***Введение в Ассемблерм RISC-V.***

*Директивы определения разделов (сегментов).*

Программа на Ассемблере состоит из директив (рассматриваются на этапе трансляции), инструкций (выполняются при запуске программы) и данных. Все они хранятся в соответствующих разделах в соответствии с их назначением. Разделы определяются с помощью директив. Основными разделами являются:

1. Раздел TEXT

Раздел, доступный только для чтения, содержит фактические инструкции программы.

*Синтаксис определения*: .text

*Описание*.

Этот раздел также известен как сегмент кода. Он содержит исполняемые инструкции, которые не могут быть изменены во время выполнения. Попытка сохранить что-то в раздел text выдаст ошибку и программа немедленно завершится. Сегмент кода может в дополнение к инструкциям также содержать константы.

*Примеры использования*:

.text

li x5, 100

addi x5, x0, 100

1. Раздел DATA

Раздел, доступный для чтения и записи, содержит данные для переменных программы.

*Синтаксис*: .data Переменные

*Описание*.

Раздел .data содержит инициализированные статические переменные, которые являются глобальными или статическими локальными переменными.

*Примеры использования*:

.data

а: .word 1

helloworld: .ascii " Hello World!"

1. Раздел BSS

Раздел для чтения и записи, содержащий неинициализированные данные.

*Синтаксис*: .bss symbol, length, align , где

symbol - локальный символ

length - число резервируемых байт по длине символа

align - выравнивание до целой степени 2

*Описание*.

Раздел BSS используется для хранения локальных переменных. Когда программа начинает работать, все содержимое этого раздела обнуляется в байтах, поэтому нет необходимости явно сохранять нулевые байты в объектном файле. Раздел .BSS был введен для устранения этих явных нулей из объектных файлов. В программе раздел BSS следует за разделом данных.

*Пример использования*

.bss label1, 8, 4

*Директивы для определения и экспорта символов*

1. GLOBAL

Директива .global определяет символ глобальным.

*Синтаксис*: .globаl (.globl) symbol,

где symbol - переменная, имя (значение) которой должно быть доступно для всей программы.

*Описание*.

Обычно значение символа доступно только в части программы, в которой он определен. После объявления директивой .global, его значение становится доступным для других частей программы, которые связаны с ней.

Пример использования

i: .word 5

.global i # Переменная i становится глобальной

1. LOCAL

Директива .local ограничивает видимость символов.

*Синтаксис*: .local symbol, где symbol - имя локальной переменной

*Описание.*

Директива .local помечает каждый символ в списке имен, разделенном запятыми, как локальный символ, так что он не будет виден извне. Если символы еще не существуют, они будут созданы.

*Пример использования*

i: .word 5

.local i # Переменная i становится локальной

1. EQU

Директива .equ задает значение символа в выражении.

*Синтаксис*: .equ symbol, выражение , где symbol – локальный символ

*Описание.*

Директива .equ имеет два операнда, разделенных запятой. Везде, где в программе есть первый операнд, ассемблер заменяет его вторым . Используется при сборке вашего кода, как только символ определен, его значение не может быть изменено в оставшейся части исходного кода.

*Пример использования*:

.equ counter, 3 # counter ← 3

*Директивы ассемблера для задания данных*

1. BYTE

Директива .byte - служит для задания невыровненных 8-битных слов, разделенных запятыми.

*Синтаксис*: .byte value , где value - инициализируемое значение

*Описание.* Директива .byte инициализирует значение в 1 байт или 8-разрядное целое числа без выравнивания. Может задавать несколько значений, разделенных запятыми. Они могут быть десятичными, шестнадца-теричными, двоичными или символьными константами, но не метками.

*Пример использования*: .byte 0x10

1. 2BYTE

Директива .2byte служит для задания невыровненных 2-байтовых или 16-битных слов, разделенных запятыми.

*Синтаксис*: .2byte value , где value - инициализируемое значение.

*Описание.* Директива .2byte инициализирует значение в 2 - байтовые или 16-разрядные невыровненные целые числа. Может задавать несколько значений, разделенных запятыми, они могут быть десятичными, шестнад-цатеричными, двоичными или символьными константами, но не метками.

*Пример использования*:

.2byte 0x1000

1. 4BYTE

Директива .4byte служит для задания невыровненных 4-байтовых или 32-битных слов, разделенных запятыми.

*Синтаксис*: .4byte value , где value - инициализируемое значение.

*Описание.* Директива .4byte инициализирует значение в 4- байтовые или 32-разрядные невыровненные целые числа. Может задавать несколько значений, разделенных запятыми. Они могут быть десятичными, шестнадца-теричными, двоичными или символьными константами, но не метками.

*Пример использования*:

.4byte 0x1000000

1. HALF

Директива .half - служит для задания естественно выровненных 2-байтовых или 16-битных слов, разделенных запятыми.

*Синтаксис*: .half value , где value - инициализируемое значение

*Описание.* Директива .half инициализирует значение в 2 - байтовые или 16-разрядные выровненные целые числа. Может задавать несколько значений, разделенных запятыми, Они могут быть десятичными, шестнадца-теричными, двоичными или символьными константами, но не метками.

*Пример использования*:

.half 0x1000

1. WORD

Директива .word - служит для задания естественно выровненных 4-байтовых или 32-битных слов, разделенных запятыми.

*Синтаксис*: .word value, где value - инициализируемое значение

*Описание*. Директива .word инициализирует значение в 4 - байтовые или 32-разрядные выровненные целые числа. Может задавать несколько значений, разделенных запятыми. Они могут быть десятичными, шестнадца-теричными, двоичными или символьными константами, но не метками.

*Пример использования*:

.word 0x40000000

1. DWORD

Директива .dword - служит для задания естественно выровненных 8-байтовых или 64-битных слов, разделенных запятыми.

*Синтаксис*: .dword value, где value - инициализируемое значение

*Описание*. Директива .dword инициализирует значение в 8 - байтовые или 64-разрядные выровненные целые числа. Может задавать несколько значений, разделенных запятыми. Они могут быть десятичными, шестнадца-теричными, двоичными или символьными константами, но не метками.

*Пример использования*:

.dword 0x7000000000000000

*Директивы задания строк.*

1. ASCIZ

Директива .asciz – подобна директиве .ascii и задает строку в пределах двойных кавычек.

*Синтаксис*: .asciz "string", где "string" - задаваемая пользователем строка.

*Описание.* По директиве .asciz за каждой строкой следует нулевой байт. Для этой директивы ассемблер увеличивает счетчик местоположения на длину строки, включая нулевой символ в конце.

*Пример использования*:

.asciz "Hello World"

1. STRING

Директива .string – задает строку в пределах двойных кавычек.

*Синтаксис*: .string "string", где "string" - задаваемая пользователем строка.

*Описание.* По директиве .string за каждой строкой следует нулевой байт. Для этой директивы ассемблер увеличивает счетчик местоположения на длину строки, включая нулевой символ в конце.

*Пример использования*:

.string "Hello World"

Псевдоинструкции

Многие команды программ на ассемблере RISC-V не используют три аргумента, так как являются псевдоинструкциями. Это означает, что они являются сокращениями для других инструкций.

Например, рассмотрим инструкцию NEG:

NEG x2, x4

Она берёт содержимое x4 , и помещает его отрицательное значение в  x2. На самом деле записывается его значение в дополнительном коде. Посмотрим, какую реальную инструкцию представляет  NEG :

SUB x2, zero, x4 # x2 ← zero - x4

Эта реальная инструкция называется расширением псевдоинструкции.

Из примера можно видеть некоторые преимущества использования регистра "zero", он упрощает создание множества таких псевдоинструкций.

Список псевдоинструкций ассемблера представлен в таблице 1.

*Таблица 1*

| **Pesudo-instruction** | **Expansion** | **Description** |
| --- | --- | --- |
| nop | addi zero,zero,0 | No operation |
| li rd, expression | (several expansions) | Load immediate |
| la rd, symbol | (several expansions) | Load address |
| mv rd, rs1 | addi rd, rs1, 0 | Copy register |
| not rd, rs1 | xori rd, rs1, -1 | One’s complement |
| neg rd, rs1 | sub rd, x0, rs1 | Two’s complement |
| negw rd, rs1 | subw rd, x0, rs1 | Two’s complement Word |
| sext.w rd, rs1 | addiw rd, rs1, 0 | Sign extend Word |
| seqz rd, rs1 | sltiu rd, rs1, 1 | Set if = zero |
| snez rd, rs1 | sltu rd, x0, rs1 | Set if ≠ zero |
| sltz rd, rs1 | slt rd, rs1, x0 | Set if < zero |
| sgtz rd, rs1 | slt rd, x0, rs1 | Set if > zero |
| fmv.s frd, frs1 | fsgnj.s frd, frs1, frs1 | Single-precision move |
| fabs.s frd, frs1 | fsgnjx.s frd, frs1, frs1 | Single-precision absolute value |
| fneg.s frd, frs1 | fsgnjn.s frd, frs1, frs1 | Single-precision negate |
| fmv.d frd, frs1 | fsgnj.d frd, frs1, frs1 | Double-precision move |
| fabs.d frd, frs1 | fsgnjx.d frd, frs1, frs1 | Double-precision absolute value |
| fneg.d frd, frs1 | fsgnjn.d frd, frs1, frs1 | Double-precision negate |
| beqz rs1, offset | beq rs1, x0, offset | Branch if = zero |
| bnez rs1, offset | bne rs1, x0, offset | Branch if ≠ zero |
| blez rs1, offset | bge x0, rs1, offset | Branch if ≤ zero |
| bgez rs1, offset | bge rs1, x0, offset | Branch if ≥ zero |
| bltz rs1, offset | blt rs1, x0, offset | Branch if < zero |
| bgtz rs1, offset | blt x0, rs1, offset | Branch if > zero |
| bgt rs, rt, offset | blt rt, rs1, offset | Branch if > |
| ble rs, rt, offset | bge rt, rs1, offset | Branch if ≤ |
| bgtu rs, rt, offset | bltu rt, rs1, offset | Branch if >, unsigned |
| bleu rs, rt, offset | bltu rt, rs1, offset | Branch if ≤, unsigned |
| j offset | jal x0, offset | Jump (uncoditional) |
| jr offset | jal x1, offset | Jump register |

Пример.

1) Псевдоинструкция li rd, x5,0x123456 (загрузка большого значения в регистр) преобразуется ассемблером в 2 инструкции:

lui rd, 0x123 (загрузить константу в старшие биты 31-12 регистра)

addi rd, rd, 0x456 (сложение того же регистра с младшими 12 битами).

2) Псевдоинструкция lа rd, symbol (загрузка адреса в регистр) преобразуется ассемблером в 2 инструкции:

auipc rd, sym[31:12](сложение PC +(sym << 12) и запись в биты 31-12 регистра)

addi rd, rd, sym[11:0] (сложение того же регистра с младшими 12 битами sym).

В дополнение к этому списку рассмотрим еще ряд псевдокоманд, пред-ставленных в таблице 2, которые будут полезны для выполнения лабораторных работ по Ассемблеру RISC-V,.

Таблица 2

|  |  |  |
| --- | --- | --- |
| Название | Псевдоинструкция | Расширение |
| Вызов процедуры (функции) типа **near** | Call offset12 | Jalr ra, ra, offset12 |
| Вызов процедуры (функции) типа **far** | Call offset | Auipc ra, offset [31:12]  Jalr ra, ra, offset [11:0] |
| Возврат из процедуры | Ret | Jalr x0, 0(ra) |

Напомним, что инструкция auipc (Add Upper Imm to PC) выполняет следующие действия rd = PC + (imm << 12)

***Использование режимов адресации***

В архитектуре RISC-V используются четыре режима адресации: регистровый, непосредственный, базовый и относительно счетчика команд. Первые три режима (регистровый, непосредственный и базовый) определяют способы чтения и записи операндов. Последний режим (относительно счетчика команд) определяет способ записи счетчика команд.

*Регистровая адресация*

При регистровой адресации регистры используются для всех операндов-источников и операндов-назначений (иными словами – для всех операндов и результата). Все инструкции типа R используют именно такой режим адресации.

add rd,rs1,rs2 # rd = rs1 + rs2

*Непосредственная адресация*

При непосредственной адресации в качестве операндов наряду с регистрами используют константы (непосредственные операнды). Этот режим адресации используют некоторые инструкции типа I, такие как сложение с 12-битной константой (addi) и логическая операция andi.

addi rd,rs1,12 # rd = rs1 + 12

andi rd,rs1,-8 # rd = rs1 & 0xFFF8

Чтобы использовать константы большего размера, следует использовать инструкцию непосредственной записи в старшие разряды lui (load upper immediate), за которой следует инструкция непосредственного сложения addi Инструкция lui загружает 20-битное значение сразу в 20 старших битов и помещает нули в младшие биты:

lui s2, 0xABCDE # s2 = 0xABCDE000

addi s2, s2, 0x123 # s2 = 0xABCDE123

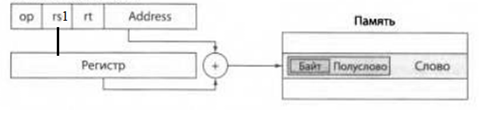
При использовании многоразрядных непосредственных операндов, если указанный в addi 12-битный непосредственный операнд отрицательный, старшая часть постоянного значения в lui должна быть увеличена на единицу. Помните, что знак addi расширяет 12-битное непосредственное значение, поэтому отрицательное непосредственное значение будет содержать все единицы в своих старших 20 битах. Поскольку в дополнительном коде все единицы означают число –1, добавление числа, у которого все разряды установлены в 1, к старшим разрядам непосредственного операнда приводит к вычитанию 1 из этого числа. Пример иллюстрирует ситуацию, когда мы хотим в s2 получить постоянное значение 0xFEEDA987:

lui s2, 0xFEEDB # s2 = 0xFEEDB000 (число, которое нужно записать в старшие 20 разрядов (0xFEEDA), предварительно увеличено на 1)

addi s2, s2, −1657 # s2 = 0xFEEDA987 (0x987 – это 12-битное представление числа -1657)(0xFEEDB000 + 0xFFFFF987 = 0xFEEDA987)

*Базовая адресация*

Инструкции для доступа в память, такие как загрузка слова(чтение памяти) (lw) и сохранение слова(запись в память) (sw), используют базовую адресацию. Эффективный адрес операнда в памяти вычисляется путем сложения базового адреса в регистре rs1 и 12-битного смещения с расширенным знаком, являющегося непосредственным операндом. Операции загрузки (lw) – это инструкции типа I, а операции сохранения (sw) – инструкции типа S.



lw rd, 36(rs1) # rd = M[rs1+imm][0:31]

Поле rs1 указывает на регистр, содержащий базовый адрес, а поле rd указывает на регистр-назначение. Поле imm, хранящее непосредственный операнд, содержит 12-битное смещение, равное 36. В результате регистр rd содержит значение из ячейки памяти rs1+36

sw rs2, 8(rs1) # M[rs1+imm][0:31] = rs2[0:31]

Инструкция сохранения слова sw демонстрирует запись значения из регистра rs2 в слово памяти, расположенное по адресу rs1+8

*Адресация относительно счетчика команд*

Инструкции условного перехода, или ветвления, используют адресацию относительно счетчика команд для определения нового значения счетчика командв том случае, если нужно осуществить переход. Смещение со знаком прибавляется к счетчику команд (PC) для определения нового значения PC, поэтому тот адрес, куда будет осуществлен переход, называют адресом относительно счетчика команд.

Инструкции перехода по условию (beq, bne, blt, bge, bltu, bgeu) типа B и jal (переход и связывание) типа J используют для смещения 13- и 21-битные константы со знаком соответственно. Старшие значимые биты смещения располагаются в 12- и 20-битных полях инструкций типа B и J. Наименьший значащий бит смещения всегда равен 0, поэтому он отсутствует в инструкции.

beq rs1, rs2, imm # if(rs1 == rs2) PC += imm

jal rd, imm #rd = PC+4; PC += imm

Инструкция jal может быть использована как для вызова функций, так и для простого безусловного перехода. В RISC-V используется соглашение, что адрес возврата должен быть сохранён в регистре адреса возврата ra ( x1).

Инструкция jal не имеет достаточного места для кодирования полного 32-битного адреса. Это означает, что вы не можете сделать переход куда-либо в коде, если ваша программа больше максимального значения смещения. Но если адрес перехода хранится в регистре, вы можете сделать переход на любой адрес (инструкция jalr типа I).

jalr rd, imm (rs1) # rd = PC + 4, PC = rs1 + imm

Большая разница состоит в том, что переход JALR не происходит относительно PC. Вместо этого он происходит относительно rs1

Инструкция auipc типа U (сложить старшие разряды константы смещения с PC) также использует адресацию относительно счетчика команд.

auipc rd,imm # rd = PC + (imm << 12)

auipc s3, 0xABCDE # s3 = PC + 0xABCDE000

Примеры программ на ассемблере (или их фрагментов), использующих различные режимы адресации.

**1.Регистровая адресация**

*Задание.*

Сложить 2 константы n1=511 и n2=-10, заданные с помощью директивы .equ. Константы разместить в регистрах t0 и t1, а результат сложения в регистре t2

*Код программы*

.text

start:

.global \_start

.EQU n1, 511

.EQU n2, -10

# Регистровая адресация

addi t0, zero, n1

addi t1, zero, n2

add t2, t0, t1

**2. Непосредственная адресация**

*Задание.*

Переслать константу n1=511 в регистр t0 и выполнить логическую операцию И с константой -8. Результат операции сохранить в регистре t1

*Фрагмент*  *программы*

addi t0, zero, n1

andi t1, t0,-8

**3.Базовая адресация**

*Задание.*

Сложить n=8 первых четных целых чисел sum = a[0] + a[1] + a[2] + ... + a[7], результат записать в память.

Регистровый файл Память

|  |  |
| --- | --- |
| X1 | Addr of a[i] |
| X2 | End of array |
| S1 | Sum |
| X3 | a[i] |
| X10 | 100 |

|  |  |
| --- | --- |
| 0 | a[0] |
| 4 | a[1] |
| … | … |
| 28 | a[7] |
| … | … |
| 104 | sum |

*Код программы*

*.data*

array: .word *0,2,4,6,8,10,12,14*

end\_arr: .word *28*

sum: .word  *0*

*.text*

start:

.global \_start

la x1, array # адрес начала массива

la x2, end\_arr # адрес конца массива

la s1, sum # адрес суммы

add s1, x0, x0

loop:

lw x3, 0x0(x1) # a[i]

add s1, s1, x3 # накопление суммы

addi x1, x1, 4 # след элем. массива

bne x1, x2, loop # не конец массива

sw s1, 0x4(x10) # сохранение суммы

exit:

# Exit program

li a0, 0

li a7, 93

ecall

4. Адресация относительно счетчика команд

*Задание.* Найти c = max(a, b)

Регистровый файл до выполнения программы

|  |  |
| --- | --- |
| a0 | a |
| a1 | b |

Регистровый файл после выполнения программы

|  |  |
| --- | --- |
| a0 | c |

*Фрагмент программы.*

max:

blt a0, a1, second # if a0 < a1 then a1 is larger

jal zero, done # max = a0

second:

add a0, zero, a1 # make a1 the return value

done:

***Лабораторные работы по Ассемблеру RISC-V.***

*(Краткое пояснение целей и содержания работ)*

**Лабораторная работа 1.**

*Знакомство с рабочей средой эмулятора Ripes для работы с процессором RISC-V. Базовый ISA, система команд, состав регистров. Разработка и выполнение простой программы на ассемблере RISC-V.*

* 1. Цели работы.

1. Освоение работы с эмулятором Ripes: установка, настройка, трансляция ассемблерной программы, выполнение программы в автоматическом и отладочном режимах.

2. Изучение архитектуры RISC-V, базового набора инструкций и разработка простых программ на ассемблере.

1.2. Основные теоретические сведения

1. Инструкция по работе с эмулятором Ripes.

2. Описание состава используемых регистров и базового набора команд процессора RISC-V.

3. Краткие сведения по ассемблеру RISC-V.

1.3. Задание к лабораторной работе

1. Разработайте процедуру на ассемблере, которая для целочисленных 32-битных входных переменных x, y, z и констант a, b, c вычисляет выражение

R = f (x, y, z, a, b, c)

выбираемое в соответствии с вашим номером в списке группы. Например:

R = (x >> b) & (y + c) + (z | c)

В выражении используются следующие константы:

|  |  |
| --- | --- |
| Константа | Значение |
| a | [Сумма цифр студ. билета] |
| b | [Количество букв в фамилии] |
| c | [Количество букв в полном имени] |

2. Напишите программу, которая для двух наборов исходных данных x, y, z выполняет вычисление заданного выражения с помощью разработанной процедуры, сохраняет в регистрах и выводит на экран результаты вычислений.

Начальные значения {x1, y1, z1} расположить в регистрах a2, a3, a4; значения {x2, y2, z2} расположить в регистрах a5, a6, a7; значения констант a, b, c расположить в регистрах s0,s1,s2 . Результаты вычисления {r1, r2} записать в регистры а0, а1.

**Лабораторная работа 2.**

*Изучение режимов адресации в ассемблере RISC-V.*

2.1. Цель работы.

1. Разработка простой программы преобразования данных для приобретения практических навыков программирования на языке ассемблера.

2. Закрепление знаний по режимам адресации в процессоре *RISC-V*.

2.2. Основные теоретические сведения

1. Описание состава используемых регистров и базового набора команд и набора псевдокоманд процессора RISC-V.

2. Краткие сведения по режимам адресации в ассемблере RISC-V.

2.3. Задание к лабораторной работе

1. Для заданного набора констант

|  |  |
| --- | --- |
| Константа | Значение |
| a | [Сумма цифр студ. билета] |
| b | [Количество букв в фамилии] |
| c | [Количество букв в полном имени] |

сформировать массив array из 10 элементов, в котором

arr[0] = a+b+c ; array[i+1] = arr[i] + a + b - c

Доступ к массиву (инициализация, чтение) должен выполняться из памяти.

Значения всех регистров по умолчанию приравнять к нулю. Константы a,b,c в регистрах не сохраняются.

2. Написать программу, которая с использованием 4 режимов адресации: регистрового, непосредственного, базового и относительного к счетчику команд реализует вычисление выражения, выбираемое в соответствии с номером студента в списке группы

|  |  |  |
| --- | --- | --- |
| Номер | Операции | Используемые регистры |
| 1 | ЕСЛИ (arr[4] + arr[6] + arr[0] < threshold) ТО (res1 = arr[3] & arr[5]) ИНАЧЕ (res2 = arr[2] | c) | threshold -> t0  res1 -> s5  res2 -> a1 |

Здесь threshold – заданный порог.