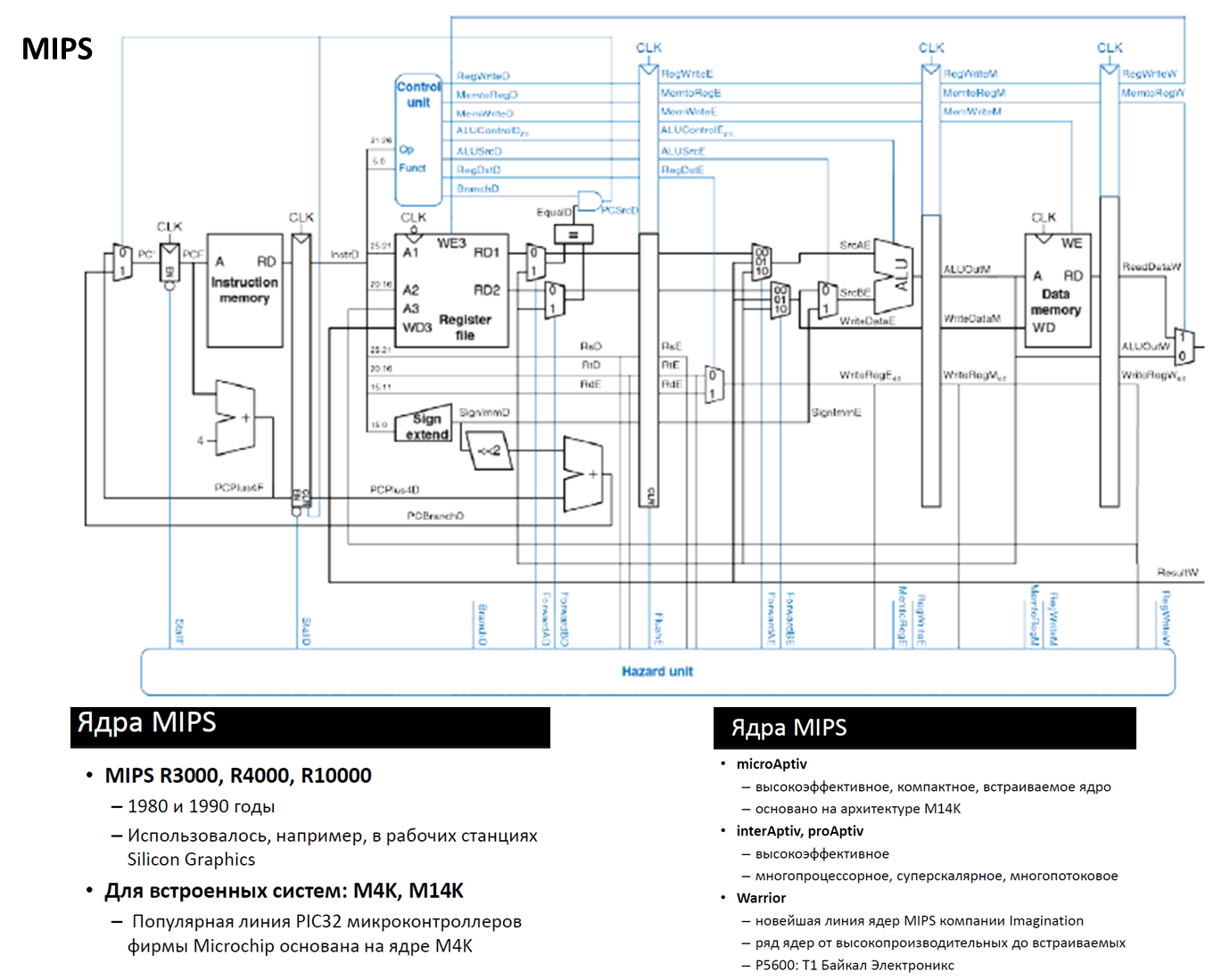


Рис. 2 Структурная схема процессорного ядра MIPS [6].

## ARM [9-11]

ARM (Advanced RISC Machine, Acorn RISC Machine — продвинутая ЭВМ с упрощённым набором команд, ПВМ / усовершенствованная RISC-машина) — семейство лицензируемых 32-битных и 64-битных микропроцессорных ядер, разрабатываемых компанией ARM Limited.

Известные семейства процессоров: ARM7, ARM9, ARM11 и Cortex.

**Процессоры ARM**

Профили архитектуры.

Архитектура развивалась с течением времени, и, начиная с ARMv7, были определены три профиля:

• ‘П’ (прикладной)/англ.‘A’ (application) — для устройств, требующих высокой производительности (смартфоны, планшеты)

• ‘В’ (реального времени) / англ.‘R’ (real time) — для приложений, работающих в реальном времени,

• ’М’ (микроконтроллер) / англ.’M’ (microcontroller) — для микроконтроллеров и недорогих встраиваемых устройств.

Профили могут поддерживать меньшее количество команд (команды определенного типа).

Примеры семейств процессоров ARM:

«Классические»

• ARM7 (с тактовой частотой до 60-72 МГц), предназначенные, например, для недорогих мобильных телефонов и встраиваемых решений средней производительности. В настоящее время активно вытесняется новым семейством

**ARM-Cortex**

• ARM9, ARM11 (с частотами до 1 ГГц) для более мощных телефонов, карманных компьютеров и встраиваемых решений высокой производительности.

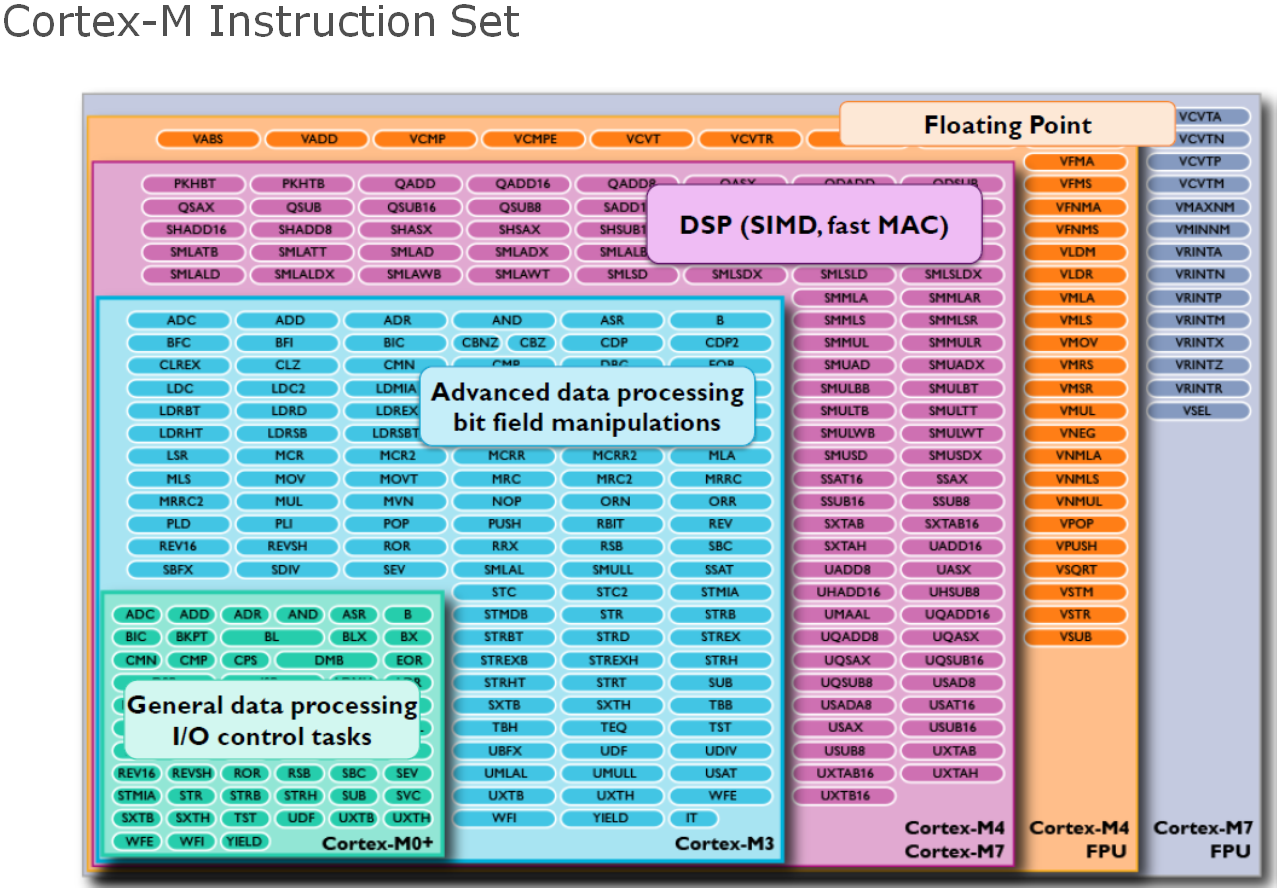


Рис. 3 Иерархия наборов команд ARM-Cortex.

**Cortex A**

• Cortex A57 – для высокопроизводительных мобильных устройств и микросерверов;

• Cortex-A76AE - с повышенной отказоустойчивостью и высокой производительностью.

**Cortex R**

• Cortex-R52 — производительный процессор для систем реального времени Cortex M

• Cortex M35P — надежный процессор для встраиваемых embedded систем.

**Neoverse**

• Neoverse E1 - Для серверов и датацентров, ориентирован на сетевые и облачные решения, системы хранения данных

• Neoverse N1 - Для серверов и датацентров, обработка больших массивов данных.

Режимы работы:

Процессор может находиться в одном из следующих операционных режимов:

• Пользовательский режим / англ. User mode — обычный режим выполнения программ. В этом режиме выполняется большинство программ.

• Режим быстрого прерывания / англ. Fast Interrupt (FIQ) — режим, где время срабатывания меньше.

• Основной режим прерывания / англ. Interrupt (IRQ).

• Системный режим / англ. System mode — защищённый режим для использования операционной системой.

• Режим аварийного отказа / англ. Abort mode — режим, куда переходит процессор, когда возникает ошибка доступа к памяти (доступ к данным или к команде на этапе prefetch-конвейера).

• Привилегированный пользовательский режим / англ. Supervisor mode.

• Неопределённый режим / англ. Undefined mode — режим, куда процессор

входит при попытке выполнить неизвестную ему команду.

Режим процессора переключается, когда возникает соответствующее исключение, или же при модификации регистра статуса.

Особенности архитектуры ARM

Функции RISC

Архитектура ARM обладает следующими особенностями RISC:

- регистровая архитектура (архитектура загрузки / хранения);

- нет поддержки нелинейного (не выровненного по словам) доступа к памяти (теперь поддерживается в процессорах ARMv6, за некоторыми исключениями, и полностью в ARMv7);

- равномерный 16х32-битный регистровый файл

- фиксированная длина команд (32 бит) для упрощения декодирования за счет

снижения плотности кода. Позднее режим Thumb повысил плотность кода.

- однотактное исполнение команд.

Чтобы компенсировать простой дизайн, в сравнении с «продвинутыми» на тот момент процессорами вроде Intel 80286 или Motorola 68020 были использованы некоторые «фишки»:

• арифметические инструкции меняют флаги условий, только когда это необходимо;

• 32-битное многорегистровое циклическое сдвиговое устройство, которое может быть использовано без потерь производительности в большинстве арифметических инструкций и адресных расчетов.

• мощные индексированные адресные режимы;

• регистр ссылок для быстрого вызова функций;

• простые, но быстрые, с двумя уровнями приоритетов подсистемы прерываний с включенными банками регистров.

• условное исполнение команд.

Одно из существенных отличий архитектуры ARM (изначальная архитектура) от

других архитектур процессоров — так называемая предикация, то есть возможность условного исполнения команд. Под «условным исполнением» здесь понимается то, что команда будет выполнена или проигнорирована в зависимости от текущего состояния флагов состояния процессора. В наборах команд Thumb и Arm64 предикация не используется — в первом режиме для неё нет места в команде (всего 16 бит), а во втором предикация бессмысленна и сложна для реализации на суперскалярных архитектурах.

В то время как для других архитектур таким свойством, как правило, обладают только команды условных переходов, в архитектуру ARM была заложена возможность условного исполнения практически любой команды. Это было достигнуто добавлением в коды их инструкций особого 4-битового поля (предиката). Одно из его значений зарезервировано на то, что инструкция должна быть выполнена безусловно, а остальные кодируют то или иное сочетание условий (флагов). С одной стороны, с учётом ограниченности общей длины инструкции, это сократило число битов, доступных для кодирования смещения в командах обращения к памяти, но с другой — позволило избавляться от инструкций ветвления при генерации кода для небольших if-блоков – «самый быстрый код – линейный!».

Один из способов, которым уплотнённый (Thumb) код достигает большей экономии объёма — это именно удаление 4-битового предиката из всех инструкций, кроме ветвлений.

**Другие особенности**

Другая особенность набора команд — это возможность соединять сдвиги и вращения в инструкции «обработки информации» (арифметическую, логическую, движение регистр-регистр) так, что, например, выражение на языке С:

a += (j << 2);

может быть преобразовано в команду из одного слова и одного цикла в ARM:

ADD Ra, Ra, Rj, LSL #2

Это приводит к тому, что типичные программы ARM становятся плотнее, чем обычно, с меньшим доступом к памяти. Таким образом, конвейер используется гораздо более эффективно. Даже несмотря на то, что ARM работает на скоростях, которые многие бы сочли низкими, он довольно-таки легко конкурирует с многими более сложными архитектурами

**Процессорное ядро**

ARM-процессор также имеет некоторые особенности, редко встречающиеся в других архитектурах RISC — такие, как адресация относительно счётчика команд (на самом деле счётчик команд ARM — это один из 16 регистров), а также пре- и пост-инкрементные режимы адресации.

Другая особенность, что стоит отметить, — это то, что некоторые ранние ARM- процессоры (до ARM7TDMI), например, не имеют команд для хранения 2-байтных чисел.

Таким образом, строго говоря, для них невозможно сгенерировать эффективный код, который бы вел себя так, как ожидается от объектов С, типа «volatile int16\_t».

**Конвейер и другие аспекты реализации**

ARM7 и более ранние версии имеют трехступенчатый конвейер. Это ступени переноса, декодирования и исполнения. Более производительные архитектуры, типа ARM9, имеют более сложные конвейеры. Cortex-a8 имеет 13-ступенчатый конвейер.

В то же время стоит отметить, что в архитектурах Cortex A65AE и Cortex A76AE

конвейер отсутствует (out-of-order), но при этом ядро остается суперскалярным.

**Сопроцессоры**

Архитектура предоставляет способ расширения набора команд, используя сопроцессоры, которые могут быть адресованы, используя MCR, MRC, MRRC, MCRR и похожие команды. Пространство сопроцессора логически разбито на 16 сопроцессоров с номерами от 0 до 15, причем 15-й зарезервирован для некоторых типичных функций управления, типа управления кэш-памятью и операции блока управления памятью (на процессорах, в которых они есть).

В машинах на основе ARM периферийные устройства обычно подсоединяются к процессору путём сопоставления их физических регистров в памяти ARM или в памяти сопроцессора, или путём присоединения к шинам, которые, в свою очередь, подсоединяются к процессору. Доступ к сопроцессорам имеет большее время ожидания, поэтому некоторые периферийные устройства проектируются для доступа в обоих направлениях. В остальных случаях разработчики чипов лишь пользуются механизмом интеграции сопроцессора.

Например, движок обработки изображений должен состоять из малого ядра ARM7TDMI, совмещенного с сопроцессором, который поддерживает примитивные операции по обработке элементарных кодировок HDTV.

**Усовершенствованный SIMD (NEON)**

Расширение усовершенствованного SIMD, также называемое технологией NEON — это комбинированный 64- и 128-битный набор команд SIMD (single instruction multiple data), который обеспечивает стандартизованное ускорение для медиаприложений и приложений обработки сигнала. NEON может выполнять декодирование аудиоформата mp3 на частоте процессора в 10 МГц, и может работать с речевым кодеком GSM AMR (adaptive multi-rate) на частоте более 13МГц. Он обладает внушительным набором команд, отдельными регистровыми файлами, и независимой системой исполнения на аппаратном уровне. NEON поддерживает 8-, 16-, 32-, 64-битную информацию целого типа, одинарной точности и с плавающей запятой, и работает в операциях SIMD по обработке аудио и видео (графика и игры). В NEON SIMD поддерживает до 16 операций единовременно.

Один из недостатков (или же особенностью) усовершенствованного SIMD — то, что сопроцессор выполняет команды усовершенствованного SIMD с достаточно значительной задержкой относительно кода основного процессора, задержка достигает двух десятков тактов и более (зависит от архитектуры и конкретных условий). По этой причине при попытке основного процессора воспользоваться результатами вычисления сопроцессора исполнение будет заморожено на значительное время.

**VFP**

Технология VFP (Vector Floating Point, вектора чисел с плавающей запятой) —

расширение сопроцессора в архитектуре ARM. Она производит низкозатратные вычисления над числами с плавающей запятой одинарной/двойной точности, в полной мере соответствующие стандарту ANSI/IEEE Std 754—1985 Standard for Binary Floating-Point Arithmetic. VFP производит вычисления с плавающей запятой, подходящие для широкого спектра приложений — например, для КПК, смартфонов, сжатие звука, трёхмерной графики и цифрового звука, а также принтеров и телеприставок. Архитектура VFP также поддерживает исполнение коротких векторных команд. Но, поскольку процессор выполняет операции последовательно над каждым элементом вектора, то VFP нельзя назвать истинным SIMD-набором инструкций. Этот режим может быть полезен в графике и приложениях обработки сигнала, так как он позволяет уменьшить размер кода и выработку команд.

Другие сопроцессоры с плавающей запятой и/или SIMD, находящиеся в ARM- процессорах, включают в себя FPA, FPE, iwMMXt. Они обеспечивают ту же функциональность, что и VFP, но не совместимы с ним на уровне опкодов.

**Расширения безопасности**

Расширения безопасности, позиционируемые как TrustZone Technology, находятся в ARMv6KZ и других, более поздних, профилированных на приложениях архитектурах. Оно обеспечивает низкозатратную альтернативу добавлению специального ядра безопасности, обеспечивая 2 виртуальных процессора, поддерживаемых аппаратным контролем доступа.

Это позволяет ядру приложения переключаться между двумя состояниями, называемыми «миры» (чтобы избежать путаницы с названиями возможных доменов), чтобы не допустить утечку информации из более важного мира в менее важный.

Этот переключатель миров обычно ортогонален всем другим возможностям процессора. Таким образом, каждый мир может работать независимо от других миров, используя одно и то же ядро. Память и периферия соответственно изготавливаются с учетом особенностей мира ядра, и могут использовать это, чтобы получить контроль доступа к секретам и кодам ядра.

Типичные приложения TrustZone Technology должны запускать полноценную операционную систему в менее важном мире, и компактный, специализированный на безопасности, код в более важном мире, позволяя Digital Rights Management’у намного точнее контролировать использование медиа на устройствах на базе ARM, и предотвращая несанкционированный доступ к устройству.

На практике же, так как конкретные детали реализации TrustZone остаются собственностью компании и не разглашаются, остается неясным, какой уровень безопасности гарантируется для этой модели угроз.

## Немного критики RISC архитектур от создателей RISC-V [12]

**MIPS**

Архитектура набора команд MIPS является одной из самых элегантных RISC ISA. Первоначально разработанная в Стэнфорде в начале 1980-х годов, на ее дизайн большое влияние оказал мини-компьютер IBM 801. Обе архитектуры представляют собой архитектуры типа load/store с регистрами общего назначения, в которых доступ к памяти осуществляется только с помощью инструкций, которые копируют данные в регистры и из них, а арифметика работает только с регистрами. Такая конструкция снижает сложность как набора команд, так и аппаратного обеспечения, облегчая недорогую конвейерную реализацию, полагаясь при этом на улучшенные технология компилятора. MIPS был впервые коммерчески реализован в процессоре R2000 в 1986 году.

В своем первоначальном воплощении набор целочисленных команд пользовательского уровня MIPS состоял всего из 58 инструкций и был прост в реализации в виде конвейера с одним выпуском по порядку. Постепенно набор команд превратился в гораздо более крупный ISA, который теперь содержит около 400 инструкций.

В то время как простые микроархитектурные реализации MIPS-I вполне доступны академическим компьютерным архитекторам, ISA имеет ряд технических недостатков, которые делают его менее привлекательным для высокопроизводительных реализаций:

ISA чрезмерно оптимизирован для конкретной микроархитектурной схемы, пятиступенчатой, конвейер с одной проблемой по порядку. Ветви и переходы задерживаются на одну команду, усложняя суперскалярные и суперпереходные реализации. Отложенные ветви увеличивают размер кода и тратят впустую полосу пропускания команд, когда интервал задержки не может быть надлежащим образом заполнен. Даже для классического пятиступенчатого конвейера удаление интервала задержки и добавление небольшого целевого буфера ветвления обычно приводит к повышению абсолютной производительности и производительности на единицу площади.

Другие риски конвейера, включая риски данных при загрузке, умножении и делении, были обнаружены в MIPS-I, но более поздние версии ISA удалили эти бородавки, отражая тот факт, что блокировка этих опасностей одновременно проще для программного обеспечения и может обеспечить более высокую производительность. Слот задержки перехода, с другой стороны, не может быть удален при сохранении обратной совместимости.

ISA обеспечивает слабую поддержку позиционно-независимого кода (positional independent code - PIC) и, следовательно, динамической компоновки. Инструкции прямого перехода являются псевдоабсолютными, а не относительными к программному счетчику, что делает их бесполезными в PIC; вместо этого MIPS использует исключительно косвенные переходы, при значительном размере кода и снижении производительности. (В версии MIPS 2014 улучшена адресация относительно ПК, но вызовы функций относительно ПК по-прежнему обычно требуют более одной инструкции.)

Когда архитекторы MIPS попытались уменьшить размер кода с помощью сжатой кодировки команд, у них не было другого выбора, кроме как создать вторую кодировку команд, включено переключателем режимов, поскольку они не смогли преобразовать новые инструкции в исходную кодировку.

Для умножения и деления используются специальные архитектурные регистры, увеличивающие размер контекста, количество команд, размер кода и микроархитектурную сложность.

ISA предполагает, что модуль с плавающей точкой является отдельным сопроцессором и является неоптимальным для однокристальных реализаций. Например, преобразования с плавающей запятой в целое число записывают свои результаты в файл регистра с плавающей запятой, что обычно требует дополнительной команды перемещения для использования результата. Усугубляя эту стоимость, перемещения между целыми файлами и файлами регистров с плавающей запятой имеют программный интервал задержки.

В стандартном ABI (соглашение по вызову функций) два целочисленных регистра зарезервированы для программного обеспечения ядра, уменьшение количества регистров, доступных пользовательским программам. Несмотря на это, регистры имеют ограниченное применение для ядра, поскольку они не защищены от доступа пользователей.

Обработка несогласованных загрузок и хранилищ с помощью специальных инструкций занимает значительное пространство в коде операции и усложняет все реализации, кроме простейших.

Архитекторы опустили инструкции сравнения целочисленных величин и ветвления - соотношение тактовой частоты и CPI, которое сегодня менее уместно с появлением предсказания ветвления и переходом к статической логике CMOS.

Помимо технических проблем, MIPS непригоден для многих целей, поскольку является проприетарным набором команд. Исторически сложилось так, что патент MIPS Technologies на несогласованные инструкции загрузки и сохранения помешал другим полностью внедрить ISA. В одном случае судебный процесс был направлен против компании, чьи реализации MIPS исключали инструкции, утверждая , что эмуляция инструкций в программном обеспечении ядра по-прежнему нарушает патент. Хотя срок действия патента с тех пор истек, MIPS остается торговой маркой Imagination Technologies; MIPS совместимость не может быть заявлена без их разрешения.

И да – перевод архитектуры MIPS в открытую ее не спас – лавинообразное повальное увлечение и переход на RISC-V вынудил прекратить поддержку архитектуры даже Imagination Technologies.

**SPARC**

Архитектура Oracle SPARC, первоначально разработанная Sun Microsystems, ведет свое происхождение от проектов Berkeley RISC-I и RISC-II. Самая последняя 32-разрядная версия ISA, SPARC V8, не является чрезмерно сложным: целочисленный ISA пользовательского уровня имеет простую, обычную кодировку и содержит всего 90 инструкций. Аппаратная поддержка IEEE 754-1985 floating point добавляет еще 50 инструкций, а режим супервизора - еще 20. Тем не менее, несколько дизайнерских решений ISA делают его несколько менее привлекательным для реализации, чем MIPS.

Для ускорения вызовов функций SPARC использует большой оконный файл регистров. На границах вызова процедуры окно сдвигается, создавая у вызываемого объекта видимость нового набора регистров. Такая конструкция устраняет необходимость в коде сохранения и восстановления регистра, сохраняемом вызываемым пользователем, что уменьшает размер кода и, как правило, повышает производительность. Однако, если рабочий набор стека вызовов процедур превышает количество окон регистрации, производительность резко снижается: операционная система должна регулярно вызываться для обработки окна поверх потоков и под потоками.

Значительно возросшее архитектурное состояние увеличивает затраты времени выполнения на переключение контекста и необходимость вызывать операционную систему для использования регистровых окон, что полностью исключает потоковую обработку на уровне пользователя.

Аппаратно регистровые окна занимают значительную площадь и потребляют много энергии для всех реализаций. Методы снижения их стоимости, в частности, усложняют суперскалярные реализации . Например, чтобы избежать выделения большого количества портов во всем наборе архитектурных регистров, UltraSPARC-III предоставляет теневую копию активного регистрового окна. Теневая копия должна обновляться при смене регистрового окна, что приводит к разрыву конвейера при большинстве вызовов и возвращений функций. Реализации неупорядоченного выполнения Fujitsu пошли на столь же героический шаг, объединив логику адресации окна регистра в схему переименования регистров.

В ветвях используются коды условий, которые дополняют архитектурное состояние и усложняют реализации, создавая дополнительные зависимости между некоторыми инструкциями. Вышедшие из строя микроархитектуры с переименованием регистров нуждаются в отдельном переименовании условия коды для устранения частых проблем с сериализацией. Отсутствие объединенной инструкции compareand-branch также увеличивает количество статических и динамических команд для общих кодовых последовательностей.

Инструкции, которые загружают и сохраняют соседние пары регистров, привлекательны для простых микроархитектур, поскольку они увеличивают пропускную способность при небольшой дополнительной аппаратной сложности. Увы, они усложняют реализацию переименованием регистров, поскольку значения данных больше не являются физически смежными в файле регистров.

Перемещения между файлами регистров с плавающей запятой и целыми числами должны использовать систему памяти в качестве посредника, что ограничивает производительность для кода смешанного формата.

ISA предоставляет неточные исключения с плавающей точкой посредством архитектурно доступной очереди отложенных перехватчиков, которая предоставляет программному обеспечению супервизора информацию для восстановления состояния процессора при таком исключении.

Единственной операцией с атомарной памятью является выборка и сохранение, чего недостаточно для реализации многих структур данных, не требующих ожидания .

SPARC разделяет многие близкие черты ISA других RISC-архитектур 1980-х годов.

Он был разработан для реализации в виде одноступенчатого, последовательного, пятиступенчатого конвейера, и вся ISA это подтверждает. В SPARC есть интервалы задержки ветвления и множество уязвимостей для данных и управления, которые усложняют генерацию кода. Кроме того, отсутствует поддержка позиционно-независимой адресации данных (ну так ли это это необходимо – вопрос на самом деле).

Наконец, SPARC не может быть легко модифицирован для поддержки сжатого расширения ISA, поскольку ему не хватает достаточного свободного пространства для кодировки.

В отличие от других коммерческих RISC архитектур, SPARC V8 является открытым стандартом, к чести Sun. SPARC International продолжает предоставлять разрешительные лицензии на V8 и V9, 64-разрядную версию ISA, за административный сбор в размере 99 долларов. Открытый ISA привел к появлению свободно доступных реализаций, две из которых являются производными от собственной микроархитектуры Sun Niagara. Увы, продолжающаяся разработка архитектуры Oracle SPARC [74] является проприетарной, и высокопроизводительное программное обеспечение, скорее всего, последует их примеру, оставив позади реализации более старого открытого набора команд.

**Alpha**

Архитекторы Digital Equipment Corporation воспользовались несколькими годами ретроспективного анализа, когда в начале 1990-х годов они разработали свой RISC ISA, Alpha. Они опустили многие из наименее привлекательных функций первых коммерческих RISC ISA, включая интервалы задержки перехода, коды условий и окна регистрации, и создали 64-разрядный ISA с адресным пространством, который был четко спроектирован, прост в реализации и обладал высокой производительностью. Кроме того, архитекторы Alpha тщательно выделили большинство деталей привилегированной архитектуры и аппаратная платформа, стоящая за абстрактным интерфейсом, библиотека привилегированной архитектуры (PALcode).

Тем не менее, DEC чрезмерно оптимизировала альфа-версию для микроархитектур in-order и добавила несколько функций, которые менее чем желательны для современных реализаций:

В погоне за высокой тактовой частотой оригинальная версия ISA отказалась от 8- и 16-разрядных операций с памятью, эффективно создав систему памяти с адресацией по 32-битным словам. Чтобы компенсировать производительность приложений, которые широко использовали эти операции, они добавлены специальные несогласованные инструкции загрузки и сохранения и несколько целочисленных инструкций для ускорения перестройки. В конце концов, архитекторы осознали ошибочность своих методов производительность приложений по-прежнему страдала, и было невозможно реализовать некоторые драйверы устройств и добавили в ISA подзаголовки загрузки и сохранения. Но они по-прежнему были обременены старыми инструкциями по обработке выравнивания, которые больше не были особенно полезны.

Чтобы облегчить выполнение инструкций с плавающей точкой с длительной задержкой выполения, Alpha имеет неточную модель ловушки-исключения с плавающей точкой. Это решение могло бы быть приемлемым по отдельности, но ISA также определяет, что флаги исключений и значения по умолчанию, при желании, должны предоставляться программными процедурами. Такое сочетание губительно для программ, совместимых с IEEE-754: инструкции с барьером ловушки должны вставляться после большинства арифметических инструкций с плавающей запятой (или, если соблюдается список ограничений на генерацию кода в стиле барокко, один раз на базовый блок).

В Alpha отсутствует инструкция по целочисленному делению, вместо этого предлагается использовать программное обеспечение

Итерационная схема Ньютона-Рафсона. Такой подход значительно увеличивает количество команд для некоторых программ и экономит лишь небольшое количество аппаратного обеспечения. Неожиданным следствием является то , что в большинстве реализаций деление с плавающей запятой значительно быстрее, чем целочисленное.

Как и в случае с его предшественниками RISC, не было продумано возможное расширение сжатого набора команд, и поэтому для его обновления остается недостаточно места для кода операции.

ISA содержит условные перемещения, которые усложняют микроархитектуры с переименованием регистров: в случае, если условие перемещения не выполняется, команда все равно должна скопировать старое значение в новый физический регистр назначения. Это эффективно делает условное перемещение единственной инструкцией в ISA, которая считывает три исходных операнда.

Действительно, первая реализация DEC с выполнением не по порядку использовала некоторые ухищрения , чтобы избежать дополнительного пути к данным для этой инструкции. Выполненный Alpha 21264 команда условного перемещения путем разделения ее на две микрооперации, первая из которых оценивала условие перемещения, а вторая выполняла перемещение.

Этот подход также требовал, чтобы файл физического регистра был расширен на один бит для хранения промежуточного результата.

Вскоре после того, как Compaq приобрела то, что осталось от слабеющего DEC в конце 1990-х, они решили отказаться от Alpha в пользу архитектуры Intel Itanium. Compaq продала Интеллектуальная собственность Alpha перешла к Intel [80], и вскоре после этого HP, которая с тех пор приобрела Compaq, выпустила окончательную реализацию Alpha в 2004 году.

**ARMv7**

ARMv7 - популярная 32-разрядная ISA, основанная на RISC, и, безусловно, наиболее широко реализуемая архитектура в мире. При начале разработки авторы RISC-V оцеивали, стоит ли разрабатывать наш собственный набор команд, ARMv7 стал естественной альтернативой из-за большого количества программного обеспечения, которое было портировано на данную ISA, и его повсеместного распространения во встроенных и мобильных устройствах. В конечном счете, и, единственный возможный недостаток ARMv7 - это закрытый стандарт. Подстановка ISA или расширение его новыми инструкциями явно запрещены; даже микроархитектурные инновации ограничены лицензионным соглашением (ARM называет архитектурной лицензией).

Однако, если бы обременения интеллектуальной собственностью не были проблемой, в ARMv7 есть несколько технических недостатков, которые сильно не позволяют на её использование:

Одно время не было поддержки 64-разрядных адресов, а в ISA отсутствовала аппаратная поддержка стандарта IEEE 754-2008. (ARMv8 исправил эти недостатки)

Детали привилегированной архитектуры просачиваются в определение архитектуры пользовательского уровня. Это касается не только эстетики. ARMv7 классически не поддается виртуализации, поскольку, среди прочих причин, инструкция возврата из исключения, RFE, не предназначена для перехвата при выполнении в пользовательском режиме. ARM добавила гипервизор режим привилегий в последних версиях архитектуры, но на момент написания этой статьи классическая виртуализация остается невозможной без динамической двоичной трансляции.

ARMv7 поставляется со сжатым ISA с 16-разрядными инструкциями фиксированной ширины, называемыми Thumb. Thumb предлагает конкурентоспособный размер кода, но низкую производительность, особенно в коде с интенсивным использованием операций с плавающей точкой. Позже появился набор команд переменной длины Thumb-2, обеспечивающий гораздо более высокую производительность. К сожалению, поскольку Thumb-2 был задуман после того, как была определена базовая ARMv7 ISA, 32-разрядные инструкции в Thumb-2 кодируются по-разному чем 32-разрядные инструкции в базовом ISA. (16-разрядные инструкции в Thumb-2 также кодируются иначе, чем 16-разрядные инструкции в исходном Thumb ISA.)

Эффективно, декодеры команд должны понимать три ISA, что увеличивает энергопотребление, задержку и стоимость проектирования.

ISA имеет множество функций, которые усложняют реализацию. Это не совсем универсальная регистровая архитектура: программный счетчик является одним из адресуемых регистров, что означает, что почти любая команда может изменить поток управления. Что еще хуже, программный счетчик является одним из адресуемых регистров. наименее значимый бит программного счетчика отражает, какой ISA выполняется в данный момент (ARM или Thumb) команда Thumb ADD может изменить, какой ISA в данный момент выполняется на процессоре! Использование кодов условий для ветвей и предикации еще больше усложняет высокопроизводительные реализации.

ARMv7 обширен и сложен. Между ARM и Thumb в одном только integer ISA3 содержится более 600 инструкций. NEON, расширение integer SIMD и с плавающей точкой, добавляет еще сотни.

**ARMv8**

В 2011 году, через год после того, как был запущен проект RISC-V, ARM анонсировала полностью переработанный ISA, ARMv8, с 64-разрядными адресами и расширенным набором целочисленных регистров.

Новая архитектура удалила несколько функций ARMv7, которые усложняли реализацию:

- например, программный счетчик больше не является частью набора целочисленных регистров; инструкции больше не используются.

- больше не предикат; инструкции load-multiple и store-multiple были удалены; и кодировка команд была упорядочена.

Но остается много проблем, включая использование кодов условий и регистров не совсем общего назначения (регистр ссылок является неявным и, в зависимости от контекста, x31 является либо указателем стека, либо жестко привязан к нулю). И было добавлено еще больше недостатков, включая массивную архитектуру subword-SIMD, которая является фактически обязательной 4. В целом ISA сложна и громоздка: в ней 1070 инструкций, включающих 53 формата и восемь режимов адресации данных.

ARMv8 тесно переплетает пользовательскую и привилегированную архитектуры, часто таким образом, что раскрывает базовую реализацию. В одном необъяснимом примере который сочетает в себе сложную семантику, неопределенное поведение и зависящие от регистра свойства предположительно регистров общего назначения команда load-pair может предоставьте пользовательскому пространству неточное исключение:

Если кодировка инструкции определяет адресацию с предварительной индексацией или адресацию с последующей индексацией,

и (t == n || t2 == n) && n != 31, то может произойти одно из следующих действий:

- инструкция НЕ ОПРЕДЕЛЕНА.

- команда выполняется как NOP.

Команда выполняет загрузку, используя указанный режим адресации, и базовому регистру присваивается НЕИЗВЕСТНОЕ значение. Кроме того, если во время выполнения такой команды возникает исключение, базовый регистр может быть поврежден таким образом, что команда не сможет быть повторена.

Кроме того, с появлением ARMv8 ARM отказалась от поддержки сжатого кодирования команд. Набор команд compact Thumb не был перенесен в 64-разрядное адресное пространство. Это правда, что ARMv8 довольно компактен для ISA с инструкции фиксированной ширины, но, как мы покажем в главе 5, они не могут конкурировать по размеру кода с ISA переменной длины. Что, безусловно, не является совпадением, первые 64-разрядные реализации ARM имеют на 50% больший объем кэша команд, чем их 32-разрядные аналоги.

Наконец, как и его предшественник, ARMv8 является закрытым стандартом. Его нельзя подразделить, что делает реализации слишком громоздкими для использования в качестве встроенных процессоров или блоков управления для пользовательских ускорителей. Фактически, тесно связанные сопроцессоры практически невозможно спроектировать на основе этого набора команд, поскольку он не может быть расширен никем, кроме ARM. Всё упирается в стоимость лицензии.

**OpenRISC**

Проект OpenRISC - это проект проектирования процессоров с открытым исходным кодом, который развился и образовательной архитектуры DLX, представленной в учебнике Хеннесси и Паттерсона "Эффективная компьютерная архитектура". Как бесплатный и открытый ISA, OpenRISC юридически подходит для использования в академических, исследовательских и промышленных приложениях. Однако, как и DLX, он имеет ряд технических недостатков, которые ограничивают его применимость:

Проект OpenRISC - это, в основном, дизайн открытого процессора, а не открытая спецификация ISA. ISA и реализация очень тесно связаны.

Фиксированная 32-разрядная кодировка с 16-разрядными немедленными выводами исключает сжатое расширение ISA.

Версия стандарта IEEE 754 2008 года аппаратно не поддерживается. Коды условий, используемые для ветвлений и условных перемещений, усложняют высокопроизводительные реализации.

ISA обеспечивает слабую поддержку позиционно-независимой адресации данных.

OpenRISC классически не виртуализуем, поскольку инструкция возврата из исключения, L.RFE, предназначена для нормальной работы в пользовательском режиме, а не для перехвата.

У ISA OpenRISC было два дополнительных недостатка: обязательные интервалы задержки перехода и отсутствие варианта с 64-разрядным адресным пространством. К чести архитекторов, оба эти фактора были исправлены: интервалы задержки стали необязательными, и была определена 64-разрядная версия (но, насколько нам известно, так и не была реализована).

**Заключение**

В итоге по результатам анализа архитектур, разработчиками RISC-V было принято решение вести разработку архитектуры «с нуля», насколько конечно это возможно ( все равно подсознательно архитектурные решения держатся в уме и несколько влияют на процесс разработки) - так видно, что архитектура RISC-V в значительной степени напоминает MIPS.

## Литература

1. Что означает RISC и CISC\_ \_ Хабр // https://habr.com/ru/company/selectel/blog/542074/
2. Три архитектуры эльфам, семь гномам, девять людям… где же искать ту, что объединит их все // <https://habr.com/ru/articles/596193/>
3. G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke,M. E. Hopkins, and P. W. Markstein. Register allocation via graph coloring. Computer Languages, 6(1):47–57, Jan. 1981
4. Chaitin,Register allocation & spilling via graph coloring. IBM Research POBox 218, Yorktown Heights, NY 10598 // <https://cs.gmu.edu/white/CS640/p98-chaitin.pdf>
5. [6] Aho, Sethi, Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley, 1986. ISBN 0-201-10088-6
6. Дэвид М. Харрис и Сара Л. Харрис. Цифровая схемотехника и архитектура компьютера Издательство Morgan Kaufman 2013.
7. Изучаем MIPS-ассемблер // <https://habr.com/ru/articles/147685/>
8. MIPS Reference Data. Basic Instruction Formats Register Name, Number, Use, Call Convention Core Instruction Set // https://inst.eecs.berkeley.edu/~cs61c/resources/MIPS\_Green\_Sheet.pdf
9. Объяснение терминологии Arm\_ Arm ARM, Armv9, ARM9, ARM64, Aarch64, A64, A78… \_ Хабр.pdf // <https://habr.com/ru/articles/722824/>
10. ARM (Advanced RISC Machine) // <https://btpit36.ru/pluginfile.php/4559/mod_resource/content/1/ARM.pdf>
11. Дэвид М. Харрис, Сара Л. Харрис. Цифровая схемотехника и архитектура компьютера. Дополнение по архитектуре ARM / пер. с англ. Слинкин А. А. / науч. ред. Косолобов Д. А. – М.: ДМК Пресс, 2019. – 356 с.: ил.
12. Electrical Engineering and Computer Sciences, University of California at Berkeley, Technical Report No. UCB/EECS-2016-1 // <http://www.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-1.html>