Архитектура RISC-V. Система команд. Лекция №2.

Levels of Representation/Interpretation


```
lw    $t0, 0($2)
lw    $t1, 4($2)
sw    $t1, 0($2)
sw    $t0, 4($2)
```

High Level Language Program (e.g., C)

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

Compiler

Assembly Language Program (e.g., MIPS)

Anything can be represented as a ***number***, i.e., data or instructions

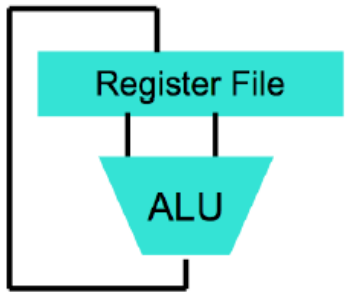
Assembler

Machine Language Program (MIPS)

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

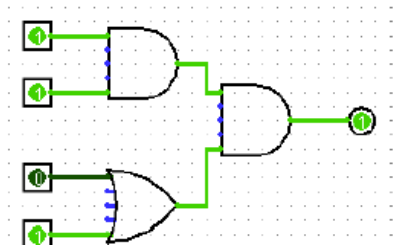
Machine Interpretation

Hardware Architecture Description (e.g., block diagrams)



Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)



Instruction Set Architecture (ISA)

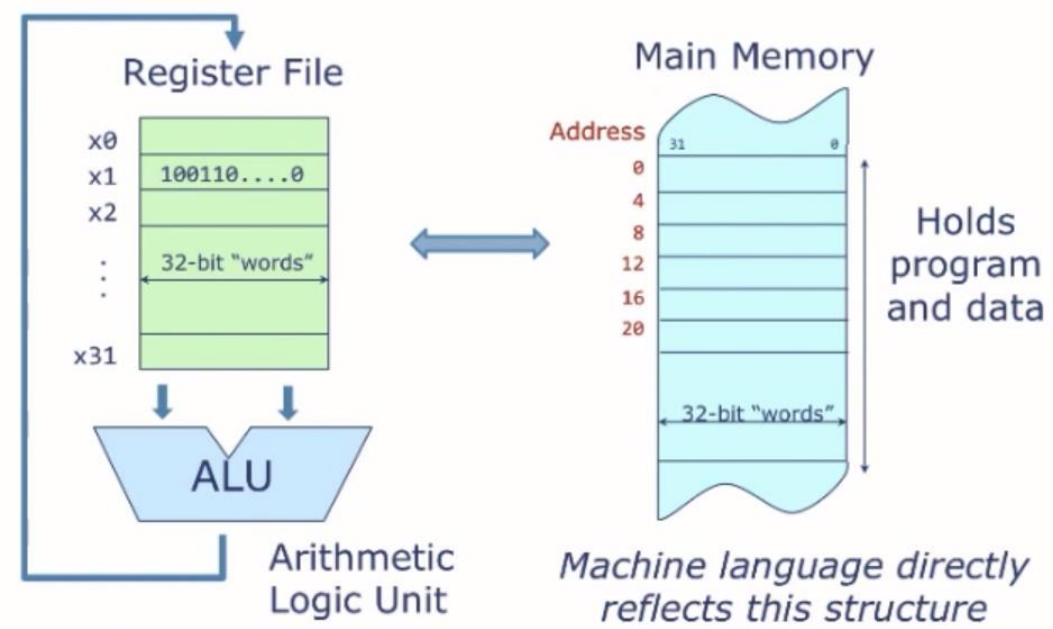
- Система команд
- Средства для выполнения команд
  - Форматы данных
  - Системы регистров
  - Способы адресации
  - Модели памяти

Instruction Set Architecture (ISA)

RISC-V

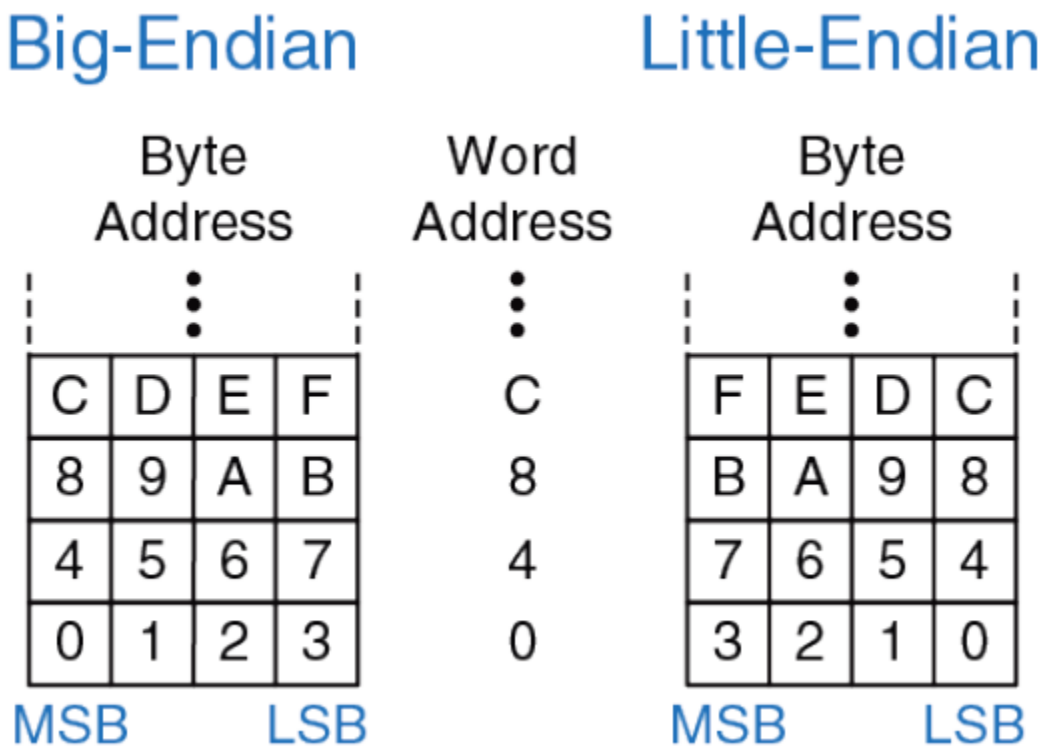
- Пятое поколение ISA RISC, созданное в 2010 году исследователями из Беркли
- Спецификация ISA доступна для свободного и бесплатного использования
- Предназначена для использования в коммерческих и академических целях
- Поддерживается общая растущая программная экосистема
- Архитектура имеет стандартную версию, а также несколько расширений системы команд
- Подходит для вычислительных систем всех уровней: от микроконтроллеров до суперкомпьютеров

Модель процессора RISC-V



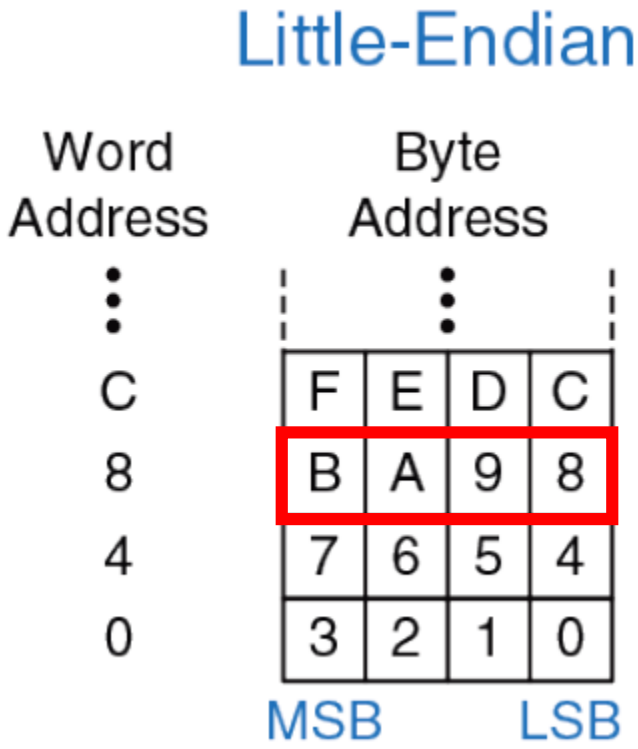
- Регистровый файл
  - 32 регистра общего назначения
  - Каждый регистр 32 бита
  - x0 = 0
- Память
  - Каждая ячейка памяти имеет ширину 32 бита (1 слово)
  - Память имеет побайтовую адресацию
  - Адреса соседних слов отличаются на 4
  - Адрес 32 бита
  - Может быть адресовано 2^32 байт или 2^30 слов

Побайтовая адресация памяти



Выровненный доступ к памяти

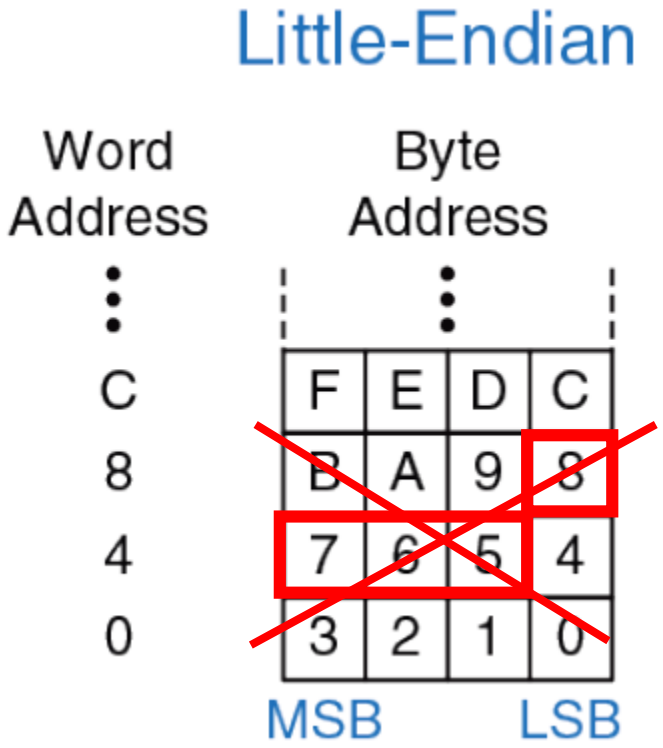
```
Address = 0x8
Read Word
```

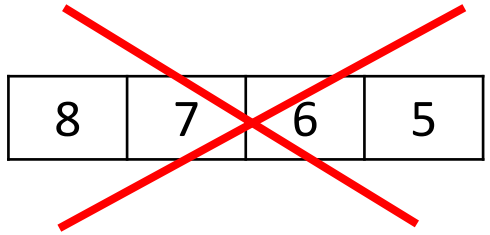


B	A	9	8

Выровненный доступ к памяти

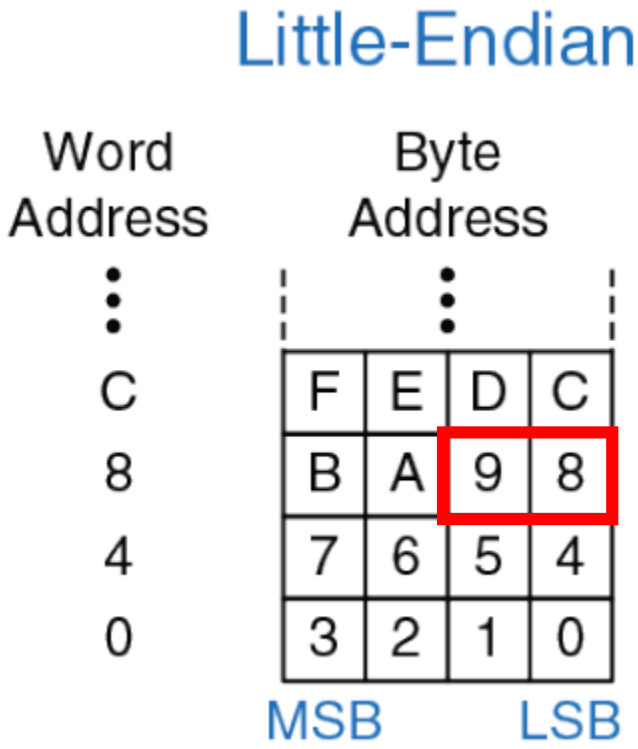
```
Address = 0x5
Read Word
```





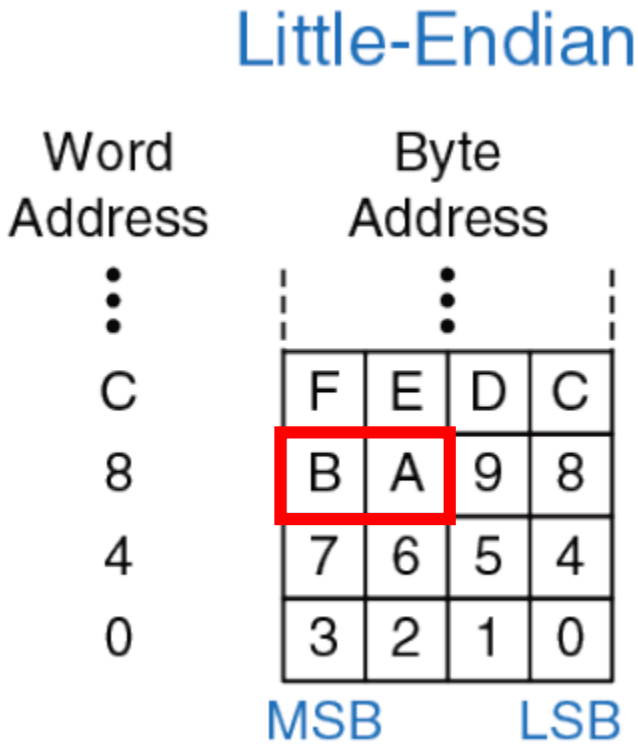
Выровненный доступ к памяти

```
Address = 0x8
Read Half
```



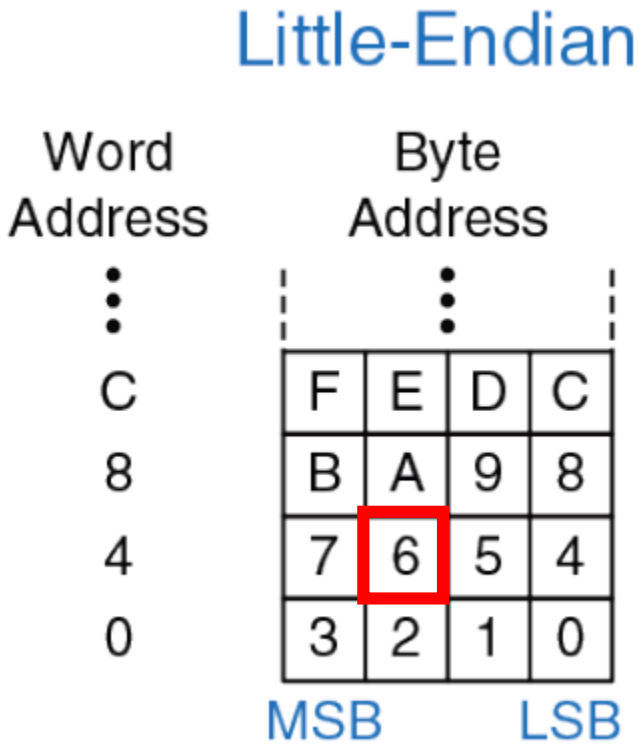
Выровненный доступ к памяти

```
Address = 0xA
Read Half
```



Выровненный доступ к памяти

```
Address = 0x6
Read Byte
```



Структура процессора/Язык ассемблера

- Каждый регистр фиксированного размера 32 бита
- Количество регистров – 32
- АЛУ выполняет операции между операндами, хранящимися в регистровом файле
  - $x_i \leftarrow op(x_j, x_k)$, где $op \in \{+, AND, OR, <, >, \dots\}$
- Основная память может хранить гигабайты и хранит программы и данные
- Данные можно перемещать между основной памятью и регистровым файлом
  - `ld x M[addr]`
  - `st M[addr] x`

ЯВУ vs язык ассемблера

Язык высокого уровня

- Примитивные арифметические и логические операции
- Сложные типы и структуры данных
- Сложная структура управления – условные операторы, циклы, процедуры
- Не подходит для прямой реализации в аппаратном обеспечении

Язык ассемблера

- Примитивные арифметические и логические операции
- Примитивные структуры данных – биты и числа
- Инструкции передачи управления
- Разработан для непосредственного размещения в аппаратуру (утомительное программирование)

RISC-V ISA: инструкции

- Три типа операций
  - **Вычислительные** – выполняют арифметические и логические операции над операндами в регистрах
  - **Загрузки и сохранения** – перемещают данные между основной памятью и регистрами
  - **Управления** выполнением программой – изменяют ход выполнения программы (условные переходы, циклы)

Вычислительные инструкции

- Арифметические, сравнения, логические и операции сдвига
  - Инструкции register-register
    - 2 источника регистров операндов
    - 1 регистр назначения

Арифметические	Сравнения	Логические	Сдвига
add, sub	slt, sltu	and, or, xor	sll, srl, sra

- Формат записи: `опер назнач, ист1, ист2`

```
• add x3, x1, x2        # x3 ← x1 + x2
• slt x3, x1, x2        # if x1 < x2 then x3 = 1 else x3 = 0
• and x3, x1, x2        # x3 ← x1 & x2
• sll x3, x1, x2        # x3 ← x1 << x2
```

Инструкции register–immediate

- Один операнд находится в регистре, второй – маленькая константа, закодированная в инструкции
  - Формат записи: `опер назнач, ист1, константа`
  - `addi x3, x1, 3` `# x3 ← x1 + 3`
  - `andi x3, x1, 3` `# x3 ← x1 & 3`
  - `slli x3, x1, 3` `# x3 ← x1 << 3`

Формат	Арифметические	Сравнения	Логические	Сдвига
register–register	add, sub	slt, sltu	and, or, xor	sll, srl, sra
register–immediate	addi	slti, sltiu	andi, ori, xori	slli, srli, srai

Инструкции управления программой

- Инструкции условного перехода
  - Формат записи: `comp src1, src2, label`
  - Сначала выполняется сравнение, чтобы определить произойдет ли переход
    `src1 comp src2`
  - Если результат сравнения True, то переход происходит, в противном случае исполняется следующая в памяти инструкция

Инструкция	beq	bne	blt	bge	bltu	bgeu
Сравнение	==	!=	<	≥	<	≥

```
a = 3
b = 8
```

```
if (a < b)
        c = a + 1;
else
        c = b + 2;
```

```
    bge x1, x2, else
    addi x3, x1, 1
    beq x0, x0, end
else: addi x3, x2, 2
end:
```

```
x1 = a;
x2 = b;
c3 = c;
```

Инструкции безусловного перехода

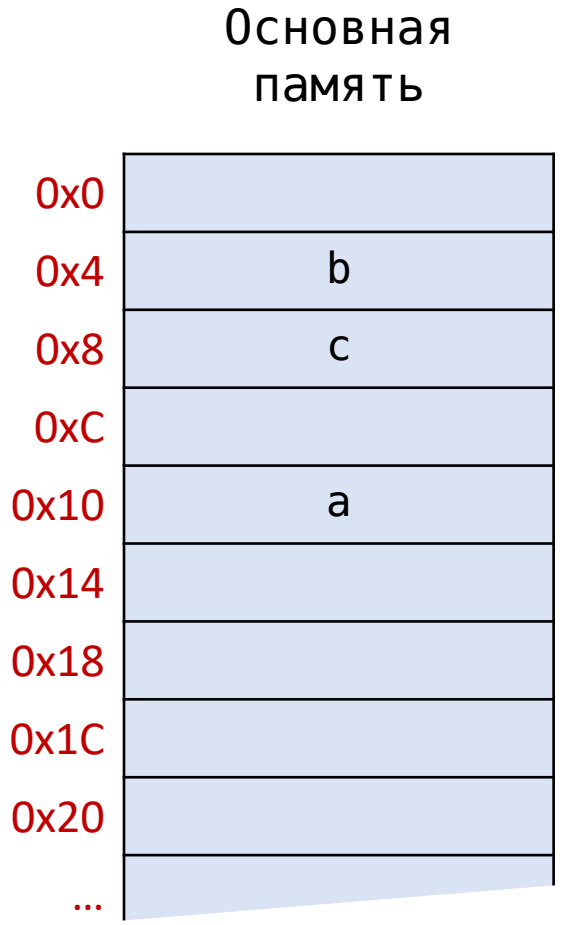
- `jal`: Безусловный переход с сохранением адреса возврата
  - Пример: `jal x3, label`
  - Место перехода определяется меткой `label`
  - `label` является смещением относительно текущей инструкции
  - Адрес возврата сохраняется в `x3`
- `jalr`: Безусловный переход по значению из регистра с сохранением адреса возврата
  - Пример: `jalr x3, 4(x1)`
  - Место перехода определяется значением из регистра плюс смещение
  - Пример: `Адрес перехода = x1 + 4`
  - Можно перейти по любому 32-битному адресу

Вычисления значений из памяти

```
a = b + c

x1 ← load(Mem[b])
x2 ← load(Mem[c])
x3 ← x1 + x2
store(Mem[a]) ← x3

x1 ← load(0x4)
x2 ← load(0x8)
x3 ← x1 + x2
store(0x10) ← x3
```



Инструкции загрузки и сохранения (load and store)

- Адрес указывается как пара <Базовый адрес, смещение>
  - Базовый адрес всегда располагается в регистре
  - Смещение задается маленькой константой
  - **Формат записи:**

```
lw dest, offset(base)
sw src, offset(base)
```

Язык ассемблера

```
lw x1, 0x4(x0)
lw x2, 0x8(x0)
add x3, x1, x2
sw x3, 0x10(x0)
```

Поведение

```
x1 ← load(Mem[x0 + 0x4])
x2 ← load(Mem[x0 + 0x8])
x3 ← x1 + x2
store(Mem[x0 + 0x10]) ← x3
```

Псевдоинструкции

- Псевдонимы инструкций для упрощенного программирования на ассемблере

Псевдоинструкции	Инструкции ассемблера
`mv x2, x1`	`addi x2, x1, 0`
`li x2, 3`	`addi x2, x0, 3`
`ble x1, x2, label`	`bge x2, x1, label`
`beqz x1, label`	`beq x1, x0, label`
`bnez x1, label`	`bne x1, x0, label`
`j label`	`jal x0, label`

Регистры и память

```
addi x1, x2, x3
```
x1 = 0x1C

```
mv x4, x3
```
x4 = 0x14

```
lw x5, 0(x3)
```
x5 = 0x23

```
lw x6, 8(x3)
```
x6 = 0x16

```
sw x6, 0xC(x3)
```
Mem[x3+0xC] = 0x16

Работа с константами

- Пример `a = b + 3`
  - Маленькая константа (12 бит) может быть передана через инструкцию
  - `addi x1, x2, 3  # x1 = a, x2 = b`

- Пример `a = b + 0x123456`
  - Максимальный размер 12-битного числа в дополнительном коде это `2^11-1 = 2047 (0x7FF)`
  - Можно использовать псевдоинструкцию `li` для загрузки больших констант

```
li x4, 0x123456
```

```
lui x4, 0x123
addi x4, x4, 0x456
```

`x4 = 0x123000`

Компиляция простых выражений

Пример на C

```c
int x, y, z;
…
y = (x + 3) | (y + 123456);
z = (x * 4) ^ y;
```

RISC-V ассемблер

```
// x: x10, y: x11, z: x12
// x13, x14 – временное хранилище
```

Компиляция простых выражений

Пример на C

```c
int x, y, z;
…
y = (x + 3) | (y + 123456);
z = (x * 4) ^ y;
```

RISC-V ассемблер

```
// x: x10, y: x11, z: x12
// x13, x14 – временное хранилище
addi x13, x10, 3
```

Компиляция простых выражений

Пример на C

```c
int x, y, z;
…
y = (x + 3) | (y + 123456);
z = (x * 4) ^ y;
```

RISC-V ассемблер

```
// x: x10, y: x11, z: x12
// x13, x14 – временное хранилище
addi x13, x10, 3
li x14, 123456
add x14, x11, x14
```

Компиляция простых выражений

Пример на C

```
int x, y, z;
…
y = (x + 3) | (y + 123456);
z = (x * 4) ^ y;
```

RISC-V ассемблер

```
// x: x10, y: x11, z: x12
// x13, x14 – временное хранилище
addi x13, x10, 3
li x14, 123456
add x14, x11, x14
or x11, x13, x14
```

Компиляция простых выражений

Пример на C

```c
int x, y, z;
…
y = (x + 3) | (y + 123456);
z = (x * 4) ^ y;
```

RISC-V ассемблер

```
// x: x10, y: x11, z: x12
// x13, x14 – временное хранилище
addi x13, x10, 3
li x14, 123456
add x14, x11, x14
or x11, x13, x14
slli x13, x10, 2
```

Компиляция простых выражений

Пример на C

```c
int x, y, z;
…
y = (x + 3) | (y + 123456);
z = (x * 4) ^ y;
```

RISC-V ассемблер

```
// x: x10, y: x11, z: x12
// x13, x14 – временное хранилище
addi x13, x10, 3
li x14, 123456
add x14, x11, x14
or x11, x13, x14
slli x13, x10, 2
xor x12, x13, x11
```

RISC-V инструкции

- Вычислительные
  - Register-register `op dest, src1, src2`
  - Register-immeliate `op dest, src1, const`
- Загрузки и сохранения
  - `lw dest, offset(base)`
  - `sw src, offset(base)`
- Управления
  - Безусловный переход `jal label` и `lalr register`
  - Условный переход `comp src1, src2, label`
- Псевдоинструкции

Instr	Название	Функция	Описание	Формат	Opcode	Func3	Func7	Пример использования
add	ADDition	Сложение	rd = rs1 + rs2			0x0	0x00	
sub	SUBtraction	Вычитание	rd = rs1 - rs2			0x0	0x20	
xor	eXclusive OR	Исключающее ИЛИ	rd = rs1 ^ rs2			0x4	0x00	
or	OR	Логическое ИЛИ	rd = rs1 | rs2			0x6	0x00	op rd, rs1, rs2
and	AND	Логическое И	rd = rs1 & rs2			0x7	0x00	
sll	Shift Left Logical	Логический сдвиг влево	rd = rs1 << rs2	R	0110011	0x1	0x00	xor x2, x5, x6
srl	Shift Right Logical	Логический сдвиг вправо	rd = rs1 >> rs2			0x5	0x00	sll x7, x11, x12
sra	Shift Right Arithmetic	Арифметический сдвиг вправо	rd = rs1 >>> rs2			0x5	0x20	
slt	Set Less Then	Результат сравнения A < B	rd = (rs1 < rs2) ? 1 : 0			0x2	0x00	
sltu	Set Less Then Unsigned	Беззнаковое сравнение A < B	rd = (rs1 < rs2) ? 1 : 0			0x3	0x00	
addi	ADDition Immediate	Сложение с константой	rd = rs1 + imm			0x0		
xori	eXclusive OR Immediate	Исключающее ИЛИ с константой	rd = rs1 ^ imm			0x4	-	
ori	OR Immediate	Логическое ИЛИ с константой	rd = rs1 | imm			0x6		
andi	AND Immediate	Логическое И с константой	rd = rs1 & imm			0x7		op rd, rs1, imm
slli	Shift Left Logical Immediate	Логический сдвиг влево	rd = rs1 << imm	I	0010011	0x1	0x00	
srli	Shift Right Logical Immediate	Логический сдвиг вправо	rd = rs1 >> imm			0x5	0x00	addi x6, x3, -12 ori x3, x1, 0x8F
srai	Shift Right Arithmetic Immediate	Арифметический сдвиг вправо	rd = rs1 >>> imm			0x5	0x20	
slti	Set Less Then Immediate	Результат сравнения A < B	rd = (rs1 < imm) ? 1 : 0			0x2	-	
sltiu	Set Less Then Immediate Unsigned	Беззнаковое сравнение A < B	rd = (rs1 < imm) ? 1 : 0			0x3		
lb	Load Byte	Загрузить байт из памяти	rd = SE(Mem[rs1 + imm][7:0])			0x0		
lh	Load Half	Загрузить полуслово из памяти	rd = SE(Mem[rs1 + imm][15:0])			0x1		op rd, imm(rs1)
lw	Load Word	Загрузить слово из памяти	rd = SE(Mem[rs1 + imm][31:0])	I	0000011	0x2	-	
lbu	Load Byte Unsigned	Загрузить беззнаковый байт из памяти	rd = Mem[rs1 + imm][7:0]			0x4		lh x1, 8(x5)
lbh	Load Half Unsigned	Загрузить беззнаковое полуслово из памяти	rd = Mem[rs1 + imm][15:0]			0x5		
sb	Store Byte	Сохранить байт в память	Mem[rs1 + imm][7:0] = rs2[7:0]			0x0		op rs2, imm(rs1)
sh	Store Half	Сохранить полуслово в память	Mem[rs1 + imm][15:0] = rs2[15:0]	S	0100011	0x1	-	
sw	Store Word	Сохранить слово в память	Mem[rs1 + imm][31:0] = rs2[31:0]			0x2		sw x1, 0xCF(x12)
beq	Branch if Equal	Перейти, если A == B	if (rs1 == rs2) PC += imm			0x0		
bne	Branch if Not Equal	Перейти, если A != B	if (rs1 != rs2) PC += imm			0x1		comp rs1, rs2, imm
blt	Branch if Less Then	Перейти, если A < B	if (rs1 < rs2) PC += imm			0x4		
bge	Branch if Greater or Equal	Перейти, если A >= B	if (rs1 >= rs2) PC += imm	B	1100011	0x5	-	beq x8, x9, offset
bltu	Branch if Less Then Unsigned	Перейти, если A < B беззнаковое	if (rs1 < rs2) PC += imm			0x6		bltu x20, x21, 0xFC
bgeu	Branch if Greater or Equal Unsigned	Перейти, если A >= B беззнаковое	if (rs1 >= rs2) PC += imm			0x7		
jal	Jamp And Link	Переход с сохранением адреса возврата	rd = PC + 4; PC += imm	J	1101111	-	-	jal x1, offset
jalr	Jamp And Link Register	Переход по регистру с сохранением адреса возврата	rd = PC + 4; PC = rs1	I	1100111	0x0		jalr x1, 0(x5)
lui	Load Upper Immediate	Загрузить константу в сдвинутую на 12	rd = imm << 12	U	0110111	-	-	lui x3, 0xFFFFF
auipc	Add Upper Immediate to PC	Сохранить счетчик команд в сумме с константой << 12	rd = PC + (imm << 12)		0010111			auipc x2, 0x000FF
ecall	Environment CALL	Передача управления операционной системе	Воспринимать как **nop**	I	1110011	-	-	-
ebreak	Environment BREAK	Передача управления отладчику						

RISC-V расширения

Base	*Version*	*Draft Frozen?*
RV32I	2.0	Y
RV32E	1.9	N
RV64I	2.0	Y
RV128I	1.7	N
Extension	Version	Frozen?
M	2.0	Y
A	2.0	Y
F	2.0	Y
D	2.0	Y
Q	2.0	Y
L	0.0	N
C	2.0	Y
B	0.0	N
J	0.0	N
T	0.0	N
P	0.1	N
V	0.7	N
N	1.1	N

RISC-V расширения

RV32M Multiply Extension

Inst	Name	FMT	Opcode	F3	F7	Description (C)
mul	MUL	R	0110011	0x0	0x01	rd = (rs1 * rs2)[31:0]
mulh	MUL High	R	0110011	0x1	0x01	rd = (rs1 * rs2)[63:32]
mulsu	MUL High (S) (U)	R	0110011	0x2	0x01	rd = (rs1 * rs2)[63:32]
mulu	MUL High (U)	R	0110011	0x3	0x01	rd = (rs1 * rs2)[63:32]
div	DIV	R	0110011	0x4	0x01	rd = rs1 / rs2
divu	DIV (U)	R	0110011	0x5	0x01	rd = rs1 / rs2
rem	Remainder	R	0110011	0x6	0x01	rd = rs1 % rs2
remu	Remainder (U)	R	0110011	0x7	0x01	rd = rs1 % rs2

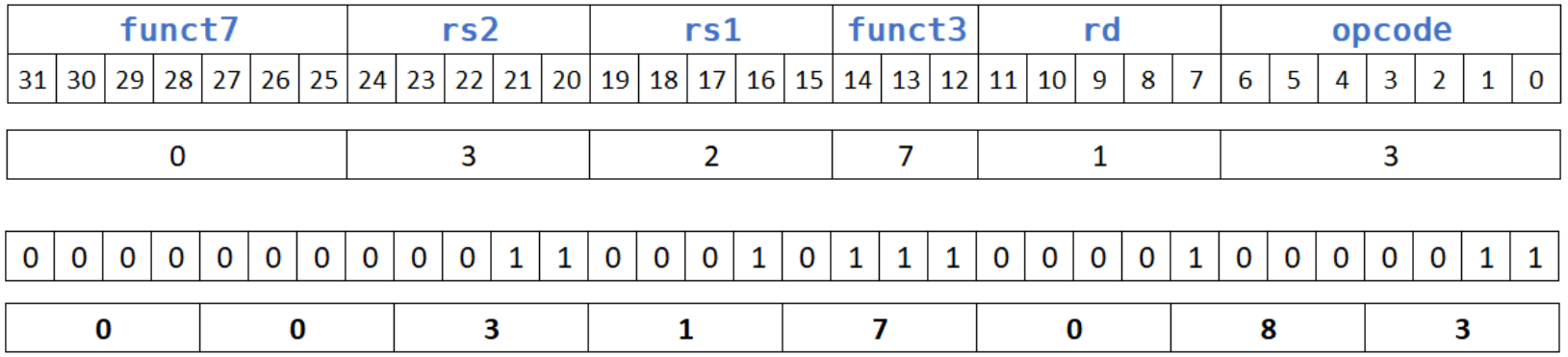
Кодирование инструкций RISC-V

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

Кодирование инструкций RISC-V

```
and x1, x2, x3
```

Inst	Name	FMT	Opcode	F3	F7	Description (C)	Note
and	AND	R	0000011	0x7	0x00	rd = rs1 & rs2	



R –type

0x00317083