А)Arm - семейство процессоров на принципах reduced instruction set computing (RISC). Используются в телефонах и микроконтроллерах.

Является альтернативой класса архитектур cisc - процессоры создавались в расчете на быструю (относительно процессора) и очень ограниченную по объему память и написанние программ на ассемблере. В последствии было выяснено, что память дешевеет, увеличивается в объеме, но замедляется относительно скорости процессора. Так же появились высокоуровневые языки программирования , развиваются технологии компиляции, поэтому разработана risc.

Б) RISC -особенности : 1) упрощен набор инструкций 2)команды просты в декодировании- они имеют одинаковый простой формат 3) кодируются словом фиксированной длины. 4) отсутствуют команды , команды которые выполнялись бы много тактов 5) большой по сравнению с cisc регистровый файл. Регистры нумеруются , все арифметические команды выполняются над регистрами, а не над оперативной памятью . для работы с памятью есть только чтение - load запись- store, позволяющее считывать значения ячеек памяти в регистр и записать значение из регистров в память (память таким образом не используется для хранения временных значений ( в cisc обычное дело)

6) risc - не рассчитаны на удобство программирования вручную, на языке ассемблера(для компилятора, для компилирования, которые сами сгенерируют код и оптимизируют аатоъматически). 7) risc - менее емкий, занимает больше места и обычно менее удобен для восприятия человеком.

В) Регистры : программисту доступны 16 32-ух битных регистров общего назначения (r0-r15) и 32 битный регистр состояния( cpsr).

Г) регистры общего назначения используются для хранения произвольных целочисленных данных. В использовании большинстве команд они взаимозаменяемы, но некоторые используют конкретные регистры. Например: r15 - счетчик команд, r14- хранения адреса возврата в результате выполнения специальной команды условного перехода , r13- используется как указатель вершины стека.

Д)Регистр состояния хранит однобитовые флаги : знаковый флаг - N

Флаг нуля - z

Флаг переноса -c

Флаг переполнения - v

И еще другие которые отражают состояние процессора.

Е) команды

Определим на группы:

1)Команды преобразования данных: +-\*, умножение со сложением, побитовые логические операции, сравнение.

2) команды передачи данных: копирование данных между регистрами, чтение и запись данных в память, поддерживается автоматическое увеличение регистра указателя до или после обращения в память. Поддерживаются многорегитсровые операции с памятью, использующие сразу несколько регистров общего назначения.

3) команды передачи управления: команды безусловного, условного перехода В архитектуре поддерживаются условное исполнение любой команды в зависимости от значения флагов. Кроме того существует быстрый способ вызова подпрограммы с использованием специальной команды перехода, сохранящей адрес следующей за ней программы в регистре r14.

Для 64-ех битной

Регистры:

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:  
для применения регистров при вызове функций существуют рекомендации Стандарта вызова процедур архитектуры Arm

1. **X0-x7** -для передачи параметров в функцию и возвращении результата
2. При непрямой адресации **регистор X8** используется для передачи адреса блока памяти.
3. В кадре стека сохрнаяются регистры **X9-X15**, что позволяет подпрограмме изменять регистры. Сохраняет ВЫЗЫВАЮЩАЯ функция.
4. Регистры **X16-X18** используются для хранения промежуточных результатом между вызовами функций
5. 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.
6. **Регистр X29** указатель на фрейм стека, еще ***называется FR(***frame pointer)
7. **Х30** – хранит адрес возврата из функции ***LR(link register***)
8. **Плавающая точка в сопросоцессоре неон!!**

В ARMv8 (64-битная архитектура) существует 32 регистра с плавающей запятой, которые могут использоваться для хранения значений с плавающей запятой одинарной точности (32 бита) или двойной точности (64 бита). Эти регистры обозначаются как d0 до d31.

Регистры одинарной точности: s0 до s31 (32 бита).

Регистры двойной точности: d0 до d31 (64 бита).

1. Существует также **FPSR** (регистр состояния с плавающей запятой), который, подобно собственному PSR процессора ARM, содержит информацию о состоянии, которая может потребоваться приложению. Каждый из доступных флагов имеет «ловушку», которая позволяет приложению включать или отключать ловушки, связанные с данной ошибкой.  
   31 24 23 16 15 8 7 0 - Идентификатор системы | Включение прерывания | Управление системой | Флаги исключений  
   флаги исклбчений   
   INX UFL OFL DVZ IVO

DVZ DiVision by Zero

OFL OverFLow

UFL UnderFLow

После выполнения fcmpe, флаги в FPSR устанавливаются в зависимости от результата сравнения:

**Z (Zero)**: Устанавливается, если d0 равно d1.

**C (Carry)**: Устанавливается, если d0 больше или равно d1.

**N (Negative)**: Устанавливается, если d0 меньше d1.

**V (Overflow)**: Устанавливается, если произошло переполнение.

***Дополнительные регистры:***

Регистр **PC – регистр счетчика программ** – 64 бита, содержит адрес текущей инструкции. Перед выполнением каждая инструкция загружается из области памяти на которую ссылается регистр РС, после выполнения инструкции РС автоматически переходит к следующей инструкции.

Регистр **SР – указатель на верхушку стека( хранит адрес)** 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. Регистр SP является 64-битным. К его младшим 32 битам можно обращаться как к отдельному регистру, который называется **WSP**.

Нулевой регситр XZR всегда соержит нудевое значение. Фактически представляет собой тот же г=регистр , что и SP (XZR и SP перекрывают друг друга) Любое чтение из регистра **XZR** возвращает значение 0, а записи в нулевой регистр игнорируются. Также доступна 32-битная форма регистра - **WZR**. Нулевой регистр часто используется в инструкциях, которые устанавливают флаги условий.

*CMP X0, XZR // Сравнивает X0 с нулем*

В некоторых инструкциях, где требуется сохранить значение регистра без изменений, можно использовать XZR

*MOV X1, XZR // X1 будет содержать 0*

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.

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 **PSTATE** хранит флаги состояния, которые могут устанавливаться в зависимости ль 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, если результат представляет отрицательное чисто (негатив)**

**z- флаг нуля уставнавливается если операция равна 0**

**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 которые есть у меня.

|  |  |  |
| --- | --- | --- |
| Stp | Сохраняет значение в память сразу из двух регистров | В качестве первых двух регистров могут выступать либо 64-разрядные X0-X30, либо 32-разрядные W0-W30. В случае с 64-разрядной версией в Xn помещаются младшие 64 разряда, а в Xm старшие 64 разряда. В случае с 32-разрядной версией в Wn помещаются младшие 34 разряда, а в Wm старшие 34 разряда.  **stp** x1, x2, [x0]   // сохраняем данные из Х1 и Х2: в обсласт по адресу x0stp x29, x30, [sp, -96]! Значение регистра x29 сохраняется по адресу sp - 96. Символ ! в конце команды указывает на то, что после выполнения операции сохранения значений регистров, указатель стека sp должен быть обновлен. В данном случае, sp будет уменьшен на 96 байт. |
| str | Сохранение значения в памяти ( одно) |  |
| ldr | Она загружает в регистр адрес данных и затем использует этот адрес для загрузки в регистр реальных данных. | **LDR** Xd, label Здесь Xd - регистр, в который загружается адрес, а label - метка (например, глобальная переменная), данные по адресу которой загружаются. |
| bl | инструкция перехода | выполняет переход и помещает адрес следующей инструкции, которая идет после BL, в регистр **LR** (link register, он же регистр **X30**). отличие, что bl сохраняет в регистр, а call в стек указатель на следующую команду? |
| mov | Помещает значение в регистр |  |
| b | Инструкция b используется для безусловного перехода к указанной метке или адресу. | b не сохраняет адрес возврата в регистр x30 (link register). Это означает, что после выполнения перехода управление не возвращается на инструкцию, следующую за b. |
| bhi | Условные переход(ниже суффиксы) использует регистр PC |  |
| Scvtf  ucvtf | преобразования целых чисел в числа с плавающей точкой.(для преобразования целых чисел со знаком)  преобразования целых чисел в числа с плавающей точкой.( для преобразования целых чисел без знака) |  |
| fdiv | Деление чисел с плавающей точкой и установка флага( выше флаги) | fdiv d0, d1, d2   * **d0**: Регистр, в который будет сохранен результат деления. * **d1**: Регистр, содержащий делимое. * **d2**: Регистр, содержащий делитель. |
| fcmpe | Эта команда выполняет сравнение двух чисел с плавающей запятой и устанавливает флаги в FPSR в зависимости от результата сравнения. |  |
| ldp | Загрузка регистров из памяти | ldp     x29, x30, [sp], 96  - затирает стековый фрейм |
| Ret | Смотрит на PC и возвращается к команде откуда вызвалась функция. |  |
| 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* |
| **CBZ**  **CBNZ** | Проверка на ноль. Если равен переход к метке  если не равен то переход к метке | CBZ Wt/Xt, label  В качестве первого операнда они принимают регистр (64- или 32-разрядный), который нажо проверить на 0, а в качестве второго операнда - метку. Инструкция **CBZ** проверяет регистр и, если он равен 0, то выполняет переход к метке. НА ФЛАГИ НЕ ВЛИЯЕТ. |
| movk | Разница с mov в том что помещает 64 битное значение, а что если нужно поместить значение меньше? Загуржает 16 битное число в часть регистра со сдвигом | Инструкция **MOVK** (move keep) представляет разновидность инструкции MOV и **позволяет загрузить все 64 бита регистра**. В реальности она загружает 16-битные операнды в одну из 4 частей регистра, не оказывая никакого влияния на другие 48 бит. Собственно поэтому в названии инструкции используется буква **K** - сокращение от "keep" - хранить, так как инструкция сохраняет другие 16-битные части регистра. Команда MOVK для заполнения нужной части регистра **выполняет сдвиг на определенное количество бит**, которое должно быть кратно 16.   * MOVK <Rd>, <imm16>, <shift> <Rd>: Регистр назначения. * <imm16>: 16-битное значение, которое будет загружено в регистр. * <shift>: Сдвиг, который определяет, в какой 16-битный сегмент регистра будет загружено значение. Возможные значения: LSL 0, LSL 16, LSL 32, LSL 48.(сдвинуто на {} влево) |
| fmadd | Умножение с последующим сложением | FMADD Dd, Dn, Dm, Da // Dd = Da + Dm \* Dn |

**Условные суффиксы:**

1. **EQ (Equal)**: Переход, если результат сравнения равен.
2. **NE (Not Equal)**: Переход, если результат сравнения не равен.
3. **CS/HS (Carry Set/Unsigned Higher or Same)**: Переход, если есть перенос (для беззнаковых чисел).
4. **CC/LO (Carry Clear/Unsigned Lower)**: Переход, если нет переноса (для беззнаковых чисел).
5. **MI (Minus/Negative)**: Переход, если результат отрицательный.
6. **PL (Plus/Positive or Zero)**: Переход, если результат положительный или ноль.
7. **VS (Overflow Set)**: Переход, если произошло переполнение.
8. **VC (Overflow Clear)**: Переход, если нет переполнения.
9. **HI (Unsigned Higher)**: Переход, если беззнаковое число больше.
10. **LS (Unsigned Lower or Same)**: Переход, если беззнаковое число меньше или равно.
11. **GE (Signed Greater than or Equal)**: Переход, если знаковое число больше или равно.
12. **LT (Signed Less Than)**: Переход, если знаковое число меньше.
13. **GT (Signed Greater Than)**: Переход, если знаковое число больше.
14. **LE (Signed Less than or Equal)**: Переход, если знаковое число меньше или равно.
15. AL (Always): Переход всегда выполняется (этот суффикс обычно опускается).

Для 32-ух битной.

***Регистры :***

R0-r12

R13 = sp

R14=lr

R15=pc

Соглашения   
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

 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.

**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.

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...

|  |  |  |
| --- | --- | --- |
| push{r7, lr} | Сохраняет значения регистров r7 и lr (link register) в стеке. | Сохраняет значения регистров r7 и lr (link register) в стеке. r7 используется как индексный регистр для доступа к локальным переменным, а lr хранит адрес возврата из функции. |
| **str** |  |  |
| **vmov.i32** | Записывает значение в регистр d16 для работы с плавающей точкой |  |
| **vstr.64** | Сохраняет значение d16 (0) в стеке по адресу r7 + 48 |  |
| **strd** | Сохраняет **пару** значений 64 битных(типо два регистра?) в стеке |  |
| **movs r3, #0** | Записывает 0 в регистр r3 | Флаг s указывает на то, что нужно обновить флаги состояния процессора. |
| **mvn** | Записывает значение в регистр | Команда mvn выполняет побитовое НЕ над значением #-2147483648. |
| **vcvt.f64.s32** | Преобразует 32-битное целое число в s15 в 64-битное число с плавающей точкой и сохраняет результат в d18. |  |
| **vdiv.f64** | Делит значение в d18 на значение в d17 и сохраняет результат в d16. ( все с плавающей точкой) |  |
| **vcvt.f64.u32** | для преобразования 32-битного целого числа без знака (unsigned integer) в 64-битное число с плавающей запятой двойной точности |  |
| **vcmpe.f64** | Сравнивает значения в d16 и d17. Результат сравнения записывается во флаги состояния процессора. |  |
| **vmrs APSR\_nzcv, FPSCR:** | Копирует флаги состояния из регистра FPSCR в регистр APSR\_nzcv. |  |
| **vmov**: | : Это команда для перемещения данных между регистрами. |  |
| **CBZ и CBNZ** | -сравнение с 0 |  |
| **vmla.f64** | Умножение с накоплением | Qd = Qd + (Qn \* Qm) |
| **vldm** | загрузки нескольких регистров из памяти. полезна для загрузки векторных данных, что может быть эффективно использовано в вычислениях |  |
| * **!: Символ, указывающий, что регистр sp будет обновлен после загрузки данных.** |  |  |
| * **vmov.i64** | Эта инструкция используется для перемещения 64-битных целых чисел между регистрами NEON или между регистром NEON и памятью. |  |
| * **vmov.f64** | Эта инструкция используется для перемещения 64-битных чисел с плавающей запятой между регистрами VFP или между регистром VFP и памятью. |  |
| * **movw** | инструкция в наборе команд ARM, которая используется для загрузки 16-битного значения (слова) в регистр. |  |
| * **movt** | инструкция в наборе команд ARM, которая используется для загрузки старшей половины 32-битного значения (старших 16 бит) в регистр. Эта инструкция часто используется в сочетании с инструкцией movw для загрузки полного 32-битного значения в регистр. |  |
| * **vpush** | инструкция в наборе команд ARM NEON/VFP (Vector Floating-Point), которая используется для сохранения 64-битных регистров (например, регистров с плавающей запятой d0-d31) в стек. |  |
| * **it ls** | **; Если условие LS (Less than or Same) истинно, выполнить следующую команду** |  |

* **push {r7, lr}**: Сохраняет значения регистров r7 и lr (link register) в стеке. r7 используется как индексный регистр для доступа к локальным переменным, а lr хранит адрес возврата из функции.
* **sub sp, sp, #64**: Выделяет 64 байта в стеке для локальных переменных. sp (stack pointer) уменьшается на 64, чтобы освободить место для локальных переменных.
* **add r7, sp, #0**: Устанавливает r7 равным текущему значению sp. Теперь r7 указывает на вершину стека, которая используется для доступа к локальным переменным.
* **add r7, sp, #0**: Эта команда добавляет значение #0 к значению регистра sp и сохраняет результат в регистр r7. Фактически, это просто копирует значение sp в r7.
* **Почему тут 0?**: Здесь используется #0, чтобы явно показать, что мы просто копируем значение sp в r7. Это делается для того, чтобы r7 указывал на вершину стека, которая используется для доступа к локальным переменным.
* **str r0, [r7, #4]**: Сохраняет значение аргумента r0 (количество итераций) в стеке по адресу r7 + 4. Это делается для того, чтобы сохранить значение аргумента, так как оно будет использоваться в дальнейшем.
* **str r0, [r7, #4]**: Эта команда сохраняет значение регистра r0 в памяти по адресу r7 + 4.

Разные команды используются для разных целей:

* **mov**: Когда нужно скопировать значение между регистрами.
* **str**: Когда нужно сохранить значение в память.
* **ldr**: Когда нужно загрузить значение из памяти.
* **Почему не mov?**: Команда mov используется для копирования значения между регистрами, а не для сохранения значения в памяти. str используется для записи значения в память.
* **Почему не push?**: Команда push используется для сохранения значения в стек, но она также уменьшает значение sp. В данном случае нам нужно сохранить значение в определенном месте стека, поэтому используется str.

Команды с префиксом v относятся к векторным инструкциям, которые могут быть как NEON, так и FPU. В данном случае, vstr.64 — это команда для сохранения 64-битного значения с плавающей точкой, что относится к FPU.

* **vmov.i32 d16, #0**: Записывает значение 0 в регистр d16. d16 — это регистр с плавающей точкой, который используется для хранения промежуточных результатов.
* **vstr.64 d16, [r7, #48]**: Сохраняет значение d16 (0) в стеке по адресу r7 + 48. Это инициализация переменной, которая будет использоваться в дальнейшем.
* **movs r3, #0**: Записывает 0 в регистр r3. Флаг s указывает на то, что нужно обновить флаги состояния процессора.
* **Какой флаг обновил?**: Команда movs обновляет флаги N (отрицательный), Z (ноль), C (перенос), и V (переполнение) в регистре APSR (Application Program Status Register).
* **Регистр флагов**: В данном случае используется регистр APSR, который хранит флаги состояния процессора.

**vmov s15, r3**

* **vmov**: Это команда для перемещения данных между регистрами. В данном случае она используется для перемещения данных между целочисленным регистром и регистром с плавающей точкой.
* **s15**: Это регистр с плавающей точкой (Single Precision) из набора регистров NEON/VFP. В ARM архитектуре регистры с плавающей точкой обозначаются как s0, s1, s2, ..., s31.
* **r3**: Это целочисленный регистр общего назначения.

**Зачем выполнять НЕ?**: Использование mvn вместо прямой записи 0x7FFFFFFF может быть связано с удобством или соглашениями в коде. В данном случае, mvn используется для получения максимального положительного значения для 32-битного целого числа.

* **mvn r3, #-2147483648**: Записывает значение 0x7FFFFFFF (максимальное положительное значение для 32-битного целого) в регистр r3. Команда mvn выполняет побитовое НЕ над значением #-2147483648.
* **movs r3, #0**: Записывает 0 в регистр r3. Флаг s указывает на то, что нужно обновить флаги состояния процессора.
* B - Безусловный переход к метке

**bl rand**: Вызов функции rand()

* **vdiv.f64 d16, d18, d17**: Делит значение в d18 на значение в d17 и сохраняет результат в d16. Это нормализация случайного числа.
* **vadd.f64 d16, d17, d16**: Складывает значения в d17 и d16 и сохраняет результат в d16.
* **vmrs APSR\_nzcv, FPSCR**: Эта команда копирует флаги состояния из регистра FPSCR (Floating-Point Status and Control Register) в регистр APSR\_nzcv (Application Program Status Register, только флаги N, Z, C, V).

**Если команда с приставкой s - то оно еще выставляет флаги?**: Да, команды с суффиксом s обновляют флаги состояния процессора. Например, adds, subs, movs, и т.д.

* **А если без этого суффикса, они не выставляют флаги по умолчанию?**: Да, команды без суффикса s не обновляют флаги состояния процессора.
* **Команды, которые выставляют флаги**:
  + **Целочисленные**: adds, subs, ands, orrs, eors, mvns, cmps, tst, cmp, cmn.
  + **Плавающей точки**: vadd.f32, vsub.f32, vmul.f32, vdiv.f32, vcmp.f32, vcmpe.f32.
* **Thumb** — это режим работы ARM процессора, в котором используются 16-битные инструкции вместо 32-битных. Это позволяет уменьшить размер кода и повысить эффективность использования памяти.
* **Thumb-2** — это расширение Thumb, которое добавляет 32-битные инструкции, сохраняя при этом совместимость с 16-битными Thumb инструкциями.

**IT (If-Then) блоки**

* **IT (If-Then) блоки** — это конструкции, которые позволяют выполнять условные инструкции в Thumb-2 режиме. Они позволяют указать, какие инструкции должны выполняться в зависимости от состояния флагов.
* **Пример**: ITTEE EQ — это инструкция, которая указывает, что следующие 4 инструкции должны выполняться условно:
  + Первая инструкция выполняется, если флаг Z установлен (равен 1).
  + Вторая инструкция выполняется, если флаг Z установлен.
  + Третья инструкция выполняется, если флаг Z не установлен (равен 0).
  + Четвертая инструкция выполняется, если флаг Z не установлен.

**EPSR (Execution Program Status Register)**

* **EPSR** содержит информацию о состоянии выполнения, такую как биты Thumb и IT (If-Then) блоки.
* **Какую информацию он может содержать?**:
  + **Бит Thumb**: Указывает, в каком режиме работает процессор (Thumb или ARM).
  + **IT (If-Then) блоки**: Содержит информацию о текущем состоянии IT блока, включая количество оставшихся условных инструкций и условия для их выполнения.
* **Что не так с блоками If-Then?**: Блоки If-Then являются мощным инструментом для условного выполнения кода в Thumb-2 режиме, но они требуют аккуратного использования, так как неправильное использование может привести к неожиданному поведению программы.

**FPSR (Floating-Point Status Register) и FPSCR (Floating-Point Status and Control Register)**

**FPSR**

* **FPSR** — это часть FPSCR, которая содержит только флаги состояния (без управляющих битов).
* **Флаги состояния**:
  + **N (Negative)**: Устанавливается, если результат отрицательный.
  + **Z (Zero)**: Устанавливается, если результат равен нулю.
  + **C (Carry)**: Устанавливается, если произошел перенос.
  + **V (Overflow)**: Устанавливается, если произошло переполнение.
  + **Флаги исключений**: IXC (Invalid Operation), UFC (Underflow), OFC (Overflow), DZC (Divide by Zero), IOC (Inexact).

**FPSCR**

* **FPSCR** содержит как флаги состояния, так и управляющие биты.
* **Управляющие биты**:
  + **RMode**: Режим округления (Round to Nearest, Round towards Zero, Round towards +Inf, Round towards -Inf).
  + **FZ (Flush to Zero)**: Если установлен, то результаты, которые были бы денормализованными, заменяются нулями.
  + **DN (Default NaN)**: Если установлен, то все NaN (Not a Number) заменяются на стандартный NaN.
  + **AHP (Alternative Half-Precision)**: Режим обработки полуточной арифметики.

**. PC (Program Counter)**

* **Назначение**: pc — это регистр, который указывает на текущую выполняемую инструкцию. Он содержит адрес следующей инструкции, которая должна быть выполнена.
* **Использование**:
  + **Выполнение кода**: pc автоматически увеличивается на размер текущей инструкции (обычно 4 байта для ARM режима и 2 байта для Thumb режима) после выполнения каждой инструкции.
  + **Переходы**: При выполнении команд перехода (например, b, bl, bx), pc обновляется на новый адрес, указанный в команде.

**2. LR (Link Register)**

* **Назначение**: lr — это регистр, который используется для хранения адреса возврата при вызове подпрограмм (функций).
* **Использование**:
  + **Вызов функций**: При выполнении команды bl (Branch with Link), lr сохраняет адрес следующей инструкции после bl, чтобы можно было вернуться к выполнению кода после завершения вызванной функции.
  + **Возврат из функций**: При выполнении команды bx lr или pop {pc} (в случае использования стека), lr используется для возврата к адресу, который был сохранен при вызове функции.
* **bl function**: Сохраняет адрес следующей инструкции после bl в lr и переходит к выполнению кода функции.
* **bx lr**: Возвращает управление к адресу, который был сохранен в lr при вызове функции.

Вопросы группышей:  
**Зачем в RISC уменьшили набор команд**? -Меньший набор команд упрощает дизайн процессора, что уменьшает сложность и стоимость производства.

**1.Предсказуемость**: Простые команды легче предсказать и оптимизировать, что улучшает производительность.

**2.Фиксированная длина команд**: В RISC архитектурах команды обычно имеют фиксированную длину, что упрощает декодирование и выполнение команд.

**3.Быстрое выполнение**: Простые команды могут выполняться за один такт, что увеличивает производительность.

**4.Конвейеризация**: Меньший набор команд упрощает конвейеризацию (pipeline), что позволяет выполнять несколько команд одновременно.

**5.Меньшее энергопотребление**: Простые команды требуют меньше энергии для выполнения, что делает RISC процессоры более энергоэффективными.

**6.Оптимизация**: Энергоэффективность может быть дополнительно улучшена за счет оптимизации выполнения команд.

**7.Простота компиляции**: Меньший набор команд упрощает работу компиляторов, что позволяет генерировать более эффективный машинный код.

**8.Оптимизация**: Компиляторы могут лучше оптимизировать код, так как они имеют дело с более простыми и предсказуемыми командами.

**9.Совместимость**: Меньший набор команд упрощает создание совместимых и масштабируемых систем.

**Примеры RISC архитектур**

* **ARM**: Одна из самых популярных RISC архитектур, используемая в большинстве мобильных устройств и многих встраиваемых системах.

**Сравнение с CISC**

* **CISC (Complex Instruction Set Computing)**: Архитектуры с большим набором команд, такие как x86, которые исторически использовались в персональных компьютерах.
* **Преимущества CISC**: Богатый набор команд, который может упростить написание программ, но при этом увеличивает сложность процессора и может снижать производительность.

**Можно ли в ARM сложить два числа, которые лежат в памяти (не в регистрах)?**-

В ARM архитектуре (как в классической ARM, так и в Thumb режиме) нет прямой команды для сложения двух чисел, которые находятся непосредственно в памяти. Вместо этого, числа должны быть сначала загружены из памяти в регистры, затем выполнено сложение, и результат может быть сохранен обратно в память, если это необходимо.

**Чем заменили регистр rip?** –

**Регистр rip в x86**

* **Назначение**: rip указывает на адрес следующей инструкции, которая должна быть выполнена.
* **Использование**: Используется для управления потоком выполнения программы, включая переходы, вызовы функций и возвраты из функций.

**2. Регистр pc в ARM**

* **Назначение**: pc (Program Counter) также указывает на адрес следующей инструкции, которая должна быть выполнена.
* **Использование**: Аналогично rip, pc используется для управления потоком выполнения программы. Однако в ARM архитектуре pc может быть использован и как обычный регистр для выполнения арифметических операций, хотя это не рекомендуется для стандартного кода.

**Как называются и зачем появились команды типа itt?** -   
Команды типа `itt` (If-Then-Then) в ARM архитектуре появились в рамках расширения Thumb-2, которое было разработано для улучшения производительности и эффективности кода в режиме Thumb. Давайте рассмотрим, как называются и зачем появились эти команды.

### 1. \*\*Команды IT (If-Then)\*\*

- \*\*Название\*\*: Команды IT (If-Then) — это условные блоки, которые позволяют выполнять несколько инструкций условно, основываясь на состоянии флагов.

- \*\*Формат\*\*: Команда IT имеет формат `IT{x{y{z}}} cond`, где:

- `cond` — условие, на которое проверяется флаг.

- `x`, `y`, `z` — буквы `T` (Then) или `E` (Else), которые указывают, какие инструкции должны выполняться при соответствующих условиях.

### 2. \*\*Примеры команд IT\*\*

#### Пример 1: ITT (If-Then-Then)

```assembly

cmp r0, #0 ; Сравниваем r0 с 0

itt eq ; Если r0 == 0, то выполняем следующие две инструкции

moveq r1, #1 ; Если условие выполнено, то r1 = 1

addeq r2, r2, #10 ; Если условие выполнено, то r2 = r2 + 10

```

- \*\*Описание\*\*: Если результат сравнения `r0` с `0` равен `eq` (равно), то выполняются следующие две инструкции: `moveq r1, #1` и `addeq r2, r2, #10`.

#### Пример 2: ITE (If-Then-Else)

```assembly

cmp r0, #0 ; Сравниваем r0 с 0

ite eq ; Если r0 == 0, то выполняем первую инструкцию, иначе вторую

moveq r1, #1 ; Если условие выполнено, то r1 = 1

movne r1, #2 ; Если условие не выполнено, то r1 = 2

```

- \*\*Описание\*\*: Если результат сравнения `r0` с `0` равен `eq` (равно), то выполняется первая инструкция `moveq r1, #1`. Если условие не выполнено, то выполняется вторая инструкция `movne r1, #2`.

### 3. \*\*Зачем появились команды IT?\*\*

- \*\*Улучшение производительности\*\*: Команды IT позволяют выполнять несколько инструкций условно, что уменьшает количество переходов и улучшает производительность.

- \*\*Эффективность кода\*\*: В режиме Thumb, где инструкции имеют фиксированную длину 16 бит, команды IT позволяют более эффективно использовать память и улучшать плотность кода.

- \*\*Упрощение программирования\*\*: Команды IT упрощают написание условных конструкций, делая код более читаемым и понятным.

**Как работать с вещественными числами в ARM?**-

Работа с вещественными числами (числами с плавающей точкой) в ARM архитектуре осуществляется с использованием регистров с плавающей точкой (FPU) и набора команд, предназначенных для операций с плавающей точкой. ARM поддерживает как одинарную (32-битную) точность (single precision), так и двойную (64-битную) точность (double precision).

### Регистры с плавающей точкой

- \*\*Регистры одинарной точности (Single Precision)\*\*: Обозначаются как `s0`, `s1`, `s2`, ..., `s31`.

- \*\*Регистры двойной точности (Double Precision)\*\*: Обозначаются как `d0`, `d1`, `d2`, ..., `d15`. Каждый регистр `d` соответствует двум регистрам `s` (например, `d0` соответствует `s0` и `s1`).

### Команды для работы с вещественными числами

- \*\*Перемещение данных\*\*:

- `vmov`: Перемещает данные между регистрами с плавающей точкой и целочисленными регистрами.

- Пример: `vmov s0, r0` — копирует значение из целочисленного регистра `r0` в регистр с плавающей точкой `s0`.

- \*\*Арифметические операции\*\*:

- `vadd`, `vsub`, `vmul`, `vdiv`: Сложение, вычитание, умножение и деление.

- Пример: `vadd.f32 s0, s1, s2` — складывает значения в регистрах `s1` и `s2`, результат сохраняет в `s0`.

- \*\*Сравнение\*\*:

- `vcmp`: Сравнивает два значения с плавающей точкой.

- Пример: `vcmp.f32 s0, s1` — сравнивает значения в регистрах `s0` и `s1`, результат записывается в флаги состояния.

- \*\*Преобразование типов\*\*:

- `vcvt`: Преобразует значения между целочисленными и вещественными типами.

- Пример: `vcvt.f32.s32 s0, s1` — преобразует 32-битное целое число в регистре `s1` в 32-битное число с плавающей точкой и сохраняет результат в `s0`.

### Стек с плавающей точкой

ARM архитектура поддерживает стек с плавающей точкой, который используется для хранения и извлечения значений с плавающей точкой. Стек с плавающей точкой работает по принципу LIFO (Last In, First Out).

- \*\*Загрузка и сохранение в стек\*\*:

- `vpush`: Сохраняет значения регистров с плавающей точкой в стек.

- `vpop`: Извлекает значения из стека в регистры с плавающей точкой.

- \*\*Пример использования стека\*\*:

```assembly

vpush {s0-s3} ; Сохраняем значения регистров s0-s3 в стек

; Код, который может изменить значения s0-s3

vpop {s0-s3} ; Восстанавливаем значения регистров s0-s3 из стека

```

### Пример работы с вещественными числами

```assembly

vmov.f32 s0, #1.0 ; Записываем значение 1.0 в регистр s0

vmov.f32 s1, #2.0 ; Записываем значение 2.0 в регистр s1

vadd.f32 s2, s0, s1 ; Складываем значения в s0 и s1, результат сохраняем в s2

vcvt.f64.f32 d0, s2 ; Преобразуем значение в s2 в 64-битное число с плавающей точкой и сохраняем в d0

vpush {d0} ; Сохраняем значение d0 в стек

; Код, который может изменить значение d0

vpop {d0} ; Восстанавливаем значение d0 из стека

```

**Как в ARM реализована работа с векторами чисел? (какой набор команд и какие регистры?)** –

Работа с векторами чисел в ARM архитектуре реализована с использованием расширения NEON, которое предоставляет набор команд и регистров для выполнения векторных операций. NEON позволяет выполнять параллельные вычисления над массивами данных, что ускоряет обработку видео, аудио, графики и других задач, требующих интенсивных вычислений.

### Регистры NEON

- \*\*Регистры одинарной точности (Single Precision)\*\*: Обозначаются как `s0`, `s1`, `s2`, ..., `s31`.

- \*\*Регистры двойной точности (Double Precision)\*\*: Обозначаются как `d0`, `d1`, `d2`, ..., `d15`. Каждый регистр `d` соответствует двум регистрам `s` (например, `d0` соответствует `s0` и `s1`).

- \*\*Векторные регистры (NEON)\*\*: Обозначаются как `q0`, `q1`, `q2`, ..., `q15`. Каждый регистр `q` соответствует двум регистрам `d` (например, `q0` соответствует `d0` и `d1`).

### Команды NEON

NEON предоставляет широкий набор команд для выполнения векторных операций, включая арифметические операции, логические операции, сдвиги, перемещение данных и другие.

#### 1. \*\*Арифметические операции\*\*

- \*\*Сложение\*\*: `vadd`, `vaddq`

- \*\*Вычитание\*\*: `vsub`, `vsubq`

- \*\*Умножение\*\*: `vmul`, `vmulq`

- \*\*Деление\*\*: `vdiv`, `vdivq`

Пример:

```assembly

vadd.i32 q0, q1, q2 ; Складываем векторы q1 и q2, результат сохраняем в q0

```

#### 2. \*\*Логические операции\*\*

- \*\*И\*\*: `vand`, `vandq`

- \*\*ИЛИ\*\*: `vorr`, `vorrq`

- \*\*Исключающее ИЛИ\*\*: `veor`, `veorq`

Пример:

```assembly

vand.i32 q0, q1, q2 ; Выполняем побитовое И между векторами q1 и q2, результат сохраняем в q0

```

#### 3. \*\*Сдвиги\*\*

- \*\*Арифметический сдвиг\*\*: `vshl`, `vshlq`

- \*\*Логический сдвиг\*\*: `vshr`, `vshrq`

Пример:

```assembly

vshl.i32 q0, q1, #2 ; Сдвигаем каждый элемент вектора q1 на 2 бита влево, результат сохраняем в q0

```

#### 4. \*\*Перемещение данных\*\*

- \*\*Загрузка данных\*\*: `vld1`, `vld1q`

- \*\*Сохранение данных\*\*: `vst1`, `vst1q`

Пример:

```assembly

vld1.32 {d0, d1}, [r0] ; Загружаем данные из памяти по адресу r0 в регистры d0 и d1

vst1.32 {d0, d1}, [r1] ; Сохраняем данные из регистров d0 и d1 в память по адресу r1

```

#### 5. \*\*Преобразование типов\*\*

- \*\*Преобразование между целочисленными и вещественными типами\*\*: `vcvt`, `vcvtq`

Пример:

```assembly

vcvt.f32.s32 q0, q1 ; Преобразуем 32-битные целые числа в q1 в 32-битные числа с плавающей точкой и сохраняем результат в q0

```

### Пример использования NEON

```assembly

vld1.32 {d0, d1}, [r0] ; Загружаем данные из памяти по адресу r0 в регистры d0 и d1

vld1.32 {d2, d3}, [r1] ; Загружаем данные из памяти по адресу r1 в регистры d2 и d3

vadd.i32 q0, q1, q2 ; Складываем векторы q1 (d2, d3) и q2 (d0, d1), результат сохраняем в q0

vst1.32 {d0, d1}, [r2] ; Сохраняем данные из регистров d0 и d1 в память по адресу r2

```

**Зачем появились команды ld и st?-**

Команды `ld` (Load) и `st` (Store) появились в архитектуре процессоров для обеспечения эффективного доступа к памяти и перемещения данных между регистрами и памятью. Давайте рассмотрим, зачем появились эти команды и как они используются.

### 1. \*\*Зачем появились команды `ld` и `st`?\*\*

#### Основные причины:

- \*\*Доступ к памяти\*\*: Процессоры должны иметь возможность читать данные из памяти и записывать данные в память. Команды `ld` и `st` предоставляют механизм для этого.

- \*\*Перемещение данных\*\*: Для выполнения многих операций (арифметических, логических, преобразований и т.д.) данные должны быть перемещены между регистрами и памятью. Команды `ld` и `st` упрощают этот процесс.

- \*\*Управление потоком данных\*\*: В программах часто требуется динамически управлять потоком данных, что невозможно без команд загрузки и сохранения.

### 2. \*\*Как работают команды `ld` и `st`?\*\*

#### Команда `ld` (Load)

- \*\*Назначение\*\*: Загружает данные из памяти в регистр.

- \*\*Формат\*\*: `ld{cond} {size} Rd, [Rn {, #offset}]`

- `cond`: Условие выполнения команды.

- `size`: Размер данных (например, `.b` для байта, `.h` для полуслова, `.w` для слова).

- `Rd`: Регистр назначения.

- `Rn`: Регистр, содержащий базовый адрес памяти.

- `offset`: Смещение относительно базового адреса.

Пример:

```assembly

ldr r0, [r1] ; Загружаем данные из памяти по адресу, хранящемуся в r1, в регистр r0

ldr r0, [r1, #4] ; Загружаем данные из памяти по адресу r1 + 4 в регистр r0

```

#### Команда `st` (Store)

- \*\*Назначение\*\*: Сохраняет данные из регистра в память.

- \*\*Формат\*\*: `st{cond} {size} Rd, [Rn {, #offset}]`

- `cond`: Условие выполнения команды.

- `size`: Размер данных (например, `.b` для байта, `.h` для полуслова, `.w` для слова).

- `Rd`: Регистр, содержащий данные для сохранения.

- `Rn`: Регистр, содержащий базовый адрес памяти.

- `offset`: Смещение относительно базового адреса.

Пример:

```assembly

str r0, [r1] ; Сохраняем данные из регистра r0 в память по адресу, хранящемуся в r1

str r0, [r1, #4] ; Сохраняем данные из регистра r0 в память по адресу r1 + 4

```

### 3. \*\*Пример использования `ld` и `st`\*\*

Предположим, у нас есть массив данных в памяти, и мы хотим скопировать его в другой массив.

```assembly

ldr r0, =src\_array ; Загружаем адрес исходного массива в r0

ldr r1, =dst\_array ; Загружаем адрес целевого массива в r1

mov r2, #10 ; Количество элементов для копирования

loop:

ldr r3, [r0], #4 ; Загружаем элемент из исходного массива в r3, увеличиваем r0 на 4

str r3, [r1], #4 ; Сохраняем элемент в целевой массив, увеличиваем r1 на 4

subs r2, r2, #1 ; Уменьшаем счетчик элементов

bne loop ; Если счетчик не равен нулю, повторяем цикл

src\_array:

.word 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ; Исходный массив

dst\_array:

.space 40 ; Целевой массив (40 байт для 10 элементов)

```

### Заключение

Команды `ld` (Load) и `st` (Store) появились в архитектуре процессоров для обеспечения эффективного доступа к памяти и перемещения данных между регистрами и памятью. Они позволяют процессору читать данные из памяти, записывать данные в память и управлять потоком данных в программах. Эти команды являются фундаментальными для работы с памятью и перемещения данных в процессорах.

**Почему два флаговых регистра? –**

В архитектуре ARM процессоров используются два флаговых регистра: `APSR` (Application Program Status Register) и `FPSCR` (Floating-Point Status and Control Register). Эти регистры предназначены для хранения флагов состояния, которые отражают результаты выполнения различных операций. Давайте рассмотрим, почему используются два флаговых регистра и для чего каждый из них предназначен.

### 1. \*\*APSR (Application Program Status Register)\*\*

- \*\*Назначение\*\*: `APSR` используется для хранения флагов состояния, которые отражают результаты выполнения целочисленных операций.

- \*\*Флаги\*\*:

- \*\*N (Negative)\*\*: Устанавливается, если результат отрицательный.

- \*\*Z (Zero)\*\*: Устанавливается, если результат равен нулю.

- \*\*C (Carry)\*\*: Устанавливается, если произошел перенос.

- \*\*V (Overflow)\*\*: Устанавливается, если произошло переполнение.

- \*\*Использование\*\*: `APSR` используется для условного выполнения команд, таких как `beq`, `bne`, `bgt`, `blt` и других, которые основываются на результатах целочисленных операций.

### 2. \*\*FPSCR (Floating-Point Status and Control Register)\*\*

- \*\*Назначение\*\*: `FPSCR` используется для хранения флагов состояния и управления для операций с плавающей точкой.

- \*\*Флаги\*\*:

- \*\*N (Negative)\*\*: Устанавливается, если результат отрицательный.

- \*\*Z (Zero)\*\*: Устанавливается, если результат равен нулю.

- \*\*C (Carry)\*\*: Устанавливается, если произошел перенос.

- \*\*V (Overflow)\*\*: Устанавливается, если произошло переполнение.

- \*\*Флаги исключений\*\*: `IXC` (Invalid Operation), `UFC` (Underflow), `OFC` (Overflow), `DZC` (Divide by Zero), `IOC` (Inexact).

- \*\*Управляющие биты\*\*:

- \*\*RMode\*\*: Режим округления (Round to Nearest, Round towards Zero, Round towards +Inf, Round towards -Inf).

- \*\*FZ (Flush to Zero)\*\*: Если установлен, то результаты, которые были бы денормализованными, заменяются нулями.

- \*\*DN (Default NaN)\*\*: Если установлен, то все NaN (Not a Number) заменяются на стандартный NaN.

- \*\*AHP (Alternative Half-Precision)\*\*: Режим обработки полуточной арифметики.

- \*\*Использование\*\*: `FPSCR` используется для условного выполнения команд с плавающей точкой, таких как `vcmp`, `vcmpe`, и для управления поведением операций с плавающей точкой.

### Почему два флаговых регистра?

- \*\*Разделение ответственности\*\*: Разделение флагов на два регистра позволяет четко разделить ответственность за целочисленные и вещественные операции. Это упрощает дизайн процессора и улучшает производительность.

- \*\*Управление исключениями\*\*: `FPSCR` содержит флаги исключений, специфичные для операций с плавающей точкой, что позволяет более точно управлять и обрабатывать исключения, возникающие при выполнении таких операций.

- \*\*Оптимизация\*\*: Разделение флагов на два регистра позволяет оптимизировать выполнение команд, так как процессор может быстрее определять, какие флаги нужно обновлять в зависимости от типа операции.

### Пример использования

#### Целочисленные операции

```assembly

mov r0, #10

mov r1, #5

subs r2, r0, r1 ; Вычитаем r1 из r0, результат в r2

bne not\_equal ; Если результат не равен нулю, переходим к not\_equal

equal:

; Код, если результат равен нулю

b end

not\_equal:

; Код, если результат не равен нулю

end:

```

#### Операции с плавающей точкой

```assembly

vmov.f32 s0, #1.0

vmov.f32 s1, #2.0

vsub.f32 s2, s0, s1 ; Вычитаем s1 из s0, результат в s2

vcmp.f32 s2, #0.0 ; Сравниваем результат с 0.0

vmrs APSR\_nzcv, FPSCR ; Копируем флаги из FPSCR в APSR\_nzcv

bne not\_equal\_fp ; Если результат не равен нулю, переходим к not\_equal\_fp

equal\_fp:

; Код, если результат равен нулю

b end\_fp

not\_equal\_fp:

; Код, если результат не равен нулю

end\_fp:

```

### Заключение

Использование двух флаговых регистров (`APSR` и `FPSCR`) в ARM архитектуре позволяет четко разделить ответственность за целочисленные и вещественные операции, упрощает дизайн процессора, улучшает производительность и обеспечивает более точную обработку исключений. Это разделение позволяет процессору эффективно управлять потоком выполнения программы и оптимизировать выполнение команд.

**Что такое векторные операции? –**

Векторные операции — это операции, которые выполняются над массивами данных (векторами) одновременно, что позволяет ускорить вычисления и повысить производительность. Векторные операции широко используются в различных областях, таких как обработка изображений, аудио, видео, научные вычисления, машинное обучение и другие задачи, требующие интенсивных вычислений.

### Основные понятия

- \*\*Вектор\*\*: Массив данных, состоящий из нескольких элементов одного типа. Например, вектор целых чисел или вектор чисел с плавающей точкой.

- \*\*Параллельные вычисления\*\*: Векторные операции позволяют выполнять одну и ту же операцию над несколькими элементами вектора одновременно, что ускоряет вычисления.

- \*\*SIMD (Single Instruction, Multiple Data)\*\*: Принцип выполнения одной инструкции над несколькими элементами данных одновременно.

### Примеры векторных операций

#### 1. \*\*Сложение векторов\*\*

Сложение двух векторов `A` и `B` с результатом в векторе `C`:

```

A = [a1, a2, a3, a4]

B = [b1, b2, b3, b4]

C = A + B = [a1 + b1, a2 + b2, a3 + b3, a4 + b4]

```

#### 2. \*\*Умножение векторов\*\*

Умножение двух векторов `A` и `B` с результатом в векторе `C`:

```

A = [a1, a2, a3, a4]

B = [b1, b2, b3, b4]

C = A \* B = [a1 \* b1, a2 \* b2, a3 \* b3, a4 \* b4]

```

#### 3. \*\*Сравнение векторов\*\*

Сравнение двух векторов `A` и `B` с результатом в векторе `C`:

```

A = [a1, a2, a3, a4]

B = [b1, b2, b3, b4]

C = A > B = [a1 > b1, a2 > b2, a3 > b3, a4 > b4]

```

### Примеры использования векторных операций

#### 1. \*\*Обработка изображений\*\*

При обработке изображений часто требуется выполнять операции над каждым пикселем изображения. Векторные операции позволяют обрабатывать несколько пикселей одновременно, что ускоряет процесс.

Пример:

```assembly

vld1.8 {q0}, [r0] ; Загружаем 16 байт данных из памяти по адресу r0 в регистр q0

vld1.8 {q1}, [r1] ; Загружаем 16 байт данных из памяти по адресу r1 в регистр q1

vadd.u8 q2, q0, q1 ; Складываем данные в q0 и q1, результат сохраняем в q2

vst1.8 {q2}, [r2] ; Сохраняем данные из q2 в память по адресу r2

```

#### 2. \*\*Аудио обработка\*\*

При обработке аудио данных векторные операции позволяют выполнять операции над несколькими отсчетами аудиосигнала одновременно.

Пример:

```assembly

vld1.16 {q0}, [r0] ; Загружаем 8 полуслов данных из памяти по адресу r0 в регистр q0

vld1.16 {q1}, [r1] ; Загружаем 8 полуслов данных из памяти по адресу r1 в регистр q1

vsub.s16 q2, q0, q1 ; Вычитаем данные в q1 из q0, результат сохраняем в q2

vst1.16 {q2}, [r2] ; Сохраняем данные из q2 в память по адресу r2

```

#### 3. \*\*Научные вычисления\*\*

В научных вычислениях векторные операции позволяют выполнять матричные операции, такие как умножение матриц, быстрее.

Пример:

```assembly

vld1.32 {q0, q1}, [r0] ; Загружаем 8 слов данных из памяти по адресу r0 в регистры q0 и q1

vld1.32 {q2, q3}, [r1] ; Загружаем 8 слов данных из памяти по адресу r1 в регистры q2 и q3

vmul.f32 q4, q0, q2 ; Умножаем данные в q0 и q2, результат сохраняем в q4

vmla.f32 q4, q1, q3 ; Умножаем данные в q1 и q3, добавляем результат к q4

vst1.32 {q4}, [r2] ; Сохраняем данные из q4 в память по адресу r2

```

### Заключение

Векторные операции — это мощный инструмент для ускорения вычислений, который позволяет выполнять одну и ту же операцию над несколькими элементами данных одновременно. Они широко используются в различных областях, таких как обработка изображений, аудио, видео, научные вычисления и машинное обучение. Векторные операции реализуются с использованием расширений процессоров, таких как NEON в ARM архитектуре, что позволяет эффективно использовать параллелизм данных и повысить производительность приложений.

**Как АРМ оптимизировал работу с условными переходами и зачем? (что-то там про it) –**

В ARM архитектуре условные переходы и условные выполнения команд играют важную роль в управлении потоком выполнения программы. Оптимизация работы с условными переходами позволяет улучшить производительность и эффективность кода. Одним из ключевых механизмов, который ARM использует для оптимизации условных переходов, являются команды IT (If-Then).

### Команды IT (If-Then)

Команды IT (If-Then) появились в ARM архитектуре с расширением Thumb-2 и позволяют выполнять несколько инструкций условно, основываясь на состоянии флагов. Это уменьшает количество переходов в коде и улучшает производительность.

#### Формат команды IT

Команда IT имеет формат `IT{x{y{z}}} cond`, где:

- `cond` — условие, на которое проверяется флаг.

- `x`, `y`, `z` — буквы `T` (Then) или `E` (Else), которые указывают, какие инструкции должны выполняться при соответствующих условиях.

#### Примеры использования команд IT

##### Пример 1: ITT (If-Then-Then)

```assembly

cmp r0, #0 ; Сравниваем r0 с 0

itt eq ; Если r0 == 0, то выполняем следующие две инструкции

moveq r1, #1 ; Если условие выполнено, то r1 = 1

addeq r2, r2, #10 ; Если условие выполнено, то r2 = r2 + 10

```

- \*\*Описание\*\*: Если результат сравнения `r0` с `0` равен `eq` (равно), то выполняются следующие две инструкции: `moveq r1, #1` и `addeq r2, r2, #10`.

##### Пример 2: ITE (If-Then-Else)

```assembly

cmp r0, #0 ; Сравниваем r0 с 0

ite eq ; Если r0 == 0, то выполняем первую инструкцию, иначе вторую

moveq r1, #1 ; Если условие выполнено, то r1 = 1

movne r1, #2 ; Если условие не выполнено, то r1 = 2

```

- \*\*Описание\*\*: Если результат сравнения `r0` с `0` равен `eq` (равно), то выполняется первая инструкция `moveq r1, #1`. Если условие не выполнено, то выполняется вторая инструкция `movne r1, #2`.

### Зачем оптимизировать условные переходы?

#### 1. \*\*Уменьшение количества переходов\*\*

- \*\*Проблема\*\*: Частые условные переходы могут привести к снижению производительности из-за необходимости перезагрузки конвейера процессора.

- \*\*Решение\*\*: Команды IT позволяют выполнять несколько инструкций условно, что уменьшает количество переходов и улучшает производительность.

#### 2. \*\*Улучшение плотности кода\*\*

- \*\*Проблема\*\*: В режиме Thumb, где инструкции имеют фиксированную длину 16 бит, условные переходы могут занимать много места.

- \*\*Решение\*\*: Команды IT позволяют более эффективно использовать память, улучшая плотность кода.

#### 3. \*\*Упрощение программирования\*\*

- \*\*Проблема\*\*: Написание условных конструкций может быть сложным и занимать много строк кода.

- \*\*Решение\*\*: Команды IT упрощают написание условных конструкций, делая код более читаемым и понятным.

### Пример оптимизации с использованием команд IT

#### Без использования команд IT

```assembly

cmp r0, #0 ; Сравниваем r0 с 0

beq equal ; Если r0 == 0, переходим к метке equal

mov r1, #2 ; Если условие не выполнено, то r1 = 2

b end ; Переходим к метке end

equal:

mov r1, #1 ; Если условие выполнено, то r1 = 1

end:

```

#### С использованием команд IT

```assembly

cmp r0, #0 ; Сравниваем r0 с 0

ite eq ; Если r0 == 0, то выполняем первую инструкцию, иначе вторую

moveq r1, #1 ; Если условие выполнено, то r1 = 1

movne r1, #2 ; Если условие не выполнено, то r1 = 2

```

### Заключение

Оптимизация работы с условными переходами в ARM архитектуре с использованием команд IT позволяет уменьшить количество переходов, улучшить плотность кода и упростить программирование. Команды IT позволяют выполнять несколько инструкций условно, основываясь на состоянии флагов, что улучшает производительность и эффективность кода. Этот механизм особенно полезен в режиме Thumb, где инструкции имеют фиксированную длину 16 бит.