**Архитектура RISC-V**

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

**Микроархитектура**

На этом уровне определяются реализованные в микропроцессоре конкретного типа:

- 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.

Помимо системы команд в архитектуру микропроцессора входят наборы регистров и прочие структуры для хранения и обработки данных.

Архитектура RISC-V изначально позиционируется, как открытая расширяемая архитектура.

Одна из существенных составляющих ее взрывного роста на рынке – «открытость» - т.е. никому, ни за что не надо платить «роялити» - лицензионные отчисления. Что в перспективе массового производства не такие уж и маленькие суммы. А в условиях нестабильной политической остановки открытость архитектуры, это еще и существенное снижение рисков попасть под экономические или политические запреты – на любой стадии готовности продукта или технологии.

В базовом варианте RISC-V наследует характерные черты классической RISC-архитектурами, со всеми вытекающими из этого последствиями:

- небольшой набор базовых команд, в основном регистрово-ориентированных;

- эффект «разрастания» кода из-за того, что для выполнения комплексных действий приходится прописывать последовательность нескольких простых;

- распределение «сложности формирования эффективного исполнимого программного кода» между архитектурой и компилятором, с перевесом бОльшего объема работы на компилятор.

В архитектуре RISC-V имеется обязательное для реализации небольшое подмножество команд (набор инструкций I — Integer) и несколько стандартных опциональных расширений.

В базовый набор входят команды условных/безусловных переходов, минимальный набор регистровых арифметико-логических операций, операций с памятью (load/store), а также небольшое число служебных инструкций.

Команды ветвления не используют каких-либо общих флагов, а непосредственно сравнивают свои регистровые операнды. Базис операций сравнения минимален, а для поддержки комплементарных операций (например, операций больше/меньше) операнды просто меняются местами.

Регистровая модель очень напоминает MIPS, хотя и имеет ряд отличий:

**32-регистра общего** (условно)) назначения:

- регистр x0 (zero);

- 31 целочисленный регистр общего назначения (x1 — x31);

**Регистр счётчика команд** (PC, используется только косвенно);

**Набор CSR-регистров** или регистров специального назначения (Control and Status Registers, всего их может быть до 4096, хотя реально в каждой конкретной микроархитектурной реализации их заметно меньше).

Довольно важным моментом в экосистеме RISC-V является ABI, дающий рекомендации по распределению ролей регистров. Следовать ABI не обязательно и программист может распоряжаться любыми регистрами по своему усмотрению, но в ряде случаев произвольное распределение ролей регистров может вызвать проблемы:

- очевидный случай, связанный с вызовом бинарного кода сторонних разработчиков и ему подобные проблемы совместимости;

- потеря преимуществ в случае микроархитектурных оптимизаций «железа» по рекомендациям ABI.

(Двоичный интерфейс приложений (ABI) — набор соглашений для доступа приложения к операционной системе и другим низкоуровневым сервисам, спроектированный для переносимости исполняемого кода между машинами, имеющими совместимые ABI.

В отличие от API, который регламентирует совместимость на уровне исходного кода, ABI можно рассматривать как набор правил, позволяющих компоновщику объединять откомпилированные модули компонента без перекомпиляции всего кода, в то же время определяя двоичный интерфейс.)

**Имена регистров в системе команд и соглашения о псевдонимах в EABI и psABI**

32 целочисленных регистра (базового набора инструкций)

***x0 zero*** Константа – ноль (0);

***x1 ra*** Адрес возврата (return address)

***x2 sp*** Указатель стека (stack pointer)

***x3 gp*** Глобальный указатель (global pointer)

***x4 tp*** Потоковый указатель (thread pointer)

***x5 t0*** временная переменная/ альтернативный адрес возврата

***x6 s3*** временная переменная/Вызывающий

***x7 s4*** временная переменная/Вызывающий

***x8 s0/fp*** сохраняемая переменная / Указатель фрейма (frame pointer)

***x9 s1*** сохраняемая переменная

***x10 a0*** Аргумент функции / возвращаемое значение

***x11 a1*** Аргумент функции / возвращаемое значение

***x12 a2*** Аргумент функции

***x13 a3*** Аргумент функции

***x14 s2*** Аргумент функции

***x15 t1*** Аргумент функции

***x16 s5*** Аргумент функции

***x17 s6*** Аргумент функции

***x18-27 s7-16*** сохраняемые переменные

***x28-31 s17-31*** Временные переменные

Расширения набора команд для работы с плавающей точкой добавляют в набор регистров процессорного ядра еще 32 регистра:

**32 регистра с плавающей точкой**

***f0-7 ft0-7*** Временные значения

***f8-9 fs0-1*** Сохраняемые регистры

***f10-11 fa0-1*** Аргументы функций/возвращаемые значения

***f12-17 fa2-7*** Аргументы функций

***f18-27 fs2-11*** Аргументы функций , сохраняемые значения

***f28-31 ft8-11*** Временные значения

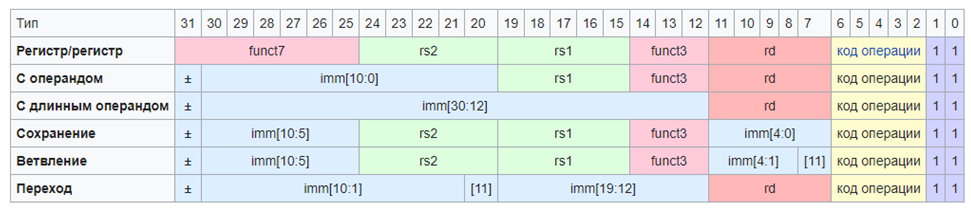
Как показывает практика, количество доступных архитектурных регистров может оказывать значительное влияние на размер кода, производительность и энергопотребление. Большее количество целочисленных регистров также повышает производительность в тех случаях, когда практикуется развертывание циклов (помним – быстрый код – линейный код), конвейерная микроархитектура, разбиение кэша на листы.

В динамическом использовании регистров, как правило, преобладают несколько часто используемых регистров, и микроархитектурные реализации регистровых файлов могут быть оптимизированы для снижения энергозатрат на доступ к часто используемым регистрам.

При одинаковой кодировке инструкций в RISC-V предусмотрены реализации архитектур с 32-, 64- и 128-битными регистрами общего назначения и операциями (RV32I, RV64I и RV128I, соответственно).

Разрядность регистровых операций всегда соответствует размеру регистра, а одни и те же значения в регистрах могут трактоваться как целые числа, как со знаком, так и без знака. Нет операций над частями регистров, нет каких-либо выделенных «регистровых пар».

Поскольку кодировка базового набора инструкций не зависит от разрядности архитектуры, то «один и тот же код потенциально может запускаться на различных RISC-V архитектурах, определять разрядность и другие параметры текущей архитектуры, наличие расширений системы инструкций, а потом автоконфигурироваться для целевой среды выполнения» (формально это действительно возможно, в стандартизованных регистрах специальных функций программа может получить информацию о характеристиках процессора на котором выполняется – версии, поддерживаемые наборы инструкций). Обязательным для процессора архитектуры RISC-V является только поддержка базового набора команд – RV32I (рис.1).



***rs1*** — номер регистра в котором находится первый операнд;

***rs2*** — номер регистра в котором находится второй операнд;

***rd*** — номер регистра в который будет записан результат.

Рис. 1 Формат команд базового набора инструкций – RV32I.

Для встраиваемых приложений с ограниченными ресурсами определено подмножество команд RV32E, которое оперирует всего 16-ю регистрами.

В абсолютном большинстве случаев в архитектуре RISC-V используется 32-битная кодировка команд за исключением некоторых расширений, допускающих команды переменной длины (о! а не скрытый ли вы CISC?)) и команд расширения С – т.н. сокращенного формата команд, иди набора сжатых инструкций (аналог наборов команд Thumb у архитектуры ARM). По всей видимости, появление у RISC-архитектур набора сжатых инструкций – «семейное» для всех RISC архитектур - комплексных команд мало, код разрастается, памяти жалко, операции с памятью дорогие по времени..

Дополнительный сжатый 16-разрядный формат команд (C) в основном имеет доступ только к 8 регистрам и, следовательно, может обеспечить плотное кодирование команд, в то время как дополнительные расширения набора команд при желании могут поддерживать гораздо большее пространство регистров.

Дополнительные сжатые 16-разрядные расширения набора команд имеют два младших бита, равных 00, 01 или 10. Стандартные расширения набора команд, закодированные более чем на 32 бита, имеют дополнительные младшие разряды, равные 1, с соглашениями для 48-разрядных и 64-разрядных длин. Длины команд от 80 до 176 бит кодируются с использованием 3-разрядного поля в битах [14:12] задает количество 16-битных слов в дополнение к первым 5 16-битным словам. Кодировка с битами [14:12], установленными на 111, зарезервирована для будущих более длинных кодировок команд – рис.2.



Рис.2 Форматы команд архитектуры RISC-V

**Список наборов команд**

Формирование системы команд для процессоров RISC-V архитектуры можно сравнить с конструктором типа «Лего», или, что наверное еще точнее – стеки и профили сетевых протоколов для сетей малой мощности – в них также выделяется базовая часть и относительно независимые уровни для прикладных задач и для рутинных задач маршрутизации и доставки пакетов.

Да, здесь мы снова видим расхождения с первичной идеологией RISC – что нам хватит и нескольких команд, чтобы сделать всё, но требования приложений, ограничения скорости работы памяти вынуждают набор команд расширять – Таблица 1.

Инновацией, предлагаемой RISC-V можно считать именно модульность системы команд – процессору не обязательно поддерживать ВСЕ наборы команд, достаточно только базового (I), ну и те наборы, которые сделают его более оптимальным для целевого класса задач. Конечно, от процессоров выполненных «в кремнии», сложно ожидать, что заложенные в них производителем наборы команд в точности совпадут с желаемыми. Например - в 32-битных микроконтроллерах и для других встраиваемых применений используется набор RV32EC. В 64-битных процессорах может быть набор групп RV64GC, то же самое в полной записи — RV64IMAFDC. А вот для софт-процессорных ядер возможны интересные перспективы (при наличии ядра с открытым кодом добавить в него своё расширение системы команд относительно не сложно).

Таблица 1

Стандартные расширения наборы системы

|  |  |  |  |
| --- | --- | --- | --- |
| **Сокращение** | **Наименование** | **Версия** | **Статус** |
| **Базовые наборы** | | | |
| **RVWMO** | Базовая модель согласованности памяти | 2 | Ratified |
| **RV32I** | Базовый набор с целочисленными операциями, 32-битный | 2,1 | Ratified |
| **RV64I** | Базовый набор с целочисленными операциями, 64-битный | 2,1 | Ratified |
| **RV32E** | [Базовый набор с целочисленными операциями для встраиваемых систем, 32-битный, 16 регистров](https://ru.wikipedia.org/wiki/%D0%92%D1%81%D1%82%D1%80%D0%B0%D0%B8%D0%B2%D0%B0%D0%B5%D0%BC%D0%B0%D1%8F_%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0) | 1,9 | Draft |
| **RV128I** | Базовый набор с целочисленными операциями, 128-битный | 1,7 | Draft |

|  |  |  |  |
| --- | --- | --- | --- |
| **Часть 1 Стандартные непривилегированные наборы команд** | | | |
| **M** | Целочисленное умножение и деление (Integer Multiplication and Division) | 2 | Ratified |
| **A** | [Атомарные операции (Atomic Instructions)](https://ru.wikipedia.org/wiki/%D0%90%D1%82%D0%BE%D0%BC%D0%B0%D1%80%D0%BD%D1%8B%D0%B5_%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%B8) | 2,1 | Ratified |
| **F** | Арифметические операции с плавающей запятой над числами одинарной точности (Single-Precision Floating-Point) | 2,2 | Ratified |
| **D** | Арифметические операции с плавающей запятой над числами двойной точности (Double-Precision Floating-Point) | 2,2 | Ratified |
| **Q** | Арифметические операции с плавающей запятой над числами четверной точности | 2,2 | Ratified |
| **C** | Сокращённые имена для команд (Compressed Instructions) | 2,2 | Ratified |
| **Counters** | Инструкции для счетчиков производительности и таймеров — наборы **Zicntr** и **Zihpm** | 2 | Draft |
| **L** | Арифметические операции над десятичными числами с плавающей запятой (Decimal Floating-Point) | 0 | Open |
| **B** | Битовые операции (Bit Manipulation) | 0,36 | Open |
| **J** | Двоичная трансляция и поддержка динамической компиляции (Dynamically Translated Languages) | 0 | Open |
| **T** | Транзакционная память (Transactional Memory) | 0 | Open |
| **P** | Короткие SIMD-операции (Packed-SIMD Instructions) | 0,1 | Open |
| **V** | Векторные расширения (Vector Operations) | 1 | Frozen |
| **Zicsr** | Инструкции для работы с контрольными и статусными регистрами (Control and Status Register (CSR) Instructions) | 2 | Ratified |
| **Zifencei** | Инструкции синхронизации потоков команд и данных (Instruction-Fetch Fence) | 2 | Ratified |
| **Zihintpause** | добавляет в инструкцию set architecture (ISA) одну PAUSE инструкцию (закодированную как HINT). | 2 | Ratified |
| **Zihintntl** | Non-temporal locality hints (NTL) — это инструкции, которые указывают, что следующая за ними инструкция доступа к памяти имеет плохую временную локальность ссылок. | 0,2 | Draft |
| **Zam** | Расширение для смещённых атомарных операций (Extension for Misaligned Atomics) | 0,1 | Draft |
| **Zfh** | Расширения для вычислений с плавающей запятой половинной точности | 1 | Ratified |
| **Zfhmin** | Расширения для вычислений с плавающей запятой половинной точности | 1 | Ratified |
| **Zfinx** | Стандартные расширения для чисел с плавающей запятой в целочисленных регистрах | 1 | Ratified |
| **Zdinx** | Стандартные расширения для чисел с плавающей запятой в целочисленных регистрах | 1 | Ratified |
| **Zhinx** | Стандартные расширения для чисел с плавающей запятой в целочисленных регистрах | 1 | Ratified |
| **Zhinxmin** | Стандартные расширения для чисел с плавающей запятой в целочисленных регистрах | 1 | Ratified |
| **Ztso** | Расширение для модели согласованности памяти RVTSO (Extension for Total Store Ordering) | 0,1 | Frozen |
| **G** | **=** **IMAFD Zicsr Zifencei**  Обобщенное/сокращёное обозначение для набора расширений | н/д | н/д |

Таблица 2

Стандартные наборы команд для привилегированных режимов

|  |  |  |  |
| --- | --- | --- | --- |
| **Machine ISA** | Инструкции аппаратного уровня | 1,12 | Ratified |
| **Supervisor ISA** | Инструкции уровня супервизора | 1,12 | Ratified |
| **Svnapot Extension** | это расширение для NAPOT Translation Contiguity. | 1 | Ratified |
| **Svpbmt Extension** | это расширение, которое определяет «Supervisor-mode: page-based memory types» для таких вещей, как некаскабельные страницы или страницы памяти ввода-вывода. | 1 | Ratified |
| **Svinval Extension** | это стандартное расширение для привилегированной архитектуры RISC-V — разделить инструкции SFENCE.VMA, HFENCE.VVMA и HFENCE.GVMA на более мелкие операции по invalidation и упорядочиванию | 1 | Ratified |
| **Hypervisor ISA** | Инструкции уровня гипервизора | 1 | Ratified |

**RV32I**

Кратко рассмотрим блок операций «ядра» архитектуры RISC-V – базовые RV32I операции 32-битной версии архитектуры. Дальнейшее рассмотрение расширений логичнее делать уже исходя из конкретных задач.

Поскольку набор базовый и должен определять весь спектр операций, пусть и минимальный, но покрывающий все необходимые операции для манипуляции с данными, управления потоком выполнения и работе с памятью, то и форматов команд в нем присутствует много – суммарно целых шесть (рис.3, 4).

Операции базового набора работают в основном в разрядной сетке 32 бита и не предусматривают каких-либо аппаратных средств контроля переполнения регистров или генерации исключений.

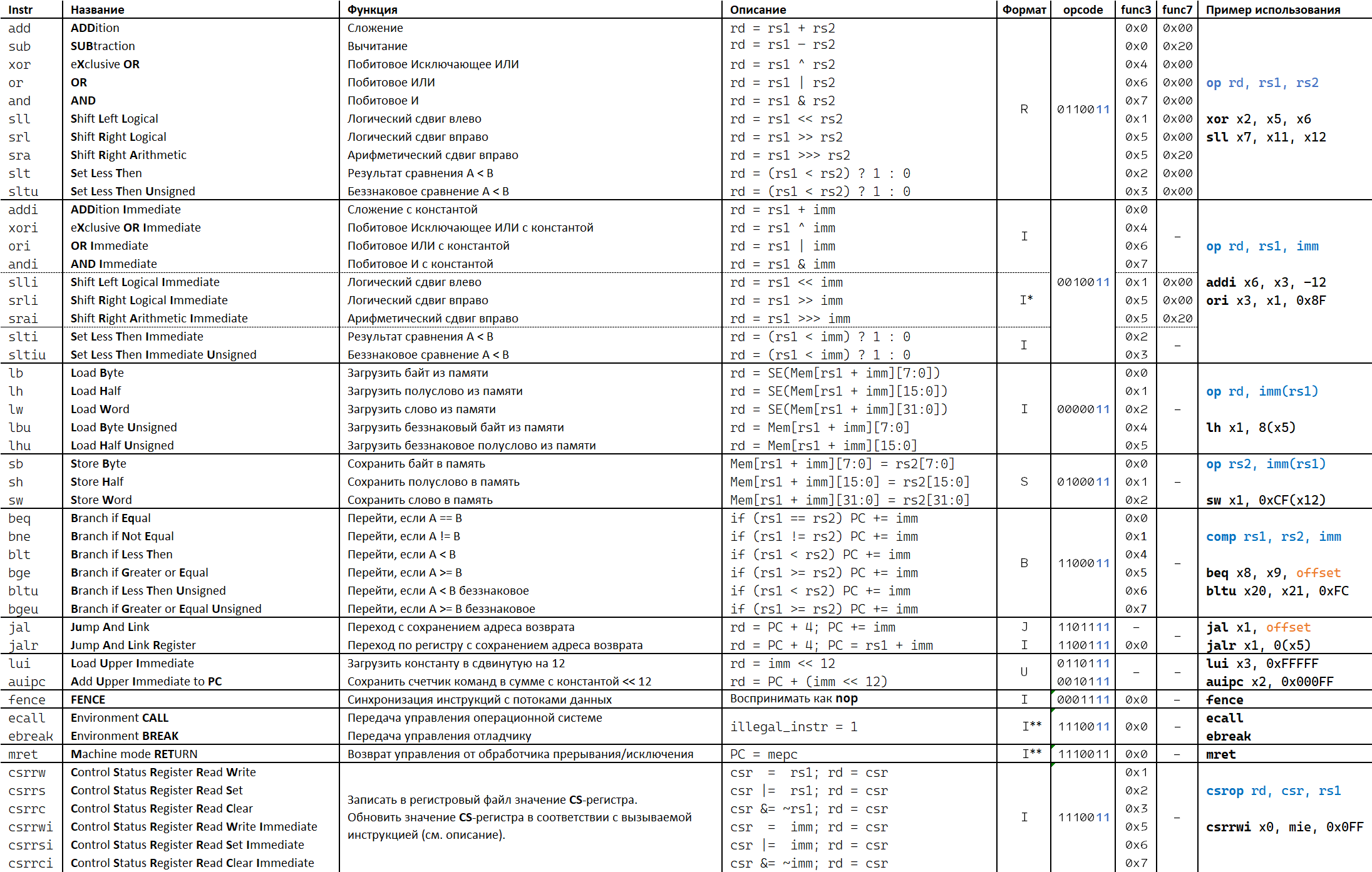


Рис. 3 Базовый набор команд RISC-V (набор I).

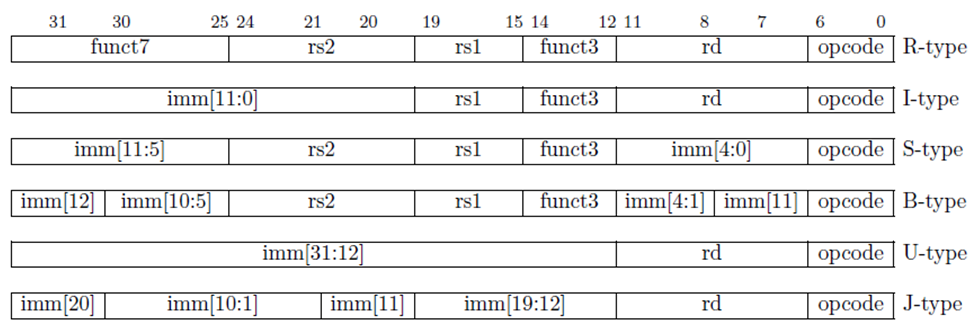
****

Рис.4 Базовые форматы команд RISC-V, показывающие варианты размещения immediate с слове инструкции.

**Операции с регистрами и константами** (immediate/непосредственными значениями) – тип-I.

В эту группу операций входят команды, оперирующие с регистрами и непосредственными операндами – арифметико-логические операции, сравнения с константами, загрузкой данных в регистры.

Таблица 3

Операции с регистрами и константами

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **Imm[11:0]** | **rs1** | **funct3** | **rd** | **opcode** | **I-type** |
| Rd = rs1+imm  NOP encoded as ADDI x0, x0, 0. | imm[11:0] | rs1 | 000 | rd | 0010011 | ADDI |
| Rd= 1 if Rs< imm else 0 | imm[11:0] | rs1 | 010 | rd | 0010011 | SLTI |
| Rd= 1 if Rs< uimm else 0 | imm[11:0] | rs1 | 011 | rd | 0010011 | SLTIU |
| Rd = rs1 XOR imm | imm[11:0] | rs1 | 100 | rd | 0010011 | XORI |
| Rd = rs1 OR imm | imm[11:0] | rs1 | 110 | rd | 0010011 | ORI |
| Rd = rs1 AND imm | imm[11:0] | rs1 | 111 | rd | 0010011 | ANDI |
| Rd = @(Rs1 + imm) 8bit | imm[11:0] | rs1 | 000 | rd | 0000011 | LB |
| Rd = @(Rs1 + imm) 16bit | imm[11:0] | rs1 | 001 | rd | 0000011 | LH |
| Rd = @(Rs1 + imm) 32bit | imm[11:0] | rs1 | 010 | rd | 0000011 | LW |
| Rd = @(Rs1 + imm) 8bit zero ext | imm[11:0] | rs1 | 100 | rd | 0000011 | LBU |
| Rd = @(Rs1 + imm) 16bit zero ext | imm[11:0] | rs1 | 101 | rd | 0000011 | LHU |
| Rd = pc+4 PC= Rs + imm&0xFFE | imm[11:0] | rs1 | 000 | rd | 1100111 | JALR |

**Операции регистр-регистр**

Группа регистровых операций (R-типа). (Собственно, то что RISC-машины должны уметь делать лучще всего.) Интересно, что поля опкодов и коды функций(3битные) операций сложения и вычитания, а также сдвигов и арифметических сдвигов одинаковы, и отличаются только значениями поля funct7. Это может быть полезно как при построении дешифратора команд, так и АЛУ процессорного ядра.

Таблица 4

Регистровые операции

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **funct7** | **rs2** | **rs1** | **funct3** | **rd** | **opcode** | **R-type** |
| Rd = Rs1 + Rs2 | 0000000 | rs2 | rs1 | 000 | rd | 0110011 | ADD |
| Rd = Rs1 - Rs2 | **0100000** | **rs2** | **rs1** | **000** | **rd** | **0110011** | **SUB** |
| Rd = Rs1 + Rs2 | 0000000 | rs2 | rs1 | 001 | rd | 0110011 | SLL |
| Rd= 1 if Rs1 < Rs2 else 0 | 0000000 | rs2 | rs1 | 010 | rd | 0110011 | SLT |
| Rd= 1 if Rs1 < Rs2 else 0(unsign)  SLTU rd, x0, rs2 sets rd to 1  if rs2 is not equal to zero | 0000000 | rs2 | rs1 | 011 | rd | 0110011 | SLTU |
| Rd = Rs1 XOR Rs2 | 0000000 | rs2 | rs1 | 100 | rd | 0110011 | XOR |
| Rd = Rs1 << Rs2 | 0000000 | rs2 | rs1 | 101 | rd | 0110011 | SRL |
| Rd = Rs1 >> Rs2 (sign ext) | **0100000** | **rs2** | **rs1** | **101** | **rd** | **0110011** | **SRA** |
| Rd = Rs1 OR Rs2 | 0000000 | rs2 | rs1 | 110 | rd | 0110011 | OR |
| Rd = Rs1 AND Rs2 | 0000000 | rs2 | rs1 | 111 | rd | 0110011 | AND |
| Rd = Rs1 << imm | 0000000 | shft | rs1 | 001 | rd | 0010011 | SLLI |
| Rd = Rs1 >> imm (0 ext) | 0000000 | shft | rs1 | 101 | rd | 0010011 | SRLI |
| Rd = Rs1 >> imm (sign ext) | **0100000** | **shft** | **rs1** | **101** | **rd** | **0010011** | **SRAI** |

**Команды условных переходов**

Все инструкции перехода используют формат команд B-типа. 12-разрядный B-immediate со знаком (выравнивание по границе 2 байт), и добавляется к текущему PC для получения целевого адреса. Условный диапазон перехода составляет 4 Кб.

Таблица 5

Условные переходы

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **imm[12|10:5]** | **rs2** | **rs1** | **funct3** | **imm[4:1|11]** | **opcode** | **B-type** |
| If (Rs1 == Rs2)  Pc=Pc+imm &0xFFE | imm[12|10:5] | rs2 | rs1 | 000 | imm[4:1|11] | 1100011 | BEQ |
| If (Rs1 != Rs2)  Pc=Pc+imm &0xFFE | imm[12|10:5] | rs2 | rs1 | 001 | imm[4:1|11] | 1100011 | BNE |
| If (Rs1 <Rs2)  Pc=Pc+imm &0xFFE | imm[12|10:5] | rs2 | rs1 | 100 | imm[4:1|11] | 1100011 | BLT |
| If (Rs1 >Rs2)  Pc=Pc+imm &0xFFE | imm[12|10:5] | rs2 | rs1 | 101 | imm[4:1|11] | 1100011 | BGE |
| If (Rs1 <Rs2) --usign  Pc=Pc+imm &0xFFE | imm[12|10:5] | rs2 | rs1 | 110 | imm[4:1|11] | 1100011 | BLTU |
| If (Rs1 >Rs2) --usign  Pc=Pc+imm &0xFFE | imm[12|10:5] | rs2 | rs1 | 111 | imm[4:1|11] | 1100011 | BGEU |

**Инструкции записи в память – S-тип.**

Инструкции SW, SH и SB сохраняют в памяти 32-разрядные, 16-разрядные и 8-разрядные значения из младших разрядов регистра rs2.

Для достижения наилучшей производительности эффективный адрес для всех загрузок и хранилищ должен быть естественным образом выровнен для каждого типа данных (т.е. по четырехбайтовой границе для 32-разрядных обращений и двухбайтовой границе для 16-разрядных обращений) – выравнивание – забота компилятора. Базовый ISA поддерживает несогласованные обращения, но они могут выполняться чрезвычайно медленно в зависимости от реализации. Кроме того, гарантируется естественное выравнивание загрузки и сохранения для выполнения атомарно, в то время как несогласованные загрузки и хранилища могут и не выполняться, и, следовательно, требуется дополнительная синхронизация для обеспечения атомарности.

Таблица 6

Запись в память содержимого регистра

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **imm[11:5]** | **rs2** | **rs1** | **funct3** | **imm[4:0]** | **opcode** | **S-type** |
| [rs1+imm]=rs2(byte) | imm[11:5] | rs2 | rs1 | 000 | imm[4:0] | 0100011 | SB |
| [rs1+imm]=rs2(half) | imm[11:5] | rs2 | rs1 | 001 | imm[4:0] | 0100011 | SH |
| [rs1+imm]=rs2(word) | imm[11:5] | rs2 | rs1 | 010 | imm[4:0] | 0100011 | SW |

**Загрузка «длинных» непосредственных значений в регистры**

Основная «почти» головная боль ISA с фиксированной длиной команд – это загрузка или передача в регистры непосредственные значения (immediate) , особенно это касается «больших» значений, которые могут быть или просто большими числовами константами, или адресами памяти

LUI (load upper immediate) используется для построения 32-разрядных констант и использует формат U-типа. LUI помещает значение U-immediate в верхние 20 бит регистра назначения Rd, заполняя младшие 12 бит нулями. (желаемое 32-битное значение может быть

AUIPC (add upper immediate to pc) используется для построения относительных к PC адресов и использует формат U-type. AUIPC формирует 32-разрядное число из 20-разрядного U-immediate, заполняя младшие 12 бит нулями и добавляя его к значению PC, затем помещает результат в регистр rd.

Таблица 7

Команды U-типа

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **imm[31:12]** | **rd** | **opcode** | **U-type** |
| Rd = imm[31:12][012] | imm[31:12] | rd | 0110111 | LUI |
| Rd=PC+ imm[31:12] [012] | imm[31:12] | rd | 0010111 | AUIPC |

Инструкция AUIPC поддерживает последовательности из двух команд для доступа к произвольным наборам данных с ПК как для передачи потока управления, так и для доступа к данным. Комбинация AUIPC и 12-разрядного immediate в JALR может передавать управление на любой 32-разрядный адрес, относящийся к ПК, в то время как AUIPC плюс 12-разрядный immediate, установленный в обычных инструкциях загрузки или сохранения, могут получать доступ к любому 32-разрядному адресу данных, относящемуся к ПК.

**Безусловные переходы**

Инструкция jump and link (JAL) использует формат J-типа, где J-immediate – знаковое смещение, кратное 2 байтам. Смещение прибавлется к программному счетчику для формирования целевого адреса перехода. Таким образом, переходы могут совершаться в пределах 1 Мб от текущего адреса.

JAL также сохраняет адрес инструкции, следующей за переходом (pc+4), в регистре rd. Стандартное соглашение о вызове программного обеспечения использует x1 в качестве регистра обратного адреса и x5 в качестве альтернативного регистра связи.

Простой безусловный переход без сохранения адреса возврата осуществляется, если в качестве регистра указан x0 – такая операция в ассемблере идет как псевдооперация J

Таблица 8

Безусловные переходы

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Imm[20|10:1|11|19:12]** | **rd** | **opcode** | **J-type** |
| Rd = PC + 4  PC = PC + imm | imm[20|10:1|11|19:12] | rd | 1101111 | JAL |

**Набор инструкций для работы с регистрами специального назначения.**

Формально набор инструкций для работы с регистрами специального назначения относится к расширению **Zicsr**.

Инструкция CSRRW (Atomic Read/Write CSR) атомарно меняет местами значения в регистре CSRS и целочисленном регистре. CSRRW считывает старое значение CSR, нулями-расширяет значение до полной целочисленной разрядности, затем записывает его в целочисленный регистр Rd. Значение из Rs1 записывается в CSR. Если Rd=x0, то команда не должна считывать CSR и не должна вызывать никаких побочных эффектов, которые могут возникнуть при считывании CSR.

Инструкция CSRRS (Атомарные биты чтения и установки в CSR) считывает значение CSR, нулями-расширяет значение до полной целочисленной разрядности и записывает его в целочисленный регистр Rd. Начальное значение в целочисленном регистре Rs1 обрабатывается как битовая маска, которая определяет позиции битов, которые должны быть установлены в CSR. Любой бит, имеющий высокое значение в Rs1, приведет к установке соответствующего бита в CSR, если этот бит CSR доступен для записи. Другие биты в CSR не затрагиваются (хотя CSR могут иметь побочные эффекты при записи).

Инструкция CSRRC (Атомарные биты чтения и очистки в CSR) считывает значение CSR, обнуляет значение до X битов LEN и записывает его в целочисленный регистр rd. Начальное значение в целочисленном регистре Rs1 обрабатывается как битовая маска, которая определяет позиции битов, подлежащие сбросу в CSR. Любой бит, имеющий высокое значение в Rs1, приведет к очистке соответствующего бита в CSR, если этот бит CSR доступен для записи. Другие биты в CSR не затронуты.

Варианты CSRRWI, CSRRSI и CSRRCI аналогичны CSR, CSRRS и CSRRC соответственно, за исключением того, что они обновляют CSR, используя полноразрядное значение, полученное путем расширения нулями 5-разрядного поля без знака uimm[4:0], закодированного в поле rs1, вместо значение из целочисленного регистра.

Для CSRRSI и CSRRCI, если поле uimm[4:0] равно нулю, то эти инструкции не будут выполнять запись в CSR и не вызовут никаких побочных эффектов, которые в противном случае могли бы возникнуть при записи CSR. Для CSRRWI, если rd=x0, то инструкция не должна считывать CSR и не должна вызывать никаких побочных эффектах, которые могут возникнуть при чтении CSR.

RV32I предоставляет ряд 64-разрядных счетчиков пользовательского уровня, доступных только для чтения, которые отображаются в 12-разрядное адресное пространство CSR и доступны в 32-разрядных фрагментах с использованием инструкций CSRRS.

Таблица 9

Команды работы с регистрами специальных функций

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **Работа со спец.регистрами** | | | | | |
|  | Адрес спец.регистра  (12 бит) | Регистр  источник |  | Регистр  Приемник | Опкод |  |
| Rd= CSR  CSR = Rs1 | csr | rs1 | 001 | Rd | 1110011 | CSRRW |
| Rd= CSR  CSR = CSR or Rs1 | csr | rs1 | 010 | Rd | 1110011 | CSRRS |
| Rd= CSR  CSR = CSR and (~Rs1) | csr | rs1 | 011 | Rd | 1110011 | CSRRC |
| Rd= CSR  CSR = zimm | csr | zimm | 101 | Rd | 1110011 | CSRRWI |
| Rd= CSR  CSR = CSR or zimm | csr | zimm | 110 | Rd | 1110011 | CSRRSI |
| Rd= CSR  CSR = CSR and (~zimm) | csr | zimm | 111 | rd | 1110011 | CSRRCI |

Примеры регистров специального назначения.

Базовый 64-разрядный счетчик, который на практике никогда не должен переполняться. Следующая кодовая последовательность преобразует действительное значение 64-разрядного счетчика циклов в x3:x2, даже если счетчик переключается между считыванием своей верхней и нижней половин.

Таблица 10

Таймеры и счетчики

|  |  |  |  |
| --- | --- | --- | --- |
| 0xC00 | Read-only | cycle | Cycle counter |
| 0xC01 | Read-only | time | Timer for |
| 0xC02 | Read-only | instret | Instructions-retired counter |
| 0xC80 | Read-only | cycleh | Upper 32 |
| 0xC81 | Read-only | timeh | Upper 32 |
| 0xC82 | Read-only | instreth | Upper 32 |

Примеры инструкций работы с регистрами специальных функций

Псевдоинструкция RDCYCLE считывает младшие биты XLEN цикла CSR, который содержит подсчет количества тактовых циклов, выполненных процессорным ядром, на котором запущен hart, с произвольного времени запуска в прошлом. RDCYCLEH - это инструкция только для RV32I, которая считывает биты 63{32 того же счетчика циклов. Базовый 64-разрядный счетчик на практике никогда не должен перегружаться. Скорость скорость, с которой увеличивается счетчик циклов, будет зависеть от реализации и операционной среды.

Среда выполнения должна предоставлять средства для определения текущей скорости (циклов в секунду) с которой увеличивается счетчик циклов.

Псевдоинструкция RDTIME считывает младшие биты XLEN временного CSR, который подсчитывает реальное время часов, прошедшее с произвольного времени начала в прошлом. RDTIMEH - это инструкция только для RV32I, которая считывает биты 63-32 того же счетчика реального времени. 64-разрядный счетчик на практике врятли когда-то переполнится).

(Среда выполнения должна обеспечивать средства определения периода счетчика реального времени (секунды/тик). Период должен быть постоянным. Часы реального времени всех устройств в одном пользовательском приложении должны быть синхронизированы с точностью до одного тика часов реального времени. Среда должна предоставлять средства для определения точности часов.)

Псевдоинструкция RDINSTRET считывает младшие ксилольные биты интернет-CSR, которая подсчитывает количество инструкций, удаленных этим hart из некоторой произвольной начальной точки в прошлом. RDINSTRETH - это инструкция только для RV32I, которая считывает биты 63-32 из того же счетчика команд.

**Команды системных вызовов и прерываний**

Команда вызова ECALL используется для отправки запроса к поддерживающей среде выполнения, которой обычно является операционная система. ABI для системы определяет, как передаются параметры для запроса среды, но обычно они находятся в определенных местах в файле целочисленного регистра.

Команда EBREAK используется отладчиками для передачи управления обратно в среду отладки.

Таблица 11

Системные прерывания

|  |  |
| --- | --- |
| опкод | команды |
| 1110011 | ECALL |
| 1110011 | EBREAK |

Базовый RISC-V ISA поддерживает несколько параллельных потоков выполнения в пределах одного адресного пространства пользователя. Каждый аппаратный поток RISC-V, или hart, имеет свое собственное состояние пользовательского регистра и счетчик программ и выполняет независимый последовательный поток команд. Среда выполнения будет определять, как создаются RISC-V hart и как ими управляют. Интерфейсы RISC-V могут взаимодействовать и синхронизироваться с другими hart (далее для простоты будем использовать «харт» или «поток») либо посредством вызовов среды выполнения, которые задокументированы отдельно в спецификации для каждой среды выполнения, либо напрямую через систему общей памяти.

RISC-V харты также могут взаимодействовать с устройствами ввода-вывода и косвенно друг с другом посредством загрузки и сохранения в части адресного пространства, назначенного для ввода-вывода.

В базовом RISC-V ISA каждый RISC-V hart наблюдает за своими собственными операциями с памятью, как если бы они выполнялись последовательно в программном порядке. RISC-V имеет упрощенную модель памяти между потоками, требующую явной инструкции FENCE, чтобы гарантировать упорядочение операций с памятью из разных потоков RISC-V (в крайнем случае очень и очень аккуратного обращения к памяти в плане одновременного обращения к одним и тем же областям, что в общем-то не есть хорошо).

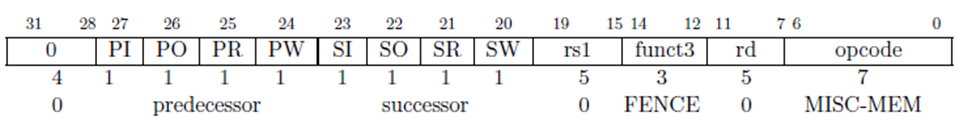


Рис. 5 Поля команды FENCE.

Команда FENCE используется для упорядочивания операций ввода-вывода устройства и доступа к памяти в соответствии с другими RISC-V hart и внешними устройствами или сопроцессорами. Любая комбинация ввода устройства (I), вывода устройства (O), операции чтения из памяти (R) и записи в память (W) могут быть упорядочены относительно любой их комбинации.

Неофициально, никакой другой RISC-V харт или внешнее устройство не может наблюдать какую-либо операцию в наборе-преемнике, следующем за FENCE, перед любой операцией в наборе-предшественнике.

Среда выполнения должна определять, какие операции ввода-вывода возможны, и, в частности, какие инструкции загрузки и сохранения могут обрабатываться и упорядочиваться как операции ввода-вывода соответственно, а не как операции чтения и записи в память. Например, доступ к устройствам ввода-вывода, отображенным в память, обычно осуществляется с помощью некэшированных загрузок и хранилищ, упорядоченных с использованием битов ввода-вывода, а не битов R и W. Расширения набора команд могут также описывать новые инструкции ввода-вывода сопроцессора, которые также будут упорядочены с использованием битов ввода-вывода в FENCE.

В архитектуре RISC-V реализована упрощенная модель памяти, чтобы обеспечить высокую производительность за счет простых машинных реализаций, однако полностью упрощенная модель памяти слишком слаба для поддержки моделей памяти на языках программирования, и поэтому модель памяти ужесточается.

Упрощенная модель памяти также наиболее совместима с вероятными будущими расширениями сопроцессора или ускорителя. В архитектуре отделяется упорядочение ввода-вывода от упорядочения R/W памяти, чтобы избежать ненужной сериализации внутри жесткого диска драйвера устройства, а также для поддержки альтернативных путей, не связанных с памятью, для управления добавленными сопроцессорами или устройствами ввода-вывода. Простые реализации могут дополнительно игнорировать поля "предшественник" и "преемник" и всегда выполнять консервативное ограничение для всех операций.

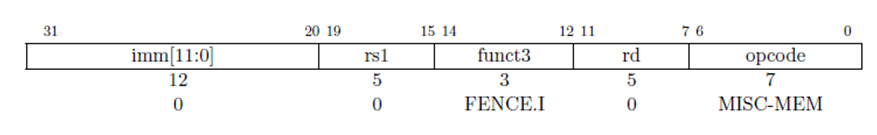


Рис. 5 Поля команды FENCE.I.

FENCE.I используется для синхронизации потоков инструкций и данных. RISC-V не гарантирует, что сохраненные в памяти инструкции будут доступны для выборки команд на том же RISC-V hart до тех пор, пока не будет выполнено FENCE.I.

FENCE.I гарантирует только то, что при последующей выборке команды на RISC-V hart будут видны все предыдущие хранилища данных, уже видимые для тот же RISC-V hart.

FENCE.I не гарантирует, что при выборке команд другими RISC-V интерфейсами будут соблюдаться локальные хранилища данных в многопроцессорной системе. Чтобы сделать хранилище в памяти команд видимым для всех RISC-V-устройств, записывающая часть должна выполнить забор данных, прежде чем запрашивать, чтобы все удаленные RISC-V-устройства выполнили FENCE.I.

Инструкция FENCE.I была разработана для поддержки широкого спектра реализаций. Простая реализация может использовать локальный кэш команд и конвейер команд при выполнении FENCE.I. Более сложная реализация может отслеживать кэш команд (данных) при каждом пропуске кэша данных (инструкций) или использовать инклюзивный единый частный кэш L2 для аннулирования строк из основного кэша команд, когда они записываются инструкцией локального хранилища.

Если кэши инструкций и данных поддерживаются согласованными таким образом, то на границе необходимо использовать только FENCE.I.

JIT-компиляторы могут генерировать большое количество инструкций перед одним FENCE.I, и уменьшите любые накладные расходы на отслеживание/аннулирование кэша команд путем записи переведенных инструкций в области памяти, которые, как известно, не находятся в I-кэше.

**Литература**

1. The RISC-V Instruction Set Manual. Volume I: User-Level ISA Document Version 2.2 // Editors: Andrew Waterman1, Krste Asanov
2. RISCVGreenCardv // https://www.cl.cam.ac.uk/teaching/1617/ECAD+Arch/files/docs/RISCVGreenCardv8-20151013.pdf
3. Сара Л. Харрис, Дэвид Харрис. Цифровая схемотехника и архитектура компьютера: RISC-V / пер. с англ. В. С. Яценкова, А. Ю. Романова; под ред. А. Ю. Романова. – М.: ДМК Пресс, 2021. – 810 с.
4. RISC-V Assembly Language Programmer Manual Part I developed by: SHAKTI Development Team @ iitm ’20 shakti.org.in
5. Стивен Смит. Программирование на языке ассемблера RISC-V / пер. с англ. А. В. Логунова; под науч. ред. А. Ю. Романова. – М.: ДМК Пресс, 2025. – 276 с.