Архитектура ARM

**КРАТКОЕ ОПИСАНИЕ АРХИТЕКТУРЫ ARM**

Архитектура ARM (Advanced RISC Machine) является широко распространенной (в особенности на рынке мобильных и встраиваемых устройств) архитектурой класса RISC (Reduced Instruction Set Computer), который является альтернативой класса архитектур CISC (Complex Instruction Set Computer – к нему, в частности, принадлежит семейство архитектур x86).

RISC-процессоры возникли позже CISC-процессоров, и в своей организации более соответствуют современному положению вещей. CISCпроцессоры создавались в расчёте на быструю (относительно процессора) и очень ограниченную по объему память и написание программ на ассемблере. Соответственно, часто используемые команды кодируются по возможности короткими кодами, имеются сложные команды, позволяющие кратко описывать сложные действия, а в качестве операндов наравне используются и регистры и память. Позднее стало ясно, что память дешевеет, увеличивается в объеме, но замедляется относительно скорости процессора. При этом развиваются технологии компиляции, и программирование переходит на языки более высокого уровня, чем ассемблер. В соответствии с этими тенденциями были разработаны RISC-процессоры.

RISC-архитектуры имеют следующие основные особенности. Во-первых, упрощен набор инструкций. Команды просты в декодировании – они имеют одинаковый простой формат и кодируются машинным словом фиксированной длины. RISC-команды просты в исполнении, т.е. они исполняются быстро, и, преимущественно, одинаково по времени – отсутствуют команды, которые бы выполнялись много тактов. Например, часто отсутствуют команды целочисленного деления или вычисления квадратного корня. Они выражаются через более простые команды компилятором. За счёт такого упрощения команд улучшается их конвейеризация и суперскалярные свойства процессора.

Во-вторых, большой (по сравнению с CISC) регистровый файл. Регистры, как правило, нумеруются, а не именуются. Все арифметические команды выполняются именно над регистрами, а не над оперативной памятью. Для работы с памятью же существует всего две команды – чтение (load) и запись (store), позволяющие считывать значения ячеек памяти в регистр и записать значения из регистров в память. Память, таким образом, не используется для хранения временных значений (что является обычным делом в CISC) системах.  Использование регистров позволяет минимизировать количество обращений к основной памяти, что повышает производительность.

**Быстрый доступ**: В RISC-архитектуре используется большое количество регистров (обычно 32 или более), которые обеспечивают быстрый доступ к данным. Регистры используются для временного хранения данных, участвующих в вычислениях.

В-третьих, RISC-архитектуры не рассчитаны на удобство программирования вручную, на языке ассемблера. Ставка сделана на компиляторы, которые сгенерируют машинный код и проведут его оптимизацию автоматически. RISC-код является менее ёмким, чем CISC-код, он занимает больше места и обычно менее удобен для восприятия человеком.

В целом, RISC-архитектура является более удобной для реализации в процессоре, чем CISC-архитектура. Поэтому современные CISC-процессоры, как правило, реализованы на базе RISC-ядра – сложные CISC-команды расщепляются на микрокоманды, и далее процессор функционирует уже как RISC. CISC-оболочка, по большому счёту, нужна лишь для поддержки бинарной совместимости с существующим кодом.

**Стандартные соглашения о связях**

Хотя при создании своей ОС программист волен определять собственные соглашения о связях и правила использования регистров, есть смысл придерживаться правил, разработанных компанией ARM и поддерживаемых всеми популярными трансляторами для этой архитектуры. Вкратце они заключаются в следующем.

 В качестве указателя стека всегда используется регистр SP (R13). Стек растёт вниз, SP указывает на последнее используемое слово. Эти правила полностью соответствуют аппаратным требованиям набора команд Thumb.

  Для передачи параметров в вызываемую подпрограмму могут использоваться регистры R0–R3. Если необходимо передать параметр, по размерам превышающий слово, при использовании обычного порядка слов «младший–старший» младшее слово помещается в регистр с меньшим номером; при использовании порядка «старший–младший» — в регистр со старшим номером. Если параметров слишком много, дополнительные параметры передаются через стек. Кроме того, при наличии арифметического сопроцессора для передачи параметров могут использоваться его регистры, но их использование здесь не обсуждается.

 Возврат результатов осуществляется через регистры R0–R3.

Вызываемая подпрограмма обязана сохранять значения регистров R4–R11, а также корректно восстанавливать значение указателя стека.

Регистры R0–R3, R12 и LR, а также флаги регистра состояния вызываемой подпрограммой могут свободно изменяться.

Компоновщик при связывании вызовов подпрограмм может использовать регистр R12 в качестве рабочего, поэтому подпрограммы не могут рассчитывать на сохранность его содержимого, если используют вызовы подпрограмм из других модулей.

 В ряде систем R7 хранит значение, зависящее от системы (например, адрес локальной памяти потока — TLS). В таких системах изменять этот регистр обычно запрещается.

**Описание**: Регистр lr (Link Register) используется для хранения адреса возврата при вызове подпрограмм.

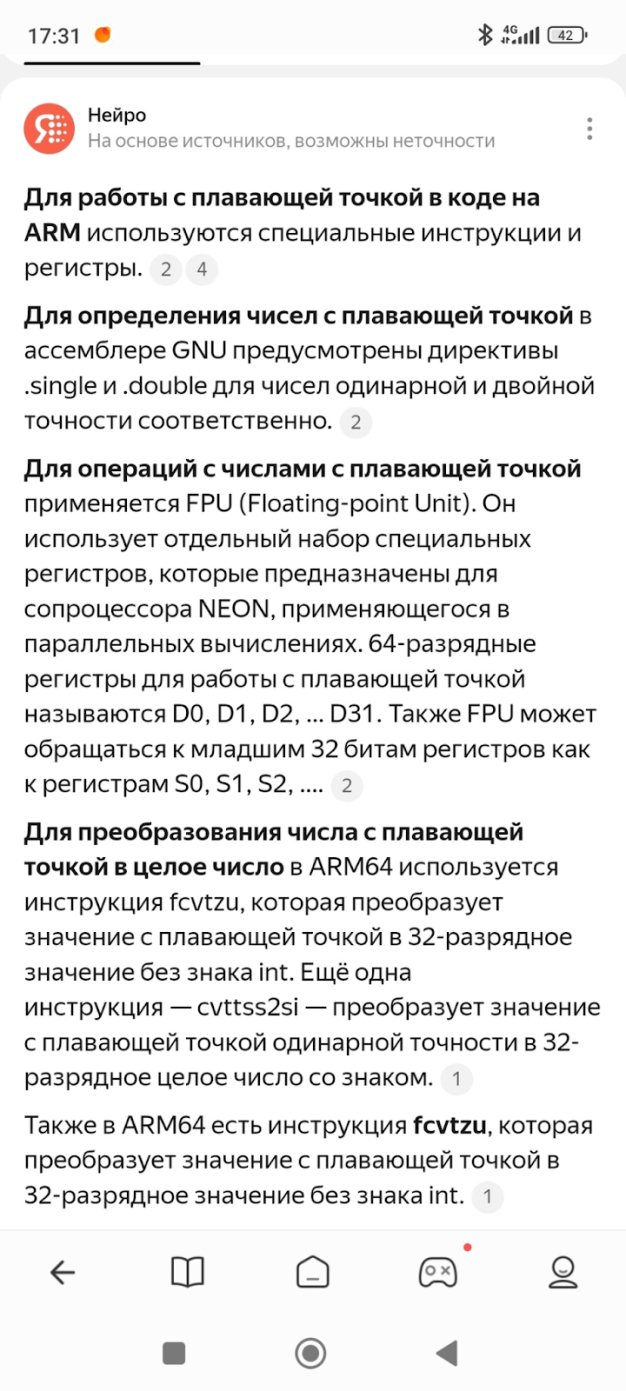
**Пример**: push {r7, lr} — сохраняет регистры r7 и lr в стек.

**Арифметический сопроцессор** предназначен для выполнения операций над числами в формате с плавающей точкой (вещественные числа) и длинными целыми числами. Он значительно (в десятки раз) ускоряет вычисления, связанные с вещественными числами.

Вещественные числа хранятся и обрабатываются в регистрах, общее число которых зависит от модели FPU:

FPU типов FPv4-SP и FPv5, входящие в состав ряда процессоров архитектуры ARMv7-M, позволяют использовать шестнадцать 64-разрядных регистров D0-D15 и 32 32-разрядных регистра S0-S31; остальные модели FPU включают 32 64-разрядных регистра D0-D31.

С точки зрения программиста FPU располагает 32- и 64-разрядными регистрами с плавающей точкой, причём они могут использоваться одновременно. Технически, однако, эти регистры перекрываются: один 64-разрядный регистр может служить двумя 32-разрядными со смежными номерами и наоборот (регистру D0 соответствуют S0 и S1, регистру D1 – S2 и S3 и т.д.). По этой причине запись в регистр одной разрядности портит содержимое соответствующих регистров другой разрядности.



1. **Регистры одинарной точности (32 бита):**
   * **s0-s31:** 32 регистра, каждый из которых имеет размер 32 бита.
2. **Регистры двойной точности (64 бита):**
   * **d0-d31:** 32 регистра, каждый из которых имеет размер 64 бита.
3. Регистры двойной точности d0-d31 могут быть связаны с регистрами одинарной точности s0-s31 следующим образом:
   * d0 соответствует s0 и s1
   * d1 соответствует s2 и s3
   * и так далее...
4. **Регистры четверной точности (128 бит):**
   * **q0-q31:** 32 регистра, каждый из которых имеет размер 128 бит.
5. Регистры четверной точности q0-q31 могут быть связаны с регистрами одинарной и двойной точности следующим образом:
   * q0 соответствует d0 и d1 (и, следовательно, s0-s3)
   * q1 соответствует d2 и d3 (и, следовательно, s4-s7)
   * и так далее...

**Команды**

Можно выделить следующие основные группы команд:

· Команды преобразования данных: сложение, вычитание, умножение, умножение со сложением, побитовые логические операции, сравнение.

· Команды передачи данных: копирование данных между регистрами, чтение и запись данных в памяти. Для операций с памятью поддерживается автоматическое увеличение регистра-указателя до или после обращения в память. Кроме того, поддерживаются многорегистровые операции с памятью, использующие сразу несколько регистров общего назначения.

· Команды передачи управления: команды безусловного и условного перехода. В архитектуре поддерживается условное исполнение любой команды в зависимости от значений флагов. Кроме того, существует быстрый способ вызова подпрограммы с использованием специальной команды перехода, сохраняющей адрес следующей за ней программы в регистре r14.

Арифметические команды:

o   ADD: Сложение.

o   SUB: Вычитание.

o   MUL: Умножение.

o   DIV: Деление.

·        Логические команды:

o   AND: Логическое И.

o   ORR: Логическое ИЛИ.

o   EOR: Логическое исключающее ИЛИ.

·        Команды сдвига:

o   LSL: Логический сдвиг влево.

o   LSR: Логический сдвиг вправо.

o   ASR: Арифметический сдвиг вправо.

Стек в ARM управляется с помощью команд PUSH и POP. Стек растет в сторону уменьшения адресов памяти.

**Работа с памятью**

В RISC-архитектуре обращение к памяти обычно происходит через специальные инструкции, которые явно указывают на операции чтения или записи данных. Основные принципы обращения к памяти в RISC-архитектуре включают:

1. Инструкции загрузки и сохранения (Load/Store)

* Load (LDR): Инструкция загрузки данных из памяти в регистр. Например, LDR R1, [R2] загружает данные из памяти по адресу, хранящемуся в регистре R2, в регистр R1.
* Store (STR): Инструкция сохранения данных из регистра в память. Например, STR R1, [R2] сохраняет данные из регистра R1 в память по адресу, хранящемуся в регистре R2.

2. Адресация памяти

* Прямая адресация: Адрес памяти указывается непосредственно в инструкции. Например, LDR R1, [0x1000] загружает данные из памяти по адресу 0x1000 в регистр R1.
* Регистровая адресация: Адрес памяти хранится в регистре. Например, LDR R1, [R2] загружает данные из памяти по адресу, хранящемуся в регистре R2, в регистр R1.
* Индексная адресация: Адрес памяти вычисляется как сумма значения регистра и смещения. Например, LDR R1, [R2, #4] загружает данные из памяти по адресу, равному значению регистра R2 плюс 4, в регистр R1.
* Базовая адресация с индексом: Адрес памяти вычисляется как сумма значения регистра и значения другого регистра. Например, LDR R1, [R2, R3] загружает данные из памяти по адресу, равному сумме значений регистров R2 и R3, в регистр R1.

3. Типы данных

* Байты, полуслова, слова: Инструкции загрузки и сохранения могут работать с данными различных размеров: байтами, полусловами (2 байта) и словами (4 байта). Например, LDRB R1, [R2] загружает байт из памяти в регистр R1, а LDRH R1, [R2] загружает полуслово.

4. Примеры инструкций

1. Загрузка слова: LDR R1, [R2] — загружает слово из памяти по адресу, хранящемуся в R2, в регистр R1.
2. Сохранение слова: STR R1, [R2] — сохраняет слово из регистра R1 в память по адресу, хранящемуся в R2.
3. Загрузка байта: LDRB R1, [R2] — загружает байт из памяти по адресу, хранящемуся в R2, в регистр R1.
4. Сохранение байта: STRB R1, [R2] — сохраняет байт из регистра R1 в память по адресу, хранящемуся в R2.
5. Особенности RISC-архитектуры

* Простота и единообразие: Инструкции загрузки и сохранения в RISC-архитектуре просты и единообразны, что упрощает их декодирование и выполнение.
* Минимизация обращений к памяти: Использование регистров для временного хранения данных позволяет минимизировать количество обращений к основной памяти, что повышает производительность.

Сохранение значений:

* Команда stp x29, x30, [sp, -16]! сохраняет значения регистров x29 и x30 в память по адресу, который вычисляется как sp - 16.

Обновление указателя стека:

* Восклицательный знак (!) указывает, что после сохранения значений указатель стека (sp) должен быть обновлен на новый адрес, то есть sp = sp - 16.

В архитектуре CISC (Complex Instruction Set Computing) обращение к памяти осуществляется через более сложные и разнообразные инструкции, которые могут выполнять несколько операций за один такт процессора. В отличие от RISC, где обращение к памяти происходит через явные инструкции загрузки и сохранения, в CISC-архитектуре инструкции могут содержать неявные обращения к памяти, что делает их более гибкими, но и более сложными для декодирования и выполнения.

Основные способы обращения к памяти в CISC-архитектуре:

1. Инструкции загрузки и сохранения

* MOV: Инструкция загрузки данных из памяти в регистр. Например, MOV AX, [BX] загружает данные из памяти по адресу, хранящемуся в регистре BX, в регистр AX.
* MOV: Инструкция сохранения данных из регистра в память. Например, MOV [BX], AX сохраняет данные из регистра AX в память по адресу, хранящемуся в регистре BX.

2. Адресация памяти

* Прямая адресация: Адрес памяти указывается непосредственно в инструкции. Например, MOV AX, [0x1000] загружает данные из памяти по адресу 0x1000 в регистр AX.
* Регистровая адресация: Адрес памяти хранится в регистре. Например, MOV AX, [BX] загружает данные из памяти по адресу, хранящемуся в регистре BX, в регистр AX.
* Индексная адресация: Адрес памяти вычисляется как сумма значения регистра и смещения. Например, MOV AX, [BX+4] загружает данные из памяти по адресу, равному значению регистра BX плюс 4, в регистр AX.
* Базовая адресация с индексом: Адрес памяти вычисляется как сумма значения регистра и значения другого регистра. Например, MOV AX, [BX+SI] загружает данные из памяти по адресу, равному сумме значений регистров BX и SI, в регистр AX.

3. Инструкции с неявным обращением к памяти

* Инструкции с непосредственным операндом: Инструкции могут содержать непосредственные значения, которые могут быть загружены в регистры или использованы в вычислениях. Например, MOV AX, 100 загружает значение 100 в регистр AX.
* Инструкции с автоинкрементом/автодекрементом: Инструкции могут автоматически изменять значение регистра, используемого для адресации памяти. Например, MOV AX, [BX++] загружает данные из памяти по адресу, хранящемуся в BX, и затем увеличивает значение BX на 1.
* Инструкции с префиксами: Префиксы могут изменять поведение инструкций, например, изменяя размер операнда или адресацию. Например, REP MOVSB выполняет копирование строки байтов из одного участка памяти в другой.

4. Примеры инструкций

* Загрузка слова: MOV AX, [BX] — загружает слово из памяти по адресу, хранящемуся в BX, в регистр AX.
* Сохранение слова: MOV [BX], AX — сохраняет слово из регистра AX в память по адресу, хранящемуся в BX.
* Загрузка байта: MOV AL, [BX] — загружает байт из памяти по адресу, хранящемуся в BX, в младший байт регистра AX.
* Сохранение байта: MOV [BX], AL — сохраняет байт из младшего байта регистра AX в память по адресу, хранящемуся в BX.

5. Особенности CISC-архитектуры

* Сложность и разнообразие: Инструкции в CISC-архитектуре могут выполнять несколько операций за один такт, что делает их более сложными и разнообразными.
* Неявные обращения к памяти: Инструкции могут содержать неявные обращения к памяти, что делает их более гибкими, но и более сложными для декодирования и выполнения.
* Микропрограммирование: Многие CISC-процессоры используют микропрограммирование для реализации сложных инструкций, что позволяет упростить их выполнение.

**ARM-32 bit**

**Регистры**

 Программисту доступны 16 32-битных регистров общего назначения (r0 – r15) и 32-битный регистр состояния (CPSR).

Регистры общего назначения используются для хранения произвольных целочисленных данных. При использовании в большинстве команд в качестве операндов эти регистры полностью взаимозаменяемы, но некоторые команды используют только конкретные регистры. Регистр r15 используется как счетчик команд (program counter, instruction pointer(PC)). Регистр r14 используется для сохранения адреса возврата в результате исполнения специальной команды условного перехода(LR). Регистр r13 обычно используется как указатель вершины стека (stack pointer(SP)), хотя и не поддержан специально программной архитектурой.

В архитектуре ARM регистр r7 часто используется как базовый регистр для хранения указателя фрейма (Frame Pointer).

Этот регистр может содержать адрес начала текущего фрейма стека, что упрощает доступ к локальным переменным и сохраненным регистрам.

Регистры R0-R3 используются для передачи параметров и возврата результатов.

Регистр состояния (CPSR – current program status register) хранит однобитовые флаги:

· знаковый флаг (N – Negative),

· флаг нуля (Z – Zero),

· флаг переноса (C – Carry),

· флаг переполнения (V – oVerflow), а также другие поля, отражающие состояние процессора

Первые 13 регистров общего назначения с точки зрения архитектуры являются абсолютно равноправными, за исключением двух моментов:

·       в командах [набора ARM](https://osdev.fandom.com/ru/wiki/%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0_%D0%BA%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4_ARM) LDRD и STRD, оперирующих сразу парой регистров, указывается всегда регистр с чётным номером, а пару с ним составляет следующий за ним регистр с нечётным номером (заметим, что в аналогичных командах набора Thumb-2 это правило не действует: оба используемых регистра указываются в них явным образом);

·        в почти всех командах набора Thumb и в некоторых командах Thumb-2 могут использоваться регистры R0–R7, но не могут регистры с большими номерами.

·        Поскольку счётчик команд содержит адрес выполняемой команды, запись в него вызывает переход на команду по записываемому адресу.

**Роль регистра pc**

Регистр pc (Program Counter) в архитектуре ARM является одним из ключевых регистров, который хранит адрес следующей команды, которая будет выполнена.

1. Хранение адреса следующей команды:
   * Регистр pc всегда указывает на адрес команды, которая будет выполнена следующей.
   * При выполнении каждой команды значение pc автоматически увеличивается на длину выполненной команды (обычно 4 байта в ARM).
2. Управление потоком выполнения:
   * Регистр pc используется для управления потоком выполнения программы. Например, при выполнении команд перехода (B, BL, BX), значение pc обновляется на адрес целевой команды.
   * При вызове подпрограммы команда BL сохраняет текущее значение pc в регистр lr (Link Register), а затем обновляет pc на адрес подпрограммы.
3. Возврат из подпрограммы:
   * При возврате из подпрограммы команда RET (или BX lr) использует значение регистра lr (который хранит адрес возврата) для обновления pc.

Примеры использования pc

Пример 1: Простой переход

assembly

Copy

B .L1      ; Переход к метке .L1

; Код, который не будет выполнен

.L1:

; Код, который будет выполнен после перехода

* Описание: Команда B .L1 обновляет значение pc на адрес метки .L1, что приводит к переходу к этой метке.

Пример 2: Вызов подпрограммы

assembly

Copy

BL my\_function ; Вызов подпрограммы my\_function

; Код, который будет выполнен после возврата из подпрограммы

my\_function:

; Код подпрограммы

RET        ; Возврат из подпрограммы

* Описание: Команда BL my\_function сохраняет текущее значение pc в регистр lr, а затем обновляет pc на адрес подпрограммы my\_function. Команда RET использует значение lr для возврата к месту вызова подпрограммы.

Пример 3: Использование pc для вычисления адреса

assembly

Copy

    LDR r0, [pc, #4] ; Загрузка значения из памяти по адресу, который находится на 4 байта после текущей команды

; Код, который будет выполнен после загрузки

* Описание: Команда LDR r0, [pc, #4] использует текущее значение pc плюс смещение 4 для вычисления адреса, по которому будет загружено значение в регистр r0.

Заключение

Регистр pc (Program Counter) играет ключевую роль в управлении потоком выполнения программы в архитектуре ARM. Он хранит адрес следующей команды, которая будет выполнена, и используется для переходов, вызовов подпрограмм и возвратов из них.

Регистр pc (Program Counter) содержит адрес следующей команды, которая будет выполнена.

При извлечении значения из стека в регистр pc, происходит переход к адресу, хранящемуся в стеке.

1. **Сохранение регистров в стек:**
   * Команда PUSH {r7, lr} сохраняет значения регистров r7 и lr (Link Register) в стек.
   * Регистр lr содержит адрес возврата, который будет использован для возврата из подпрограммы.
2. **Установка базового регистра:**
   * Команда MOV r7, sp устанавливает регистр r7 на вершину стека.
   * Это позволяет использовать r7 как базовый регистр для доступа к локальным переменным и сохраненным регистрам.
3. **Выполнение кода подпрограммы:**
   * Выполняется код подпрограммы.
4. **Восстановление регистров и возврат из подпрограммы:**
   * Команда POP {r7, pc} извлекает значения из стека и загружает их в регистры r7 и pc.
   * Значение, извлеченное из стека в регистр pc, становится адресом следующей команды, что приводит к возврату из подпрограммы.

**Команды**

|  |  |  |
| --- | --- | --- |
| **Команда** | **Назначение** | **Пример** |
| **push** | Сохранение регистров в стек | **push    {r7}** *// Сохраняем r7 в стек* |
| **sub** | Вычитание | **sub     sp, sp, #28** *// Выделяем 28 байт на стеке для локальных переменных* |
| **add** | Сложение | **add     r7, sp, #0** *// Устанавливаем r7 на вершину стека* |
| **str** | Сохранить регистр в памяти | **str     r0, [r7, #4]** *// Сохраняем аргумент функции (N) в стек* |
| **mov** | Загрузить в регистр константу | **mov     r2, #0** *// Инициализируем r2 нулем* |
| **movt** | Запись старшей половины 32-битного значения в регистр. | **movt    r3, 16368** *// Устанавливаем старшие 16 бит r3 в 16368 (что соответствует 1.0 в формате с плавающей точкой)* |
| **movs** | Перемещение значения в регистр и обновления флагов. | **movs    r3, #1 *//*** записывает значение 1 в регистр r3 и обновляет флаги. |
| **vmov** | Перемещение данных между регистрами общего назначения и регистрами с плавающей точкой. | **vmov    s15, r3 @ int** *//Перемещаем значение r3 в регистр с плавающей точкой s15* |
| **vmov.f64** | Перемещение 64-битного значения (число с плавающей точкой двойной точности) между регистрами с плавающей точкой. | **vmov.f64        d0, d16** *//Перемещаем результат в d0 (возвращаемое значение);перемещает значение из регистра d16 в регистр d0.* |
| **movw** | Запись младшей половины 32-битного значения в регистр. | **movw    r0, #:lower16:.LC0** *// Загружаем младшие 16 бит адреса строки .LC0 в r0* |
| **strd** | Сохранение пары регистров (64 бита) в память | **strd    r2, [r7, #16]** *// Сохраняем пару регистров r2 и r3 (0.0 и 1.0) в стек по адресу (r7+16)* |
| **b** | Безусловный переход к указанной метке в коде. | **b       .L6 */****/Переходим к метке .L6* |
| **ldr** | Загрузить в регистр из памяти по указанному адресу | **ldr     r2, [r7, #20]** *// Загружаем значение по адресу [r7+20]  в r2* |
| **vcvt.f64.s32** | Преобразования целого числа в число с плавающей точкой двойной точности. | **vcvt.f64.s32    d16, s15** *//Преобразуем целое число в регистре s15  в число с плавающей точкой двойной точности и сохраняет его в регистре d16* |
| **vldr.64** | Загрузка 64-битного значения (число с плавающей точкой двойной точности) из памяти в регистр с плавающей точкой. | **vldr.64 d17, [r7, #16]** *//Загружаем текущий результат факториала из памяти по адресу [r7+16] в d17* |
| **vmul.f64** | Умножение двух чисел с плавающей точкой двойной точности. | **vmul.f64        d16, d17, d16** *//Умножаем текущий результат на i; умножает значения в регистрах d17 и d16 и сохраняет результат в регистр d16.* |
| **vstr.64** | Сохранение 64-битного значения (число с плавающей точкой двойной точности) из регистра с плавающей точкой в память. | **vstr.64 d16, [r7, #16]** *//Сохраняем новый результат(newValue) в стек; сохраняет значение из регистра d16 в память по адресу [r7 + 16].* |
| **adds** | Сложение двух значений и обновления флагов. | **adds    r3, r3, #1** *//Увеличиваем счетчик i на 1;увеличивает значение в регистре r3 на 1 и обновляет флаги.* |
| **cmp** | Команда CMP используется для сравнения двух значений и установки флагов состояния процессора в зависимости от результата сравнения. | **cmp     r2, r3** *//Сравниваем i и N* |
| **ble** | Условный перехода, если результат предыдущей операции меньше или равен нулю (флаг N установлен или флаг Z установлен). | **ble     .L3 *//****переходит к метке .L3, если результат предыдущей операции меньше или равен нулю.* |
| **ldrd** | Загрузка пары регистров (64 бита) из памяти. | **ldrd    r2, [r7, #16] *//****загружает значения из памяти по адресу [r7 + 16] в регистры r2 и r3.* |
| **bx** | Команда BX используется для перехода к адресу, хранящемуся в регистре, и переключения между режимами Thumb и ARM (в зависимости от младшего бита адреса).  Это позволяет выполнять код, написанный в разных режимах, без необходимости явного переключения режима. | **bx      lr** *//Возврат из функции; Возврат из подпрограммы, используя адрес возврата в регистре LR* |
| **vpush.64** | Cохранение регистров с плавающей точкой (64 бита) в стек. | **vpush.64        {d8}** *//Сохраняем регистр d8 в стек* |
| **bl** | Команда BL используется для вызова подпрограммы.  Она сохраняет адрес следующей команды в регистре LR (Link Register) и переходит к указанной метке.  После выполнения подпрограммы можно вернуться к основному коду, используя команду BX LR. | **bl      factorial */****/ Вызываем функцию factorial(i)* |
| **vdiv.f64** | Деление двух чисел с плавающей точкой двойной точности. | **vdiv.f64        d16, d8, d17 *//*** *Делим результат pow на результат factorial; делит значение в регистре d8 на значение в регистре d17 и сохраняет результат в регистр d16.* |
| **vadd.f64** | Сложение двух чисел с плавающей точкой двойной точности. | **vadd.f64        d16, d17, d16** *// складывает значения в регистрах d17 и d16 и сохраняет результат в регистр d16.* |
| **vldm** | Загрузка нескольких регистров с плавающей точкой из стека. | **vldm    sp!, {d8}** *// Восстанавливаем регистр d8 из стека* |
| **pop** | Команда POP используется для извлечения значений из стека и загрузки их в регистры.  Стек растет в сторону уменьшения адресов памяти, поэтому команда POP извлекает значения с вершины стека. | **pop     {r7, pc}** *//освобождаем стек;Восстанавливаем r7 и pc из стека,* |
| **.ascii** | Директива .ascii используется для определения строки символов в памяти. | **.ascii  "Res: %lf\000"** *// определяет строку "Res: %lf\000" в памяти* |
| **bne** | Переход условный, если результат предыдущей операции не равен нулю (флаг Z сброшен). | **bne     .L3** *//Если i != N, переходим к метке .L3* |

Команда movs:

Команда movs перемещает значение в регистр и обновляет флаги состояния процессора на основе этого значения.

movs    r3, #1

**Как это работает:**

1. **Перемещение значения:**
   * Значение 1 перемещается в регистр r3.
2. **Обновление флагов:**
   * **N (Negative):** Сбрасывается (устанавливается в 0), так как значение 1 не отрицательное.
   * **Z (Zero):** Сбрасывается (устанавливается в 0), так как значение 1 не равно нулю.
   * **C (Carry):** Сбрасывается (устанавливается в 0), так как операция не связана с переносом.
   * **V (Overflow):** Сбрасывается (устанавливается в 0), так как операция не связана с переполнением.

**Пример:**

Рассмотрим пример, где используется команда movs и последующая проверка флагов:

movs    r3, #1и обновляем флаги

bne     .L2          // Переход на метку .L2, если Z = 0 (результат не равен нулю)

В этом примере:

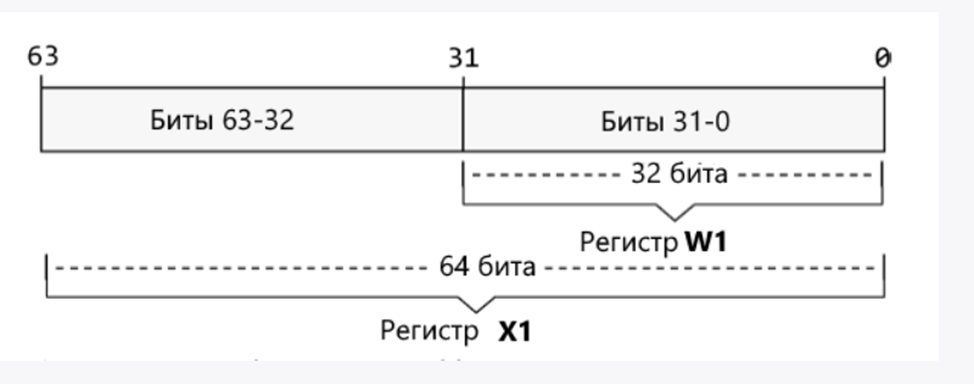
* После выполнения movs r3, #1, флаг Z будет сброшен (Z = 0), так как значение 1 не равно нулю.
* Команда bne .L2 выполнит переход на метку .L2, так как условие Z = 0 выполняется.

**ARM-64 bit**

**Регистры**

Регистры общего назначения

Архитектура Arm64 предоставляет 31 регистр общего назначения, которые применяются для любых задач, имеют размер 64 бита и называются X0, X1. . .X30. Также доступны 32-битные регистры W0–W30, которые представляют младшие 32 бита соответствующих регистров X0-X30. Например, регистры X1 и W1 схематически:

****

Хотя регистры общего назначения могут использоваться для любых задач, для определенных ситуаций есть определенные рекомендации. В частности, для применения регистров при вызове функций существуют рекомендации Стандарта вызова процедур архитектуры Arm (Arm Architecture Procedure Call Standard или AAPCS64), который предписывает следующее:

·        Регистры X0–X7 используются для передачи параметров в функцию и возвращения результата.

·        Регистр X8 используется для передачи в функцию адреса блока памяти при непрямой адресации.

·        Регистры X9–X15 следует сохранять при вызове функций. Ответственность за сохранение ложится на вызывающую сторону, которая вызывает функцию. Затронутые регистры сохраняются в кадре стека вызывающей функции, что позволяет подпрограмме изменять эти регистры..

·        Регистры X16–X18 используются для хранения промежуточных результатов между вызовами функций.

·        Регистры X19–X28 - это регистры, сохраняемые вызываемой функцией в стека, что позволяет функции изменять эти регистры, но также требует их восстановления перед возвратом к вызывающей стороне.

·        Регистр X29 используются как указатель на фрейм стека, еще называется FP (frame pointer).

·        Регистр X30 используются для хранения адреса возврата из функции, еще называется LR (link register).

В дополнение ARM64 еще ряд регистров, которые выполняют определенные функции.

Регистр PC ( Program Counter - счетчик программ) - 64-битный регистр, который содержит адрес текущей инструкции. Перед выполнением каждая инструкция загружается из области памяти, на которую ссылается регистр PC. После выполнения инструкции PC автоматически переходит к следующей инструкции. В Armv8 нельзя явным образом обращаться к PC в инструкциях. Значение регистра PC может быть явно обновлено только с помощью генерации исключений и переходов.

Регистр SP (Stack Pointer) или указатель стека хранит адрес верхушки стека. Область стека используется программами для хранения и доступа к данным локальных переменных для данной функции, а также в качестве временной памяти для хранения таких данных, как адреса возврата функции. Используя SP, можно загружать данные в стек или, наоборот, извлекать данные.

* **!** — пост-индексный режим, который означает, что после выполнения команды значение регистра sp будет обновлено.

Регистр SP является 64-битным. К его младшим 32 битам можно обращаться как к отдельному регистру, который называется WSP.

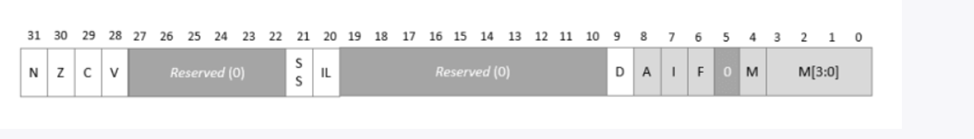
Регистр LR (Link Register) перекрывает регистр X30 (то есть по сути LR и X30 представляют один и тот же регистра). Этот регистр можно свободно использовать для обычных вычислений; однако его основная цель — хранить адреса возврата при вызове функции.

**Фрейм стека**

Регистр общего назначения X29 также используется как указатель фрейма в стеке (Frame Pointer). Например, если стеке храняться переменные для определенной функции, то указатель фрейма стека применяется хранит базовый адрес фрейм стека, выделенного для функции, и применяется для доступа к ее локальным переменным в стеке.

**Регистр состояния**

Регистр состояния PSTATE хранит флаги состояния, которые могут устанавливаться в зависимости от выполняемых инструкций и их результатов.



Пользовательские программы обычно используют только старшие 4 бита - флаги N, Z, C и V. Вкратце посмотрим, какие флаги составляют этот регистр:

·        N: флаг знака, устанавливается, если результат операции представляет отрицательное число

·        Z: флаг нуля, устанавливается, если результат операции равен нулю

|  |  |  |
| --- | --- | --- |
| Команда | Назначение | Пример |
| **sub** | Вычитание | sub     sp, sp, #32 *// Выделяем 32 байта на стеке для локальных переменных* |
| **str** | Сохранить регистр в памяти | str     w0, [sp, 12] *// Сохраняем аргумент функции (N) в стек* |
| fmov | Загрузить в регистр константу с плавающей точкой  Перемещение значения между регистрами с плавающей точкой и регистрами общего назначения | fmov    d0, 1.0e+0 *// Устанавливаем d0 в 1.0 (начальное значение факториала)(newValue)* |
| **mov** | Загрузить в регистр константу | mov     w0, 1 *// Устанавливаем w0 в 1 (счетчик i)* |
| **b** | Безусловный переход к указанной метке в коде. | b       .L2 *// Переходим к метке .L2* |
| **ldr** | Загрузить в регистр из памяти по указанному адресу | ldr     w0, [sp, 20] *// Загружаем счетчик i из стека в w0* |
| scvtf | Преобразование целого числа в число с плавающей точкой двойной точности. | scvtf   d0, w0  *// Преобразуем целое число в число с плавающей точкой двойной точности* |
| fmul | Умножение двух чисел с плавающей точкой | fmul    d0, d1, d0 *// Умножаем текущий результат на i* |
| **add** | Сложение | add     w0, w0, 1 *// Увеличиваем счетчик i на 1* |
| **cmp** | Команда CMP используется для сравнения двух значений и установки флагов состояния процессора в зависимости от результата сравнения.  Команда CMP вычитает второй операнд из первого операнда и устанавливает флаги состояния процессора в зависимости от результата вычитания.  Результат вычитания не сохраняется, а только устанавливаются флаги.  **1.Z (Zero Flag):**  Устанавливается, если результат вычитания равен нулю (значения равны).  Сбрасывается, если результат вычитания не равен нулю (значения не равны).  **2.N (Negative Flag):**  Устанавливается, если результат вычитания отрицательный (значение первого операнда меньше значения второго операнда).  Сбрасывается, если результат вычитания положительный (значение первого операнда больше или равно значению второго операнда).  **3.C (Carry Flag):**  Устанавливается, если произошел перенос из старшего бита (для беззнаковых чисел).  Сбрасывается, если переноса не было.  **4.V (Overflow Flag):**  Устанавливается, если произошло переполнение (для знаковых чисел).  Сбрасывается, если переполнения не было. | cmp     w1, w0 *// Сравниваем i и N* |
| **ble** | Условный перехода, если результат предыдущей операции меньше или равен нулю (флаг N установлен или флаг Z установлен). | ble     .L3 *//Если i <= N, переходим к метке .L3* |
| ret | Команда ret используется для возврата из подпрограммы. Она передает управление по адресу, хранящемуся в регистре lr (Link Register). | ret |
| stp | Сохранение пары регистров в память | stp     x29, x30, [sp, -64]! *// Сохраняем x29, x30 и выделяем 64 байта на стеке* |
| **bl** | Команда BL используется для вызова подпрограммы.  Она сохраняет адрес следующей команды в регистре LR(x30) (Link Register) и переходит к указанной метке.  После выполнения подпрограммы можно вернуться к основному коду, используя команду BX LR. | bl      factorial *// Вызываем функцию factorial(i)* |
| fdiv | Деление двух чисел с плавающей точкой | fdiv    d0, d8, d0 *// Делим значение в регистре d8 на значение в регистре d0 и сохраняем результат в регистре d0* |
| fadd | Сложение двух чисел с плавающей точкой | fadd    d0, d1, d0  *// Складываем значение в регистре d1 со значением в регистре d0*  *и сохраняем результат в d0* |
| .string | Определение строки символов в памяти. | .string "Res: %lf" *// Строка для вывода результата* |
| adrp | Вычисление адреса страницы (адреса, выровненного по 4 КБ) и загрузки его в регистр. | adrp    x0, .LC0 *// Загружаем старшие 48 бит адреса строки .LC0 в x0* |
| ldp | Загрузка пары регистров из памяти. | ldp     x29, x30, [sp], 48 *// Восстанавливаем x29, x30 и восстанавливаем стек* |
| **bne** | Переход условный, если результат предыдущей операции не равен нулю (флаг Z сброшен). | bne     .L3 *// Если i != N, переходим к метке .L3* |

·        C: флаг переноса, устанавливается, если при выполнении арифметической операции произошел перенос

·        V: флаг переполнения, устанавливается, если при выполнении арифметической операции произошло переполнение со знаком

·        SS: бит 21- бит одношагового состояния (single-step state bit), используется отладчиками для пошаговой отладки программы

·        IL: бит 20 - флаг недопустимого состояния исключения

·        DAIF: биты 9-6 - флаги отключения прерываний

**Команды**

**В моем коде O3:**Сохранение контекста:

* Регистры x19 и x20 являются сохраняемыми регистрами, что означает, что их значения должны быть сохранены и восстановлены при вызове подпрограмм.
* Команда stp x19, x20, [sp, 16] сохраняет значения этих регистров в стек, чтобы их можно было восстановить после выполнения подпрограммы.

1. Использование в цикле:
   * Регистр x19 используется как счетчик i в цикле вычисления результата функции result.
   * Регистр x20 используется для хранения значения N + 2, которое сравнивается с i в

**Сравнение ARM-32 bit и ARM-64 bit**

*Архитектура ARM поддерживает как 32-битные (ARMv7 и более ранние), так и 64-битные (ARMv8 и более поздние) версии. Вот основные отличия между ними:*

**1. Размер регистров и адресного пространства**

·        ARM 32-bit (ARMv7 и более ранние):

o   Регистры общего назначения (R0-R15) имеют размер 32 бита.

o   Адресное пространство ограничено 4 ГБ (2^32 байт).

·        ARM 64-bit (ARMv8 и более поздние):

o   Регистры общего назначения (X0-X30) имеют размер 64 бита.

o   Адресное пространство увеличено до 2^64 байт (16 ЭБ), хотя на практике обычно используется 48-битное адресное пространство (256 ТБ).

**2. Работа со стеком**

·        ARM 32-bit:

o   Использует команды PUSH и POP для работы со стеком.

·        ARM 64-bit:

o   Использует команды STP (Store Pair) и LDP (Load Pair) для работы со стеком.

**3. Вызов подпрограмм**

·        ARM 32-bit:

o   Использует команду BX LR или MOV PC, LR для возврата из подпрограммы.

·        ARM 64-bit:

o   Использует команду RET для возврата из подпрограммы.

Заключение

Архитектура ARM 64-bit (ARMv8 и более поздние) предоставляет ряд улучшений и расширений по сравнению с 32-битной архитектурой (ARMv7 и более ранние). Ключевые отличия включают увеличение размера регистров и адресного пространства, использование нового набора команд AArch64, расширенный набор регистров, улучшенную поддержку векторных операций и более мощные инструменты для управления режимами процессора. Эти улучшения делают 64-битную архитектуру ARM более эффективной и гибкой для современных вычислений.

**В моей лабе**

1. Сравнение ARM-32 bit и ARM-64 bit:

|  |  |  |
| --- | --- | --- |
|  | ARM-32 bit | ARM-64 bit |
| Работа со стеком | Используется регистр r7 как базовый регистр для хранения указателя стекового фрейма:  push    {r7} *//* *Сохраняем r7 в стек*  add     r7, sp, #0 *//* *Устанавливаем r7 на вершину стека*  Для работы со стеком используются push и pop | Не используется отдельный регистр для указателя на вершину стека. Вместо этого используется регистр sp напрямую.  Использует команды STP и LDP для работы со стеком. |
| Загрузка значений чисел с плавающей точкой: | В ARM-32 регистры общего назначения имеют размер 32 бита. Для представления чисел с плавающей точкой двойной точности (64 бита) требуется два 32-битных регистра. Поэтому число с плавающей точкой разбивается на две части: младшие 32 бита и старшие 32 бита.  mov     r2, #0  mov     r3, #0  Регистры r2 и r3 инициализируются нулями, то есть в начале оба регистра содержат 32 нулевых бита.  movt    r3, 16368   movt  устанавливает старшие 16 бит регистра r3 в значение 16368. Младшие 16 бит регистра r3 остаются неизменными (в данном случае они остаются нулевыми). | Константа сразу загружается в регистр:  fmov    d31, 1.0e+0 |
| Преобразование целого числа в число с плавающей точкой двойной точности | Преобразование целого числа в число с плавающей точкой двойной точности происходит в два этапа:   1. Перемещение целого числа в 32-битный регистр с плавающей точкой. 2. Преобразование целого числа в число с плавающей точкой двойной точности и сохранением его в 64-битный регистр   vmov    s15, r3 @ int  *//Перемещаем значение r3 в регистр с плавающей точкой s15*   vcvt.f64.s32    d16, s15  *//Преобразуем целое число в число с плавающей точкой двойной точности* | scvtf   d0, w0 *// Преобразуем целое число в число с плавающей точкой двойной точности* |
| Команды для работы с плавающей точкой двойной точности (64-битным значением)  В пример приведено умножение двух чисел с плавающей точкой | На примере части кода функции factorial:  vldr.64 d17, [r7, #16] *//Загружаем текущий результат факториала(newValue) по адресу [r7 + 16] в регистр d17*          vmul.f64        d16, d17, d16 *//Умножаем текущий результат на i и сохраняем полученный результат в d16*          vstr.64 d16, [r7, #16] *//Сохраняем новый результат(newValue) в стек* | На примере части кода функции factorial:  ldr     d1, [sp, 24] *// Загружаем текущий результат факториала(newValue) по адресу [sp+24] в регистр d1*          fmul    d0, d1, d0 *// Умножаем текущий результат на i и сохраняем полученный результат в d0*          str     d0, [sp, 24] *// Сохраняем новый результат в стек* |

Инструкция movt (Move Top) устанавливает старшие 16 бит регистра r3 в значение 16368. Младшие 16 бит регистра r3 остаются неизменными (в данном случае они остаются нулевыми).

**Что такое 16368 и как это соответствует 1.0:**

В формате IEEE 754 для числа с плавающей точкой двойной точности (64 бита):

* **Знак:** 1 бит (0 для положительных чисел, 1 для отрицательных).
* **Экспонента:** 11 бит.
* **Мантисса:** 52 бита.

Число 1.0 в формате IEEE 754 двойной точности имеет следующие компоненты:

* **Знак:** 0 (положительное число).
* **Экспонента:** 1023 (в десятичной системе) или 01111111111 в двоичной системе.
* **Мантисса:** 0 (все биты мантиссы равны нулю).

В двоичном представлении число 1.0 выглядит так:

0 01111111111 0000000000000000000000000000000000000000000000000000

Теперь, если мы рассмотрим старшие 32 бита этого числа:

01111111 11110000 00000000 00000000

Это значение в десятичной системе равно 16368. Таким образом, установка старших 16 бит регистра r3 в 16368 эквивалентна установке старших 16 бит числа 1.0 в формате IEEE 754.

**Сохранение пары регистров в стек:**

strd    r2, [r7, #16]  // Сохраняем пару регистров r2 и r3 (0.0 и 1.0) в стек по адресу (r7+16)

Инструкция strd (Store Doubleword) сохраняет два регистра (в данном случае r2 и r3) в память по адресу, указанному в r7 плюс смещение 16. Таким образом, в памяти будет сохранено 64-битное значение, представляющее число с плавающей точкой 1.0.

Векторные расширения (SIMD — Single Instruction, Multiple Data) в архитектурах ARM-32 и ARM-64 позволяют выполнять одну операцию над несколькими элементами данных одновременно, что значительно ускоряет обработку данных, особенно в задачах, связанных с обработкой изображений, аудио, видео и научными вычислениями.

ARM-32

В ARM-32 векторные расширения представлены в основном через NEON. NEON — это SIMD-архитектура, которая предоставляет набор регистров и инструкций для параллельной обработки данных.

Основные характеристики NEON:

1. Регистры: NEON использует 16 регистров размером 128 бит (Q0-Q15) или 32 регистра размером 64 бита (D0-D31). Регистры могут быть использованы для хранения векторов различных типов данных (целые числа, числа с плавающей точкой).
2. Инструкции: NEON предоставляет широкий набор инструкций для арифметических операций, логических операций, сдвигов, сравнений и других операций.
3. Поддержка типов данных: NEON поддерживает целые числа (8, 16, 32, 64 бита) и числа с плавающей точкой (32 бита).

ARM-64

В ARM-64 векторные расширения также представлены через NEON, но с некоторыми улучшениями и дополнениями. Кроме того, ARM-64 вводит SVE (Scalable Vector Extension), который является более мощным и гибким векторным расширением.

Основные характеристики NEON в ARM-64:

1. Регистры: В ARM-64 NEON использует 32 регистра размером 128 бит (V0-V31).
2. Инструкции: Инструкции NEON в ARM-64 аналогичны тем, что используются в ARM-32, но с некоторыми дополнениями и улучшениями.
3. Поддержка типов данных: NEON в ARM-64 поддерживает те же типы данных, что и в ARM-32.

SVE (Scalable Vector Extension):

1. Регистры: SVE использует регистры переменной длины, которые могут быть от 128 бит до 2048 бит. Это позволяет более гибко адаптироваться к различным задачам.
2. Инструкции: SVE предоставляет расширенный набор инструкций, включая операции с плавающей точкой, целочисленные операции, логические операции и другие.
3. Поддержка типов данных: SVE поддерживает целые числа (8, 16, 32, 64 бита) и числа с плавающей точкой (32 и 64 бита).

Сравнение:

* ARM-32: Основное векторное расширение — NEON, с фиксированным размером регистров (128 бит).
* ARM-64: NEON с улучшениями и дополнительное расширение SVE с регистрами переменной длины.

Применение:

* NEON: Широко используется в мобильных устройствах, где требуется эффективная обработка мультимедиа (видео, аудио, изображения).
* SVE: Предназначен для более мощных вычислений, таких как научные вычисления, машинное обучение и другие задачи, требующие высокой производительности.

**ARMv8 и набор команд ARM 64 бита**

В конце 2011 года была опубликована новая версия архитектуры, ARMv8. В ней появилось определение архитектуры AArch64, в которой исполняется 64-битный набор команд A64. Поддержка 32-битных команд получила название A32 и исполняется на архитектурах AArch32. Инструкции Thumb поддерживаются в режиме T32, только при использовании 32-битных архитектур. Допускается исполнение 32-битных приложений в 64-битной ОС, и запуск виртуализованной 32-битной ОС при помощи 64-битного гипервизора[[50]](https://ru.wikipedia.org/wiki/ARM_(%D0%B0%D1%80%D1%85%D0%B8%D1%82%D0%B5%D0%BA%D1%82%D1%83%D1%80%D0%B0)#cite_note-50). Applied Micro, AMD, Broadcom, Calxeda, HiSilicon, Samsung, STM и другие заявили о планах по использованию ARMv8. Ядра Cortex-A53 и Cortex-A57, поддерживающие ARMv8, были представлены компанией ARM 30 октября 2012 года[[51]](https://ru.wikipedia.org/wiki/ARM_(%D0%B0%D1%80%D1%85%D0%B8%D1%82%D0%B5%D0%BA%D1%82%D1%83%D1%80%D0%B0)#cite_note-51).

Как AArch32, так и AArch64 поддерживают VFPv3, VFPv4 и *advanced SIMD* (NEON). Также добавлены криптографические инструкции для работы с [AES](https://ru.wikipedia.org/wiki/Advanced_Encryption_Standard), [SHA-1](https://ru.wikipedia.org/wiki/SHA-1) и [SHA-256](https://ru.wikipedia.org/wiki/SHA-2).

Особенности AArch64:

* Новый набор команд A64
* 31 регистр общего назначения, каждый длиной 64 бита
* Отдельные регистры SP и PC
* Инструкции имеют размер 32 бита и многие совпадают с командами A32
* Большинство инструкций работает как с 32-, так и с 64-битными аргументами
* Адреса имеют размер 64 бита
* Улучшения Advanced SIMD (NEON) enhanced
* С 16 до 32 увеличено количество 128-битных регистров, доступных через NEON, VFPv4, криптоинструкции AES, SHA
* Поддерживает вычисления с числами с плавающей точкой двойной точности (64-бит double)
* Полная совместимость с IEEE 754
* Новая система исключений
* Трансляция виртуальных адресов из 48-битного формата работает с помощью существующих механизмов LPAE

**Функции RISC**

Архитектура ARM обладает следующими особенностями RISC:

* 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 бита) для упрощения декодирования за счет снижения плотности кода. Позднее режим Thumb повысил плотность кода.
* Однотактное исполнение

Чтобы компенсировать простой дизайн, в сравнении с современными процессорами вроде Intel 80286 или Motorola 68020 были использованы некоторые особенности дизайна:

* Арифметические инструкции заменяют условные коды, 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 листьев
* Простые, но быстрые, с двумя уровнями приоритетов подсистемы прерываний с включенными банками регистров.
* Архитектура ARM (от англ. Advanced RISC Machine — усовершенствованная RISC-машина; иногда — Acorn RISC Machine) — система команд и семейство описаний и готовых топологий 32-битных и 64-битных микропроцессорных/микроконтроллерных ядер, разрабатываемых компанией ARM Limited.
* ARM — пожалуй, самая распространенная, поскольку она оптимизирована для экономии заряда батареи. ARM64 — это эволюция оригинальной архитектуры ARM, поддерживающая 64-разрядную обработку данных для более мощных вычислений, и она быстро становится стандартом в новых устройствах.

**Отличия архитектур CISC и RISC**

Архитектуры CISC (Complex Instruction Set Computing) и RISC (Reduced Instruction Set Computing) представляют собой два разных подхода к проектированию процессоров. 1. Количество и сложность команд:

* CISC:
  + Большой набор сложных команд, которые могут выполнять несколько операций за один такт
  + Команды могут иметь переменное количество байтов.
  + Пример: x86/x86-64.
* RISC:
  + Меньший набор простых команд, которые выполняют одну операцию за такт
  + Команды имеют фиксированную длину (обычно 4 байта).
  + Пример: ARM.

2. Адресация памяти:

* CISC:
  + Команды могут выполнять операции с данными непосредственно в памяти.
* RISC:
  + Операции с данными обычно выполняются через регистры, а загрузка/сохранение данных в память осуществляется отдельными командами.

3. Регистры:

* CISC:
  + Меньшее количество регистров общего назначения.
* RISC:
  + Большее количество регистров общего назначения.

4. Компиляция кода:

* CISC:
  + Компилятор генерирует более компактный код, так как команды могут выполнять несколько операций.
* RISC:
  + Компилятор генерирует более длинный код, так как команды выполняют только одну операцию.

2. Что такое условные команды? Приведите примеры условных команд в архитектуре ARM.

**Условные команды** — это команды, которые выполняются только при выполнении определенного условия. Условия обычно проверяются на основе флагов, установленных предыдущими командами.

**Примеры условных команд в архитектуре ARM:**

1.  **BEQ (Branch if Equal):**

o   Переход, если результат предыдущей операции равен нулю (флаг Z установлен).

o   Пример: BEQ .L1 — переход к метке .L1, если результат предыдущей операции равен нулю.

2.      **BNE (Branch if Not Equal):**

o   Переход, если результат предыдущей операции не равен нулю (флаг Z сброшен).

o   Пример: BNE .L2 — переход к метке .L2, если результат предыдущей операции не равен нулю.

3.      **BLT (Branch if Less Than):**

o   Переход, если результат предыдущей операции меньше нуля (флаг N установлен).

o   Пример: BLT .L3 — переход к метке .L3, если результат предыдущей операции меньше нуля.

4.      **BGT (Branch if Greater Than):**

o   Переход, если результат предыдущей операции больше нуля (флаг N сброшен, флаг Z сброшен).

o   Пример: BGT .L4 — переход к метке .L4, если результат предыдущей операции больше нуля.

5.      **BLE (Branch if Less Than or Equal):**

o   Переход, если результат предыдущей операции меньше или равен нулю (флаг N установлен или флаг Z установлен).

o   Пример: BLE .L5 — переход к метке .L5, если результат предыдущей операции меньше или равен нулю.

6.      **BGE (Branch if Greater Than or Equal):**

o   Переход, если результат предыдущей операции больше или равен нулю (флаг N сброшен или флаг Z установлен).

o   Пример: BGE .L6 — переход к метке .L6, если результат предыдущей операции больше или равен нулю.

В ARM условные команды реализуются с помощью флагов процессора. Флаги - это специальные биты в регистре состояния процессора, которые изменяются в результате выполнения арифметических и логических операций.

Основные флаги:

• N (отрицательный): устанавливается в 1, если результат операции был отрицательным.

• Z (нулевой): устанавливается в 1, если результат операции был равен 0.

• C (перенос): устанавливается в 1, если в результате операции произошел перенос из старшего бита.

• V (переполнение): устанавливается в 1, если в результате операции произошло переполнение.

Условные суффиксы:

Условные инструкции в ARM обозначаются добавлением суффикса к базовой команде. Суффикс состоит из двух букв, которые указывают на флаг, который должен быть проверен, и на условие, которое должно быть выполнено.

Примеры суффиксов:

• EQ (равно): Z = 1 (результат равен нулю).

• NE (не равно): Z = 0 (результат не равен нулю).

• GT (больше): N = 0, Z = 0 (результат положительный).

• LT (меньше): N = 1 (результат отрицательный).

• GE (больше или равно): N = 0 (результат не отрицательный).

• LE (меньше или равно): Z = 1 или N = 1 (результат равен нулю или отрицательный).