Введение в архитектуру RISC-V
Язык ассемблера

Никита Поляков

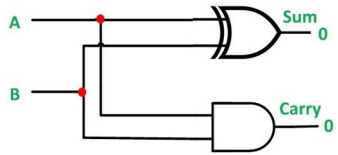
ООО “Синтакор”, старший инженер по разработке аппаратных средств

2021

Уровни представления вычислений

- Процесс вычисления или выполнения программы может быть представлены на нескольких ***уровнях абстракции***
- Каждому уровню абстракции соответствует средство проектирования

	Уровень абстракции	Средство	
программа	Алгоритм	**Языки высокого уровня (C/C++)**	*a = b + c;*
	Архитектура набора команд	**Язык ассемблера, который понятен человеку**	*add x4, x2, x3*
	Программа на машинном коде	**Двоичный код, который “понятен” аппаратуре**	*0x00310233*
аппаратура	Микроархитектура	**Блок-схемы и языки описания аппаратуры (Verilog, VHDL)**	*assign x4 = x2 + x3;*
	Физическая реализация	**Электрические схемы**	



Команды процессора

- Работа процессора (Central Processing Unit, CPU) - **выполнять команды**
- Команды процессора - **примитивные операции**, которые он может выполнять:
  - команды выполняются одна за другой последовательно
  - каждая команда выполняет какую-то небольшую часть работы
  - команда выполняет операцию над операндами
  - некоторые команды могут менять последовательность выполнения команд
- последовательность команд, хранящаяся в памяти - **программа**

Архитектура набора команд

- Процессоры делятся на “семьи”, в каждой из которых свой набор команд
- Каждый набор конкретного процессора реализует архитектуру набора команд (*Instruction Set Architecture, ISA*). Примеры ISA:
  - ARM, Intel x86, MIPS, RISC-V, IBM Power и т.д.
- ISA определяет команды с точностью до двоичной кодировки, поэтому процессоры из одной *семьи* могут выполнять *одни и те же программы*
- **Язык ассемблера** (или просто **ассемблер**, англ. **Assembly Language**) - язык программирования, прямо соответствующий ISA, но понятный человеку

RISC-V

- **Пятое** поколение архитектур набора команд **RISC**, созданое в 2010 году исследователями из калифорнийского университета в Беркли
- Спецификация ISA доступна для **свободного и бесплатного** использования - Linux в мире архитектур
- Предназначена для использования как в **коммерческих**, так и **академических** целях
- Поддерживается общая растущая **программная экосистема**
- Архитектура имеет **стандартную** версию, а также несколько **расширений** системы команд
- Подходит для вычислительных систем всех уровней: от **микроконтроллеров** до **суперкомпьютеров**
- Стандарт поддерживается некоммерческой организацией **“RISC-V Foundation”**, которая работает в тесном партнерстве с “The Linux Foundation”

RISC-V Foundation

Переменные в ассемблере - регистры

- В отличие от языков высокого уровня в ассемблере **отсутствуют переменные**
- Вместо переменных команды оперируют с **регистрами**
  - ограниченный набор **ячеек хранения чисел**, встроенных прямо в аппаратуру
  - в архитектурах RISC арифметические операции могут выполняться только с регистрами
  - с памятью возможны только операции записи и считывания (в отличие от CISC)
- Преимущество работы с регистрами - скорость доступа к ним
- Недостатки - ограниченной число регистров - 32 в RISC-V

Регистры RISC-V

- 32 регистра для основного набора команд
- каждый регистр имеет размер 32 бита = слово (word)
- x0 всегда равен 0
- 32 регистра для вещественных операций в расширении “F”
- в версия RV64 регистры имеют размер 64 бита (double word)

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

Команды RISC-V

- Каждая команда имеет код операции (opcode) и операнды

```
add  x1,  x2,  x3        # x1 = x2 + x3
```

add - код операции - сложение
x1 - регистр результата
x2, x3 - регистры-операнды
- используется в ассемблере для комментариев

- Эквивалент в языке C:

a = b + c

x1 ⇔ a, x2 ⇔ b, x3 ⇔ c

Арифметические команды

- **add** **rd, rs1, rs2**
  - сложение rd = rs1 + rs2
- **sub** **rd, rs1, rs2**
  - вычитание rd = rs1 - rs2
- **and** **rd, rs1, rs2**
  - побитовое И rd = rs1 & rs2
- **or** **rd, rs1, rs2**
  - побитовое ИЛИ rd = rs1 | rs2
- **xor** **rd, rs1, rs2**
  - побитовое исключающее ИЛИ rd = rs1 xor rs2
- **sll** **rd, rs1, rs2**
  - сдвиг влево rd = rs1 << rs2

Обращение к памяти

- 1 байт = 8 бит
- 4 байта = 1 слово (word)
- адреса в памяти - адреса в байтах
- в RISC-V байты в словах расположены в соответствии с little endian, т.е. байты с меньшим адресов расположены в младших битах (см. картинку)

Команды обращения к памяти

- **lw** **rd, addr**
  - считывание слова в регистр rd из памяти по адресу addr
- **sw** **rd, addr**
  - запись слова в из регистра rd в память по адресу addr
- также доступны обращения меньшими размерами:
  - halfword - 2 байта: **lh**, **sh**
  - byte - 1 байт: **lb**, **sb**
- адрес addr может быть указан несколькими способами, самый простой

addr = **offset**(**r1**)

где **r1** - регистр, содержащий адрес обращения

offset - дополнительное смещение, указанное в виде числа

```
lw x2, 4(x3)          # считывание слова из адреса = x3 + 4
```

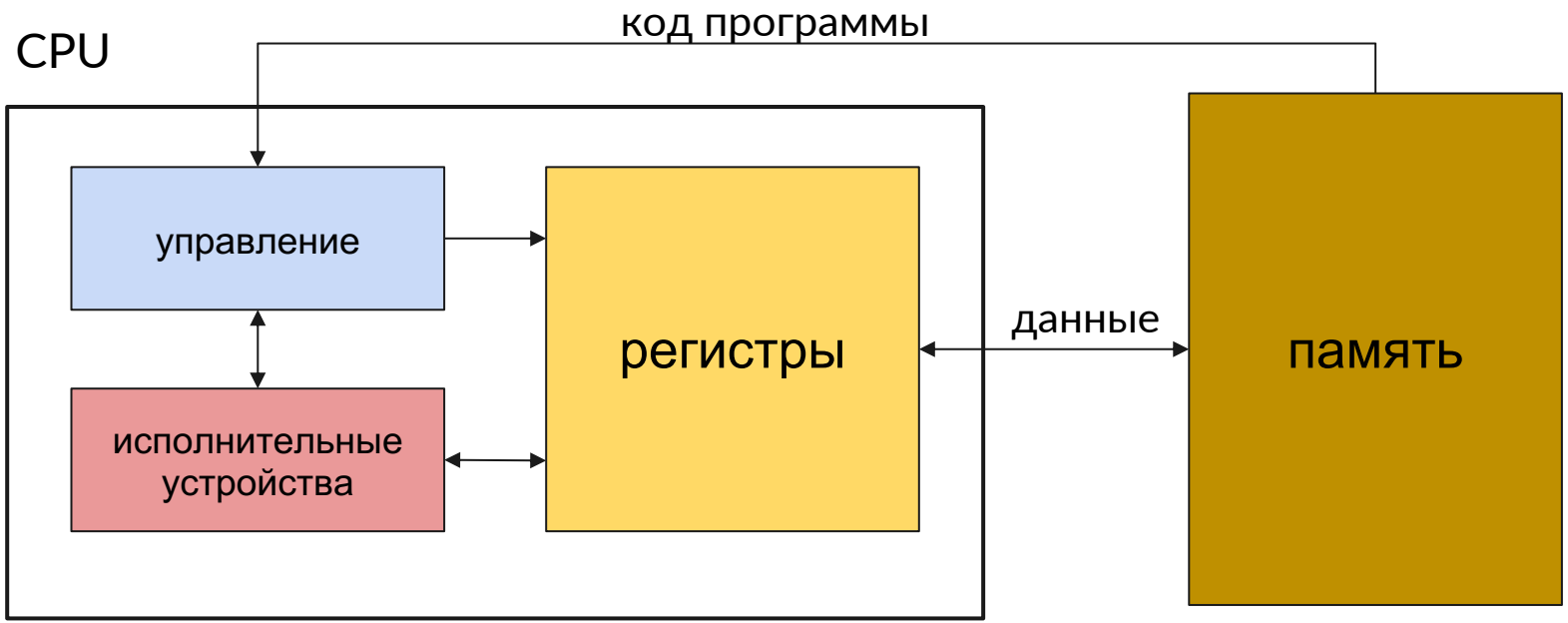
Числовые константы

- в коде ассемблера в качестве операндов можно использовать числа или непосредственные значения (immediate operand) или константы
- для использования констант в архитектуре предусмотрены специальные команды:
- **addi rd, rs1, imm**
  - сложение rd = rs1 + imm, например
  - **addi x2, x3, -4** # x2 = x3 - 4
- **li rd, imm**
  - запись константы в регистр rd = rs1
- числа по умолчанию указываются в десятичной системе и со знаком
- для использований шестнадцатиричных чисел нужно добавить “0x”, например, **0x10 = 16**

Ветвления в программе. Переходы

- ветвления подразумевают, что в зависимости от результатов некоторых вычислений нужно выполнять **разные** действия
- в языках программирования используется оператор **if**
- аналог оператора if в ассемблере - операции **условного перехода** (***branch***)
- **beq** **rs1, rs2, label** # **b**ranch if **eq**ual
  - если rs1 == rs2, сделать переход на участок кода, помеченный label, иначе выполнить следующую команду
- **bne** **rs1, rs2, label** # **b**ranch if **n**ot **e**qual
  - если rs1 != rs2, сделать переход на участок кода, помеченный label, иначе выполнить следующую команду
- также есть **безусловные** переходы **jump**
- **j** **label**

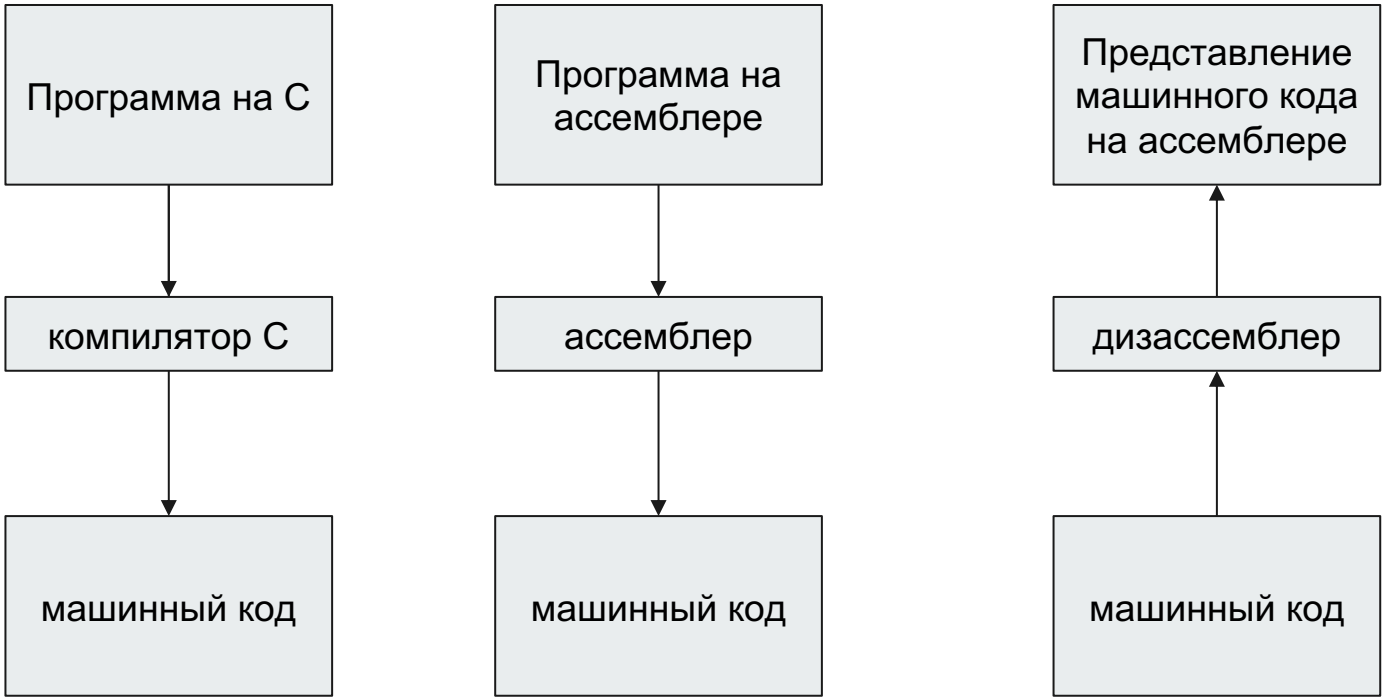
Архитектура с точки зрения программиста



Программирование на языке ассемблера

- Зачем программировать на ассемблере, если есть языки высокого уровня?
  - ассемблер до сих пор используется в системном ПО (например, ОС), чтобы получить доступ к специальным аппаратным ресурсам
  - ассемблер используется при разработке аппаратуры:
    - для написания тестовых программ
    - для изучения особенностей работы аппаратуры при выполнении программ используют дизассемблирование, т.е. получение из двоичного кода ассемблерной программы

Программирование на языке ассемблера



Псевдоинструкции

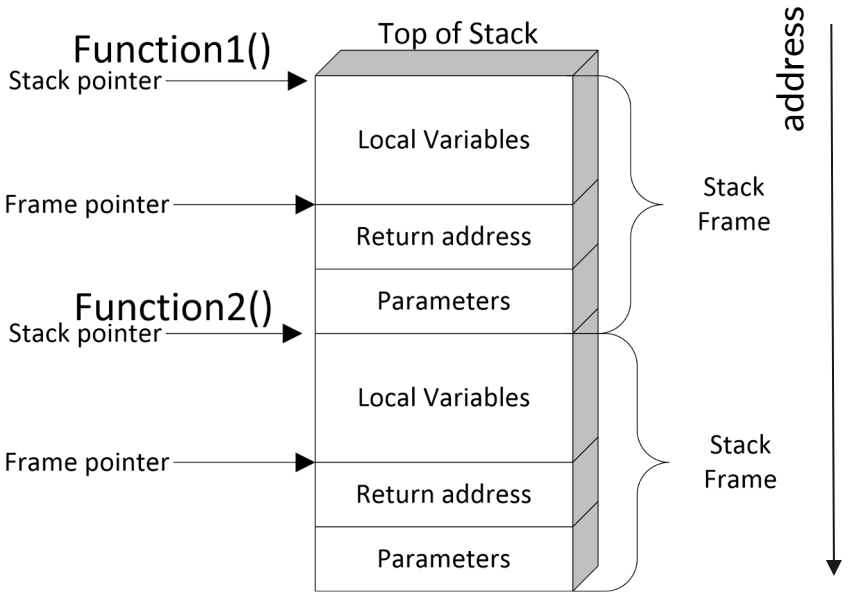
- Команды ассемблера, которые упрощают читаемость, но при сборке в двоичный код заменяются на другие
- **nop** ⇔ **addi x0, x0, 0** # no operation
- **mv rd, rs** ⇔ **addi rd, rs, 0** # copy register
- **beqz rs, offset ⇔ beq rs, x0, offset** # branch if = zero

Стэк вызова

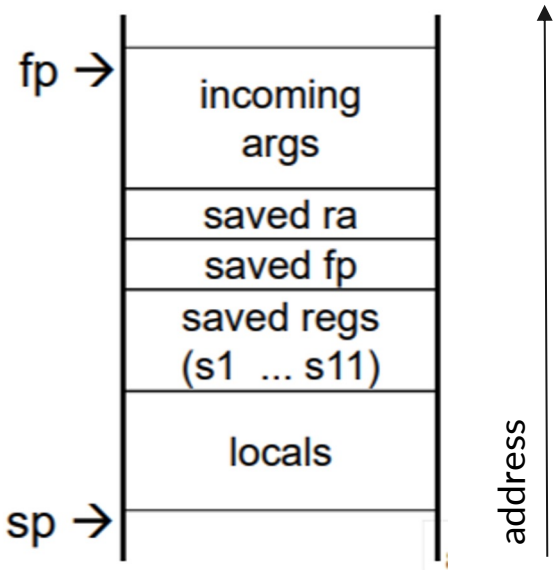
Стек вызовов (call stack) — структура данных, хранящая информацию для возврата управления из подпрограмм (процедур, функций) в программу (или подпрограмму, при вложенных или рекурсивных вызовах) и/или для возврата в программу из обработчика прерывания
Стек обычно хранится в памяти и в нем сохраняются:

- аргументы вызванной функции (если не помещаются в регистрах)
- адрес возврата или другие указатели
- локальные переменные самой функции (если не помещаются в регистрах)
- значения регистров вызывающей функции

```
int function2(int a, ...)
{
        function1(b,c, ...);
}
```



Регистры стэка вызова



Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

Команды JAL, JALR, RET

- Команды JAL/JALR выполняют безусловный переход с сохранением текущего PC для возможности возврата в тот же участок программы
- **jal rd, label** # **j**ump **a**nd **l**ink
  - сделать переход на участок кода, помеченный ***label*** (immediate address), и записать ***(pc+4)*** в регистр ***rd***
- **jalr rd, offset(rs1)** # **j**ump **a**nd **l**ink **r**egister
  - сделать переход на участок кода по адресу ***(rs1 + offset)***, и записать ***(pc+4)*** в регистр ***rd***
- **ret** # **r**eturn from subroutine: **jalr** x0, x1, 0