***Развитие архитектур процессоров на основе концепции RISC.***

В концепции RISC инженеры устранили главный недостаток CISC-архитектур: использование большого числа функционально сложных команд. В RISC-архитектурах используется небольшое число команд фиксированной длины и при этом увеличивается число регистров, чтобы иметь большее пространство для работы с данными и реже обращаться к памяти. С точки зрения программирования стало сложнее, поскольку из-за упрощения команд одно и то же действие в RISC требует больше инструкций, чем в CISC. Однако, с другой стороны единая длина для всех команд и минимизация форматов позволили существенно снизить требования к аппаратной начинке процессора. Это в свою очередь привело к упрощению разработки процессора, его удешевлению и пониженному энергопотреблению.

Преимущества концепции RISC наиболее активно стали использовать в британской компании ARM Limited (Advanced RISC Machine). Они внесли различные усовершенствования в эту концепцию и уже в 1985 году представили свой первый процессор ARM1, который стал самым простым RISC-процессором в мире, состоящим всего лишь из 30000 транзисторов. Впоследствие под ARM начали понимать авторскую **лицензируемую а**рхитектуру RISC-процессоров .

Поскольку изначально ARM задумывалась как усовершенствование архитектуры RISC, то для ранних версий ARM-процессоров были характерны некоторые общие черты:

* фиксированный размер команд;
* увеличение числа регистров общего назначения;
* специализированные команды чтения или записи для операций с медленной оперативной памятью.

Однако постепенно разработчики ARM вносили все больше изменений и модификаций, отклоняясь от классической концепции RISC. Например, число доступных программисту регистров в ранних ARM-процессорах было всего 16-31, что совсем мало. Также в некоторых случаях одна команда в ARM выполняла не только операцию над данными типа сложения, но и другие преобразования (сдвиг, инверсия и т.п.). Это позволяло частично уменьшить размер кода, но за счет увеличения сложности самих команд. Еще одна важная особенность ARM — это развитый набор видов адресации, который не предусмотрен в концепции RISC архитектуры.

В результате получается, что в архитектуре ARM граница между архитектурами х86 и RISC становится все менее явной. С одной стороны,

в x86-процессорах уже используется разбиение длинных инструкций на более мелкие - фактически, в них применяется RISC-ядро, что позволяет повысить их производительность. С другой стороны, функциональность ARM процессоров постепенно приближается к x86, но введение множества усовершенствований в концепцию RISC приводит к большему размеру кристалла и тепловыделению. Тем не менее ориентация на концепцию RISC сделали ARM наилучшим вариантом для процессоров мобильной электроники.

**Архитектура RISC-V**

Еще одной из существенных проблем ARM-процессоров является то, что использование архитектуры ARM в собственных разработках требует дорогого лицензирования (на уровне 1 миллиона долларов).

Над этой проблемой задумались в отделении информатики Калифор-нийского университета в Беркли и в 2010 году группа инженеров под руководством Крсте Асановича и Дэвида Паттерсона представили процессор RISC-V, главная отличительная черта которого — полностью открытая архитектура, которую можно использовать абсолютно бесплатно. Кроме того, RISC-V должна предусматривать единую систему команд для всех типов ЭВМ от микроконтроллеров до высокопроизводительных машин.

То есть RISC-V — это улучшенная версия классической архитектуры RISC, созданной в Беркли несколько десятилетий назад. Главное достоинство RISC-V в том, что ее архитектура подходит для самых разнообразных вычислительных задач. Базовая версия RISC-V содержит минимальный набор инструкций — всего 47, но при необходимости его можно расширять для решения своей задачи. То есть RISC-V - это модульная структура или базовый конструктор, на основе которого можно проектировать собственные системы.

В результате RISC-V позволяет достигать целый ряд практических целей:

1. **Повышение энергоэффективности и производительности**, уменьшение числа необходимых транзисторов, удешевление разработки и поддержки процессорных ядер - в минимальном варианте разработчики обязаны реализовывать лишь относительно небольшой набор инструкций для получения полноценно работающей системы. После чего они могут сосредоточиться на интересующих именно их областях применения.

2. **Модульность и расширяемость**. Минимальные наборы команд существуют и в других процессорах. Основная проблема существующих систем команд в том, что небольшой набор команд обычно не универсален. Он ориентирован на области, требующие специфической аппаратной функциональности (высокопроизводительные вычисления, многопоточность или графика). RISC-V решает эту проблему благодаря модульной структуре системы команд. Разработчики ЭВМ могут реализовывать существующие расширения (например, RVF32 для работы с плавающей точкой), либо добавлять свои собственные – система команд спроектирована с прицелом на расширение. Это в корне отличает RISC-V, например, от ARM, где не только расширяться проблематично, но и сама компания ARM запрещает подобные расширения по условиям лицензии.

3. **Удешевление разработки** различных систем за счёт создания единой **открытой экосистемы**, включающей: компиляторы, операционные системы, драйверы, периферию. Наличие **открытых** репозиториев, в которых многое уже реализовано и за счёт многолетней поддержки сообщества доведено до высокого качества, упрощает разработку новых систем.

4. **Обратная совместимость ПО** – программы, разработанные для процессоров с поддержкой меньшего количества расширений команд, должны работать на процессорах той же системы команд, но с большим количеством реализованных расширений **без перекомпиляции**. В случае коммерческих систем с наборами команд вроде x64 (Intel) или ARM это сделать существенно труднее.

5. **Безопасность.** Если в некотором RISC-V процессоре обнаружена уязвимость, вы можете легко заменить его на другой RISC-V процессор, пусть и не такой эффективный, но зато без закладок потенциального противника. При этом вы сможете запустить на нём всё существующее ПО с минимальными затратами усилий и времени.

RISC-V и ARM-архитектуры основаны на одном подходе — RISC и они обе подходят для микроэлектроники. Однако ARM по ходу своего развития претерпела множество изменений, создав целую линейку готовых ядер для разных нужд. Сравнение RISC-V и ARM-архитектур приведено в таблице 1.

Таблица 1

|  | **ARM** | **RISC-V** |
| --- | --- | --- |
| Бизнес-модель | Архитектура ARM основана на модели патентованной интел-лектуальной собственности. Это значит, что за использование процессора ARM нужно платить лицензионный сбор. | Архитектура RISC-V свободная и открытая, любой может использовать ее для исследований и разработки без внесения лицензионных платежей. |
| Экосистема | ARM отличается обширным сообществом и большим количеством библиотек с поддерживаемым ПО. | RISC-V – сравнительно новая технология со стремительно растущим сообществом и техподдержкой. |
| Энергопотребление | [По результатам тестов](https://www.slideshare.net/KarthiSugumar/architecture-exploration-of-riscv-processor-and-comparison-with-arm-cortexa53) среднее энергопотребление ARM составляет примерно 40 мВт. | Те же тесты показали, что в определенных случаях RISC-V потребляет в три раза меньше энергии, чем ARM. |
| Кастомизация (приближение к потребностям пользователя) | В ARM недавно появились стандартизированные инструкции для специальных сопроцессоров, но набор инструкций фиксирован. | RISC-V включает в себя набор базовых инструкций и позволяет инженерам добавлять любые стандартизированные расширения, необходимые для их задач. |

В 2015 г. был основан Фонд RISC-V для продвижения новой архитектуры среди коммерческих пользователей. Проект стремительно завоевал популярность у исследователей и ИТ-компаний по всему миру, и сейчас эта некоммерческая организация насчитывает более двух тысяч компаний- партнеров в лице Google, IBM, NVIDIA, Western Digital и других крупных компаний. В 2018 г. организация объявила о сотрудничестве с Linux Foundation, а в 2022 г. компания  [Intel заявила о присоединении к RISC-V International и инвестировании одного миллиарда долл. в экосистему RISC-V](https://www.networkworld.com/article/3649114/intel-partners-with-risc-v-invests-1b-in-foundry-ecosystem.html" \t "_blank).

В РФ одним из лидеров экосистемы RISC-V и основным разработчиком микропроцессорных ядер на основе RISC-V является петербургская компания Syntacore, представившая открытое ядро SCR1 в 2019 г. В настоящее время партнер Ростеха компания Yadro выкупила 51-процентную долю у компании [Syntacore](https://syntacore.com/" \t "_blank) и является инициатором разработки МП-ядер на основе RISC-V. Ориентация на RISC-V дает российским разработчикам аппаратного обеспечения возможность снизить зависимость от зарубежных поставщиков и создавать технологические решения в соответствии со стратегией импортозамещения.

# Система команд процессоров RISC-V.

В архитектуре RISC-V имеется обязательное для реализации базовое подмножество в количестве 47 команд и несколько стандартных опциональных расширений. В базовый набор входят: минимальный набор команд арифметических/битовых операций на регистрах, команд для выполнения операций с памятью (load/store), команд условной и безусловной передачи управления, а также небольшое число служебных инструкций (см. таблицу далее). Команды базового набора имеют длину 32 бита с выравниванием на границу 32-битного слова.

Операции условных переходов (ветвления) не используют каких-либо общих флагов, как результатов ранее выполненных операций сравнения, а непосредственно сравнивают свои регистровые операнды. Базис операций сравнения минимален, а для поддержки комплементарных операций операнды просто меняются местами.

Базовое подмножество команд использует следующий набор регистров: специальный регистр x0 (zero), содержащий значение 0, 31 целочисленный регистр общего назначения (x1—x31), регистр счётчика команд PC (используется только косвенно), а также множество CSR (Control and Status Registers), может быть адресовано до 4096 CSR).

|  |  |  |
| --- | --- | --- |
| Регистр(ы) | Наименование | Описание назначения |
| х0 | zero | Константа 0 |
| х1 | ra | Адрес возврата |
| х2 | sp | Указатель стека |
| х3 | gp | Глобальный указатедь |
| х4 | tp | Указатель потока |
| х5-х7 | t0-t2 | Временные регистры |
| х8 | s0/fp | Сохраняемый регистр / Указатель фрейма |
| х9 | s1 | Сохраняемый регистр |
| х10-х11 | a0-a1 | Аргументы функции / Возвращаемые значения |
| х12-х17 | a2-a7 | Аргументы функции |
| х18-х27 | s2-s11 | Сохраняемые регистры |
| х28-х31 | t3-t6 | Временные регистры |

Регистр, x0 имеет специальное назначение. Вне зависимости от того, какое значение вы в него записываете, при чтении из этого регистра вы всегда получите 0. Псевдонимом регистра x0 является zero.

Псевдонимы других регистров:

* ra return address (адрес возврата). Используется для записи адреса возврата перед вызовом подпрограммы.
* sp stack pointer (указатель стека).
* gp global pointer (глобальный указатель).
* tp thread pointer (указатель потока)
* t0 - t6 temporary registers (регистры временных переменных). Подпрограммы не обязаны их сохранять.
* s0 - s11 saved registers (сохраняемые регистры). Подпрограммы обязаны сохранять их состояние.
* a0 - a7 function arguments (аргументы функций). Перед вызовом подпрограммы вы передаёте аргументы в эти регистры.

Вся арифметико-логическая обработка данных может производиться только над регистрами, при этом в основном формате регистр-регистр используются два регистра-источника операндов Rsrc1 и Rsrc2 (далее rs1 и rs2) и регистр результата Rdest (далее rd). Для использования ячеек основной памяти применяются инструкции загрузки (Load) данных из ячеек памяти в РОНы и выгрузки (Store) данных из РОН в ячейки памяти.

Архитектура использует только [little-endian](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D1%80%D1%8F%D0%B4%D0%BE%D0%BA_%D0%B1%D0%B0%D0%B9%D1%82%D0%BE%D0%B2" \l "%D0%9F%D0%BE%D1%80%D1%8F%D0%B4%D0%BE%D0%BA_%D0%BE%D1%82_%D0%BC%D0%BB%D0%B0%D0%B4%D1%88%D0%B5%D0%B3%D0%BE_%D0%BA_%D1%81%D1%82%D0%B0%D1%80%D1%88%D0%B5%D0%BC%D1%83" \o "Порядок байтов) модель — первый байт операнда в памяти соответствует наименее значащим битам значений регистрового операнда (аналогично архитектуре х86).

RISC-V применяет память с побайтовой адресацией. Это значит, что каждый байт памяти имеет уникальный адрес. Поскольку 32-битное слово состоит из четырех 8-битных байтов, то адрес каждого слова (word address) кратен 4. Старший байт (most significant byte,MSB) находится слева, а младший байт (least significant byte, LSB) – справа.



К **базовым вычислительным инструкциям** относятся:

- арифметические add, sub ;

- сравнения slt, sltu (знаковое и беззнаковое);

- логические and, or, xor ;

- сдвиговые sll, srl, sra (логические сдвиги влево/вправо и арифметический сдвиг вправо).

Примеры вычислительных инструкций:

sub x3, x1, x2 # x3 = x1 - x2

slt x3, x1, x2 # if x1 < x2 then x3 = 1 else x3 = 0

and x3, x1, x2 # x3 = x1 & x2

sll x3, x1, x2 # x3 = x1 << x2

Вычислительные инструкции также могут использоваться в формате регистр- непосредственное данное (immidiate), при котором один операнд размещается в регистре, а второй – константа, закодированная в самой инструкции:

Oper rd, rs1, const

Например:

addi x3, x1, 7 # x3 = x1 + 7

slli x3, x1, 3 # x3 = x1 << 3

Операции умножения, деления и вычисления остатка не входят в базовый набор инструкций, а выделены в отдельное расширение (M — Multiply extension), рассматриваемое ниже.

К **базовым инструкциям управления** программой относятся:

1. инструкции условного перехода

beq (=), bne (!=), blt (<), bge (≥), bltu (<u), bgeu(≥u)

Две последних - беззнаковые.

Формат: Oper rs1, rs2, label

Примеры:

1) blt x1, x2, lab1 # if x1<x2 then PC = PC+lab1

2) для реализации сложного ветвления типа

if x1<x2 then x3=x1+2 else x3=x2+4 следует использовать код

bge x1, x2, else

addi x3, x1, 2

beq x0, x0, end

else: addi x3, x2, 4

end: продолжение

1. инструкции безусловного перехода

- безусловный переход с сохранением адреса возврата

jal ra, label (jump and link)

Здесь место перехода определяется меткой label, которая задается 20-битным смещением относительно РС (адреса текущей инструкции),

(т.е. переход возможен только по 20-битному адресу);

ra - регистр для сохранения адреса возврата (адреса следующей после jal команды); в качестве ra в архитектуре RISC-V используется РОН x1;

- безусловный переход по значению из регистра с сохранением адреса возврата

jalr ra, offset(rs1)

Здесь место перехода определяется значением из регистра rs1 плюс заданное 12-битное смещение offset; в этом случае можно перейти по любому 32-битному адресу.

1. К базовым инструкциям операций с памятьюотносятся:
2. команда чтения из памяти в регистр (Load)

LB/H/W rd, imm (rs1) # rd = M[rs1+imm]

1. команда записи из регистра в память (Store)

SB/H/W rs2, imm (rs1) # M[rs1+imm] = rs2

Здесь в зависимости от размера данного Byte(8) /Half(16) /Word(32) используются соответствующие версии операций. Для доступа к памяти задается базовый адрес, размещаемый в регистре rs1, и смещение в виде непосредственного значения (imm). Базовый адрес нужен для обращения ко всему объему памяти: в случае 32-битного адреса - к 4Гб памяти.

Пример вычисления выражения a = b + c для данных в памяти.

Память

|  |  |
| --- | --- |
| Адрес | Значение |
| 0х0 |  |
| 0х4 |  |
| 0х8 | b |
| 0хС | c |
| 0х10 |  |
| 0х14 |  |
| 0х18 | a |
| 0х1С |  |
| 0х20 |  |

x1 = Load (0x8)

x2 = Load (0xC)

Add x3, x1, x2

Store (0x18), x3

Реальные команды будут иметь вид:

LW x1, 0x8(x0)

LW x2, 0xC(x0)

Add x3, x1, x2

SW x3, 0x18(x0)

Здесь базовый адрес – это х0 (равен 0).

Полный состав базового набора команд процессора RISC-V может быть представлен таблицей 2.

Таблица 2.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Обозначение  в ассемблере | Наименование | Тип формата | Описание | Примеча-ние |
| add | ADD | R | rd = rs1 + rs2 |  |
| sub | SUB | R | rd = rs1 - rs2 |  |
| xor | XOR | R | rd = rs1 ⊕ rs2 |  |
| or | OR | R | rd = rs1 ! rs2 |  |
| and | AND | R | rd = rs1 & rs2 |  |
| sll | Shift Left Logical | R | rd = rs1 << rs2 |  |
| srl | Shift Right Logical | R | rd = rs1 >> rs2 |  |
| sra | Shift Right Arith | R | rd = rs1 >> rs2 | msb-extends |
| slt | Set Less Than | R | rd = (rs1 < rs2)?1:0 |  |
| sltu | Set Less Than (U) | R | rd = (rs1 < rs2)?1:0 | zero-extends |
| addi | ADD Immediate | I | rd = rs1 + imm |  |
| xori | XOR Immediate | I | rd = rs1 ⊕ imm |  |
| ori | OR Immediate | I | rd = rs1 ! imm |  |
| andi | AND Immediate | I | rd = rs1 & imm |  |
| slli | Shift Left Logical Imm | I | rd = rs1 << imm[0:4] |  |
| srli | Shift Right Logical Imm | I | rd = rs1 >> imm[0:4] |  |
| srai | Shift Right Arith Imm | I | rd = rs1 >> imm[0:4] | msb-extends |
| slti | Set Less Than Imm | I | rd = (rs1 < imm)?1:0 |  |
| sltui | Set Less Than Imm (U) | I | rd = (rs1 < imm)?1:0 | zero-extends |
| lb | Load Byte | I | rd = M[rs1+imm][0:7] |  |
| lh | Load Half | I | rd = M[rs1+imm][0:15] |  |
| lw | Load Word | I | rd = M[rs1+imm][0:31] |  |
| lbu | Load Byte (U) | I | rd = M[rs1+imm][0:7] | zero-extends |
| lhu | Load Half (U) | I | rd = M[rs1+imm][0:15] | zero-extends |
| sb | Store Byte | S | M[rs1+imm][0:7]=rs2[0:7] |  |
| sh | Store Half | S | M[rs1+imm][0:15]=rs2[0:15] |  |
| sw | Store Word | S | M[rs1+imm][0:31]=rs2[0:31] |  |
| beq | Branch = | B | if(rs1 = rs2) PC += imm |  |
| bne | Branch != | B | if(rs1 != rs2) PC += imm |  |
| blt | Branch < | B | if(rs1 < rs2) PC += imm |  |
| bge | Branch ≥ | B | if(rs1≥ rs2) PC += imm |  |
| bltu | Branch < (U) | B | if(rs1 < rs2) PC += imm | zero-extends |
| bgeu | Branch ≥ (U) | B | if(rs1 ≥ rs2) PC += imm | zero-extends |
| jal | Jump And Link | J | rd = PC+4; PC += imm |  |
| jalr | Jump And Link Reg | I | rd = PC+4; PC = rs1 + imm |  |
| lui | Load Upper Imm | U | rd = imm << 12 |  |
| auipc | Add Upper Imm to PC | U | rd = PC + (imm << 12) |  |
| ecall | Environment Call | I | Transfer control to OS |  |
| ebreak | Environment Break | I | Transfer control to debugger |  |

**Форматы машинных команд**

Рассмотренные базовые инструкции процессора RISC-V для 32-битной архитектуры имеют форматы, представленные в табл. 3. Признаки формата 32-битной машинной команды: младшие биты всегда «11» и 2-4 биты≠«111» )

Таблица 3

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Тип | 31 … 25 | | 24 … 20 | 19 … 15 | 14 … 12 | 11 … 7 | | 6 … 2 | 1 | 0 |
| **Регистр/регистр** | funct7 | | rs2 | rs1 | funct3 | rd | | opcode | 1 | 1 |
| **С операндом** | ± | imm[10:0] | | rs1 | funct3 | rd | | opcode | 1 | 1 |
| **С длинным операндом** | ± | imm[30:12] | | | | rd | | opcode | 1 | 1 |
| **Сохранение** | ± | imm[10:5] | rs2 | rs1 | funct3 | imm[4:0] | | opcode | 1 | 1 |
| **Ветвление** | ± | imm[10:5] | rs2 | rs1 | funct3 | imm[4:1] | [11] | opcode | 1 | 1 |
| **Переход** | ± | imm[10:1] | [11] | imm[19:12] | | rd | | opcode | 1 | 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
* opcode (код операции) – частично задают тип формата и операцию
* funct7 + funct3 – совместно с opcode определяют вид операции

Для встраиваемых применений может использоваться вариант архитектуры RV32E (Embedded) с сокращённым набором регистров общего назначения (первые 16). Уменьшение количества регистров позволяет не только экономить аппаратные ресурсы, но и сократить затраты памяти и времени на сохранение/восстановление регистров при переключениях контекста.

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, Для пары инструкций сохранения/загрузки регистра операнд в памяти определяется размером регистра выбранной архитектуры, а не кодировкой инструкции (код инструкции один и тот же для RV32I, RV64I и RV128I, но размер операндов 4, 8 и 16 байт соответственно.

Инструкции базового набора имеют длину 32 бита с выравниванием на границу 32-битного слова, но в общем формате предусмотрены инструкции различной длины (стандартно — от 16 до 192 бит с шагом в 16 бит) с выравниванием на границу 16-битного слова. Полная длина инструкции декодируется унифицированным способом из её первого 16-битного слова.

Расширения базового набора команд.

Спецификацией RISC-V предусмотрено несколько расширений наборов команд, представленных в табл.4.

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.

**Список наборов команд**  Таблица 4

|  |  |  |  |
| --- | --- | --- | --- |
| **Сокращение** | **Наименование** | **Версия** | **Статус** |
| **Базовые наборы** | | | |
| **RV32I** | Базовый набор с целочисленными операциями, 32-битный | 2.0 | Frozen |
| **RV32E** | Базовый набор с целочисленными операциями для [встраиваемых систем](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), 32-битный, 16 регистров | 1.9 | Open |
| **RV64I** | Базовый набор с целочисленными операциями, 64-битный | 2.0 | Frozen |
| **RV128I** | Базовый набор с целочисленными операциями, 128-битный | 1.7 | Open |
| **Стандартные расширеные наборы** | | | |
| **M** | Целочисленное умножение и деление (Integer Multiplication and Division) | 2.0 | Frozen |
| **A** | [Атомарные операции](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) (Atomic Instructions) | 2.0 | Frozen |
| **F** | Арифметические операции с плавающей точкой над числами одинарной точности (Single-Precision Floating-Point) | 2.0 | Frozen |
| **D** | Арифметические операции с плавающей точкой над числами двойной точности (Double-Precision Floating-Point) | 2.0 | Frozen |
| **Q** | Арифметические операции с плавающей точкой над числами четверной точности | 2.0 | Frozen |
| **C** | Сокращённые имена для команд (Compressed Instructions) | 2.0 | Frozen |
| **B** | [Битовые операции](https://ru.wikipedia.org/wiki/%D0%91%D0%B8%D1%82%D0%BE%D0%B2%D1%8B%D0%B5_%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D0%B8) (Bit Manipulation) | 0.36 | Open |
| **J** | [Двоичная трансляция](https://ru.wikipedia.org/wiki/%D0%94%D0%B2%D0%BE%D0%B8%D1%87%D0%BD%D0%B0%D1%8F_%D1%82%D1%80%D0%B0%D0%BD%D1%81%D0%BB%D1%8F%D1%86%D0%B8%D1%8F) и поддержка [динамической компиляции](https://ru.wikipedia.org/wiki/JIT-%D0%BA%D0%BE%D0%BC%D0%BF%D0%B8%D0%BB%D1%8F%D1%86%D0%B8%D1%8F) (Dynamically Translated Languages) | 0.0 | Open |
| **T** | [Транзакционная память](https://ru.wikipedia.org/wiki/%D0%A2%D1%80%D0%B0%D0%BD%D0%B7%D0%B0%D0%BA%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C) (Transactional Memory) | 0.0 | Open |
| **P** | Короткие [SIMD](https://ru.wikipedia.org/wiki/SIMD)-операции (Packed-SIMD Instructions) | 0.1 | Open |
| **V** | [Векторные расширения](https://ru.wikipedia.org/wiki/%D0%92%D0%B5%D0%BA%D1%82%D0%BE%D1%80%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F_(%D0%BF%D0%B0%D1%80%D0%B0%D0%BB%D0%BB%D0%B5%D0%BB%D1%8C%D0%BD%D1%8B%D0%B5_%D0%B2%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F)) (Vector Operations) | 0.2 | Open |
| **N** | Инструкции [прерывания](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B5%D1%80%D1%8B%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) (User-Level Interrupts) | 1.1 | Open |

Здесь статус Frozen соответствует наборам команд с завершенной разработкой, а статус Open – развивающимся наборам команд.

Операции умножения, деления и вычисления остатка не входят в базовый набор инструкций, а выделены в отдельное расширение (M — Multiply extension). Также стандартизован отдельный набор атомарных операций (A -Atomic extension) - операция, которая либо выполняется целиком, либо не выполняется вовсе. Операция в общей области памяти называется **атомарной**, если она завершается в один шаг относительно других потоков, имеющих доступ к этой памяти.

Для операций над числами в бинарных форматах c плавающей точкой используется набор дополнительных 32 регистров FPU (Floating Point Unit), которые совместно используются расширениями базового набора инструкций для трёх вариантов точности: одинарной — 32 бита (F extension), двойной — 64 бита (D — Double precision extension), а также четверной — 128 бит (Q — Quadruple precision extension).

**Доступ к памяти**

Как и многие проекты с RISC-концепцией, RISC–V является архитектурой load-store: вычислительные инструкции адресуют только регистры, а инструкции load и store передают данные между регистрами и памятью.

Большинство инструкций загрузки и хранения включают 12-битное смещение и два идентификатора регистра. Один регистр является базовым регистром. Другой регистр является источником (для записи) или местом назначения (для чтения.) Смещение добавляется в базовый регистр для получения адреса. Формирование адреса в виде Базового регистра плюс смещение позволяет отдельным инструкциям получить доступ к структурам данных. Например, если базовый регистр указывает на вершину стека, отдельные инструкции могут обращаться к локальным переменным подпрограммы в стеке. Использование постоянного нулевого регистра в качестве базового адреса позволяет отдельным командам обращаться к памяти вокруг нулевого адреса. Хотя память адресуется как 8-битные байты, слова вплоть до размера регистра, могут быть доступны с инструкциями загрузки Load и хранения Store.

Как многие наборы команд RISC процессор RISC-V не имеет адресных режимов, которые записывают данные обратно в регистры. Например, не предусмотрено команд с автоинкрементом адресных регистров.

**Непосредственная адресация**

RISC-V обрабатывает 32-битные константы и адреса с помощью инструкций, которые устанавливают старшие 20 бит 32-битного регистра. Инструкция непосредственной загрузки lui, загружает 20 бит в биты от 31 до 12. Затем вторая инструкция, такая как addi, может установить младшие 12 бит.

При выполнении операций с константами большого размера возникают проблемы, связанные с их загрузкой в регистры. Так при вычислении выражения a = b + 0x123456 размещение константы в инструкции addi требует 24 бита, которых в формате не предусмотрено. Для разрешения проблемы можно использовать псевдоинструкцию li для загрузки больших констант, которая преобразуется ассемблером в две инструкции: lui x5, 0x123 (загрузить константу в старшие биты 31-12 регистра) и addi x5, x5, 0x456 (сложение того же регистра с младшими 12 битами).

Этот метод может быть расширен для того, чтобы позволить создавать позиционно-независимый код путем добавления, например, инструкции auipc, которая генерирует 20 старших битов адреса путем добавления смещения к счетчику программы и сохранения результата в базовом регистре. Это позволяет программе генерировать 32-разрядные адреса, используя смещение относительно счетчика программы.

**Вызовы подпрограмм, переходы и ветвления**

Вызов подпрограммы RISC-V jal (jump and link) помещает свой адрес возврата в регистр. Это быстрее, чем во многих компьютерных архитектурах, потому что экономит доступ к памяти по сравнению с системами, которые сохраняют адрес возврата непосредственно в стеке в памяти.

Процессоры RISC-V могут переходить к вычисляемым адресам с помощью jump и link-регистров. Инструкция jal добавляет большее 20-битное смещение к счетчику программ PC. В отличие от этого инструкции jalr получают свой адрес назначения, добавляя 12-битное смещение к базовому регистру. То есть команда JALR  передает управление по адресу, равному сумме содержимого базового регистра и 12-разрядного смещения.

JALR rd, offset(rs1) # rd ← PC + 4, PC ← rs1 + offset

При этом переход в JALR не происходит относительно PC. Вместо этого он происходит относительно rs1 .

Если надо, чтобы адрес был относителен к программному счётчику (РС) следует использовать специальную инструкцию  AUIPC , что означает Add Upper Immediate to PC (добавить константу к старшим битам РС).

AUIPC rd, immediate # rd ← PC + immediate[31:12] << 12

По ней мы сохраняем старшие 20 бит смещения до некоторой метки в верхние 20 бит регистра  rd. Смещение JALR может быть шириной 12 бит, и вместе мы получаем (20 + 12 = 32), что даёт нам 32-битный адрес.  То есть комбинация AUIPC и JALR  таким образом:

AUIPC t0, proc1 # Store upper 20-bits address of proc1

JALR ra, proc1(t0)

RISC-V перерабатывает команды jal и jalr для получения безусловных 20-разрядных переходов относительно счетчика программ PC и безусловных 12-разрядных переходов на основе регистров. Переходы просто устанавливают регистр связи 0, чтобы обратный адрес не сохранялся.

RISC-V также использует команду jalr для возврата из подпрограммы: для этого базовый регистр команды jalr устанавливается как регистр связи, сохраненный jal или jalr. Если смещение jalr и регистр связи равны нулю, то смещение отсутствует, и обратный адрес не сохраняется.

ISA RISC-V требует предсказания ветвлений по умолчанию для проектируемых процессоров: условные ветви с переходом назад должны быть предсказаны. Предсказания легко декодировать в конвейерном процессоре: адреса перехода - это знаковые числа в дополнительном коде, которые прибавляются к счетчику программ PC. Ветви с переходом назад имеют отрицательные смещения адреса в дополнительном коде, и, таким образом, имеют 1 в старшем значащем бите адреса.

Руководство ISA рекомендует оптимизировать программное обеспечение, чтобы избежать остановок конвейера, используя прогноз ветвлений по умолчанию. Это позволяет повторно использовать старший значащий бит знакового относительного адреса в качестве бита подсказки, чтобы предсказать, будет ли выполнен условный переход или нет. Таким образом, никакие другие биты подсказки не требуются в кодах операций ветвления RISC-V. Эта особенность дает возможность использовать больше битов в кодах операций ветвления. Простые, недорогие процессоры могут просто следовать прогнозам по умолчанию и по-прежнему хорошо работать с оптимизирующими компиляторами.

**Арифметические и логические наборы команд**

RISC-V разделяет математику на минимальный набор целочисленных инструкций (set I) со сложением, вычитанием, сдвигом, битовой логикой и ветвлением со сравнением. Конкретные реализации могут имитировать большинство других наборов инструкций RISC-V с помощью программного обеспечения.

Целочисленные инструкции умножения (набор M) включают в себя знаковое и беззнаковое умножение и деление. Целочисленные операции умножения и деления двойной точности включены, как умножения и деления, которые вычисляют старшее слово результата.

Инструкции с плавающей точкой (набор F) включают арифметику с одинарной точностью, а также условные переходы, аналогичные целочисленной арифметике. Для этого требуется дополнительный набор из 32 регистров с плавающей точкой. Они отделены от целочисленных регистров. Инструкции с плавающей точкой двойной точности (набор D) обычно предполагают, что регистры с плавающей точкой являются 64-разрядными (т. е. двойной длины), и подмножество F совместимо с набором D. Также определяется 128-битный ISA (Q) с плавающей точкой с квадратичной точностью. Процессоры RISC-V без плавающей точки могут использовать библиотеку программной эмуляции работы с плавающей точкой.