

# RISC-V

Обзор системы команд

Yet another ISA

Зачем нужна новая система команд (RISC-V)?

# Yet another ISA

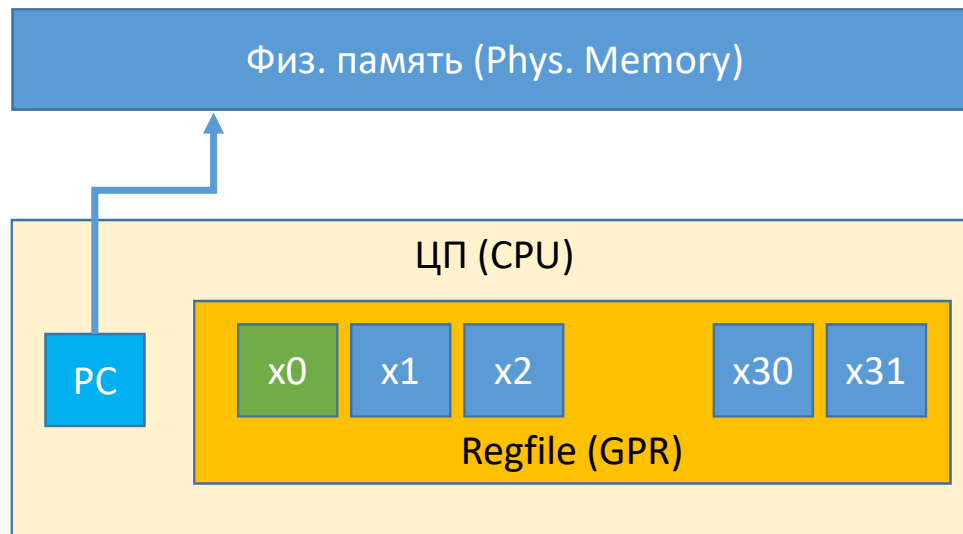
- Полностью открытая система команд(СК), которая доступна как для академических целей, так и для индустрии.
- СК подходящая для реализации в железе.
- СК без переусложнений (“over-architecting”).
- Модульная и расширяемая СК: большие возможности по пользовательскому расширению
- Поддержка IEEE-754 2008.
- Переменная длина инструкций.
- Полная поддержка виртуализации на уровне архитектуры

# Обзор RISC-V

- Основные варианты системы команд
  - RV32I – рассматриваемый в рамках лекций вариант RISC-V
  - RV64I
- Подмножества:
  - RV32E
  - RV64E

# Простейший RISC-V компьютер

## Элементы состояния компьютера



XLEN – константа разрядности (32/64/128), для RV32I XLEN=32

Double = 8 байт, Word = 4 байта, half = 2 байта

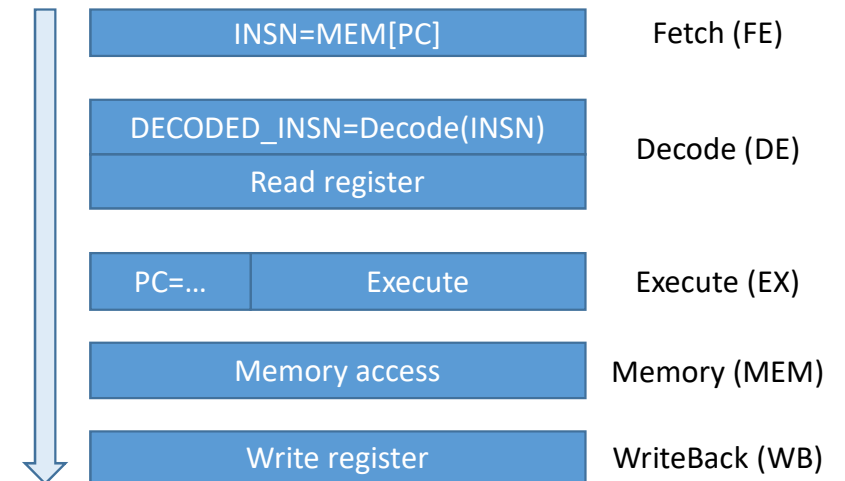
Размер регистров, PC и ширина адреса памяти равны XLEN

PC – указатель на инструкцию (адрес инструкции для исполнения)

x0...x31 – регистры, x0 всегда равен 0

Размер инструкции 1 word

## Классический RISC-конвейер (стадии исполнения инструкций)

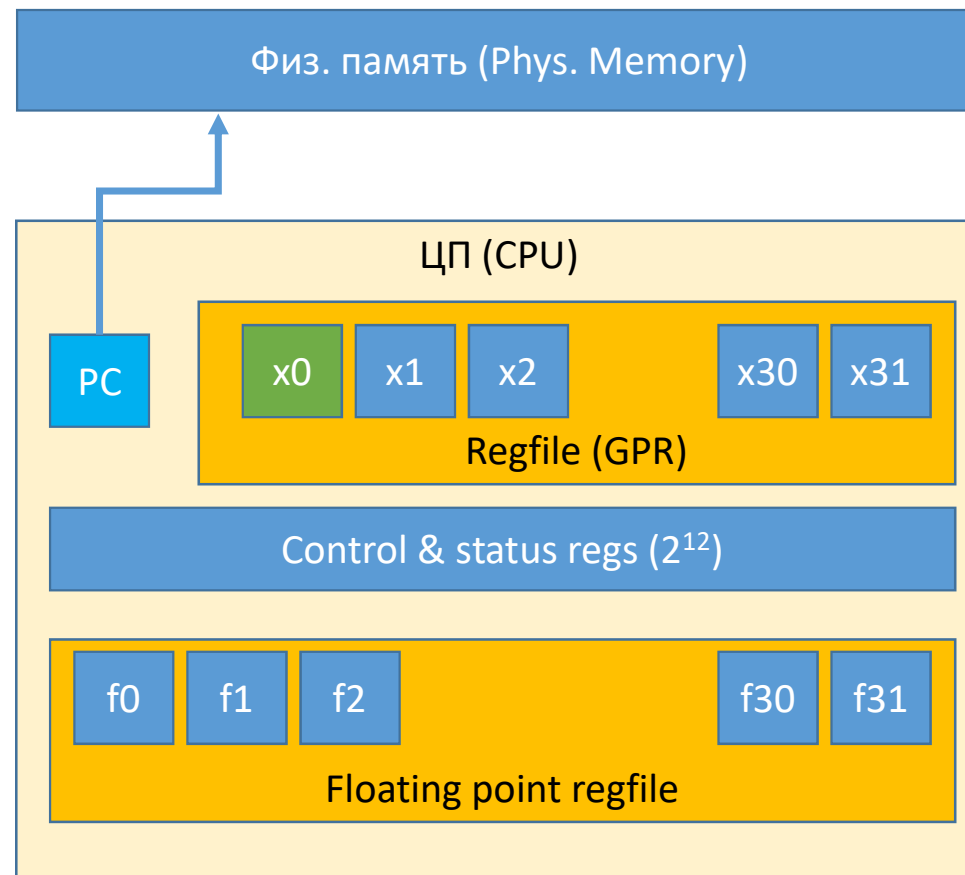


Примеры инструкций:

`add x1, x2, x3 ; -> regfile[1]=regfile[2]+regfile[3]`

`lw x1, 0x100(x2); -> regfile[1]=MEM[regfile[2]+0x100]`

# Элементы состояния RISC-V (дополнительные)



# Расширяемость и модульность RISC-V

- Имеется большой набор стандартных расширений
- Есть возможность добавлять собственные расширения
- Допускается возможность, чтобы нестандартные расширения СК противоречили стандартным (но не рекомендуется)

# Кодировка длины инструкции

	xxxxxxxxxxxxxxxxaa		16-bit (aa $\neq$ 11)	
В RV32I есть только этот формат	xxxxxxxxxxxxxxxxxxxx	xxxxxxxxxxxxbbb11	32-bit (bbb $\neq$ 111)	
	···xxxx	xxxxxxxxxxxxxxxxxxxx	xxxxxxxxxxx011111	48-bit
	···xxxx	xxxxxxxxxxxxxxxxxxxx	xxxxxxxxxxx0111111	64-bit
	···xxxx	xxxxxxxxxxxxxxxxxxxx	xnnnxxxxx1111111	(80+16*nnn)-bit, nnn $\neq$ 111
	···xxxx	xxxxxxxxxxxxxxxxxxxx	x111xxxxx1111111	Reserved for $\geq$ 192-bits
Byte Address:	base+4	base+2	base	



# Особенности кодирования

- Базовая кодировка определена как “little-endian”.
- Допускается возможность “big-endian” – используются пакеты по 16-бит.
- В рамках лекций используем только “little-endian”.

# Преимущества такого кодирования

- Базовая длина инструкции RV32I/RV64I – 30 бит – экономия 6,25% от площади кеша L1I.
- Возможность добавления сжатых инструкций (с 16-битной кодировкой) – RISC-V “C” extension (далее RVC)
- RV32I использует менее 1/8 от общего пространства кодирования.

# Исключения, ловушки и прерывания

- Исключение (exception) – исключительная ситуация в процессе исполнения кода и ассоциированная с конкретной ситуацией и с конкретной инструкцией.
- Ловушка (trap) – процесс передачи управления в обработчик исключения.
- Прерывание (interrupt) – внешнее событие (по отношению к потоку исполнения), происходящее асинхронно.
- Большинство исключений должны приводить к синхронной ловушке.

# RV32I и RV64I

А также RV32E/RV64E

# Регистры

- Регистр PC – счётчик инструкций.
- RV32I/RV64I/RV128I - 32 регистра общего назначения:
  - Регистры x0-x31.
  - x0/zero – аппаратный ноль.
  - Константа XLEN – длина регистров (32/64/128 бит соответственно).

# Регистры

- RV32E/RV64E – доступны только первые 16 регистров (x0-x15):
  - RV32E необходим для простых микроконтроллеров.
  - Площадь регистрового файла уменьшается более чем в 2 раза.
  - Общая площадь кристалла уменьшается на ~25%.
  - RV64E – сделан «за компанию» с RV32E.
  - Нет некоторых системных инструкций, обязательных для базовой СК.
  - Возникает исключение в случае обращения к x16-x31.

# Основные форматы инструкций

31	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[31:12]				rd	opcode		U-type

add x1, x2, x3 ; regfile[1]=regfile[2]+regfile[3] – R type  
rs1=000010 # x2  
rs2=000011 # x3  
rd=000001 # x1  
opcode=0110011 (2\*7)  
funct3=000 funct7=0000000

addi x1, x2, 3 ; regfile[1]=regfile[2]+ se(3) # I type  
rs1=000010 # x2  
imm=000 000 000 011 # 3  
rd=000001 # x1  
opcode=?  
funct3=000

```

int8_t:
-128 -> 8'b1000 0000
-127 -> 8'b1000 0001
...
-2 -> 8'b1111 1110
-1 -> 8'b1111 1111
0 -> 8'b0000 0000
1 -> 8'b0000 0001
...
127 -> 8'b0111 1111

int c=a+b; # add
unsigned x=y+z; # add

```

```

32'0 = 0000 0000 0000 0000 0000 0000 0000 0000
signed char - 8-bit
int - 32-bit
(signed char) (-2) + 5
-2 -> 1111 1111 1111 1111 1111 1111 1111 1110
(signed char) (-2) -> 8'b11111110
(int) (signed char) (-2) -> 32'b11...10
8->32:
(-2): 8'b111111110 ->
      1111 1111 1111 1111 1111 1111 1111 1110

(signed char) (255): 32'b 0000 ... 0000 1111 1111 ->
                    -1 (8'b1111 1111)

```

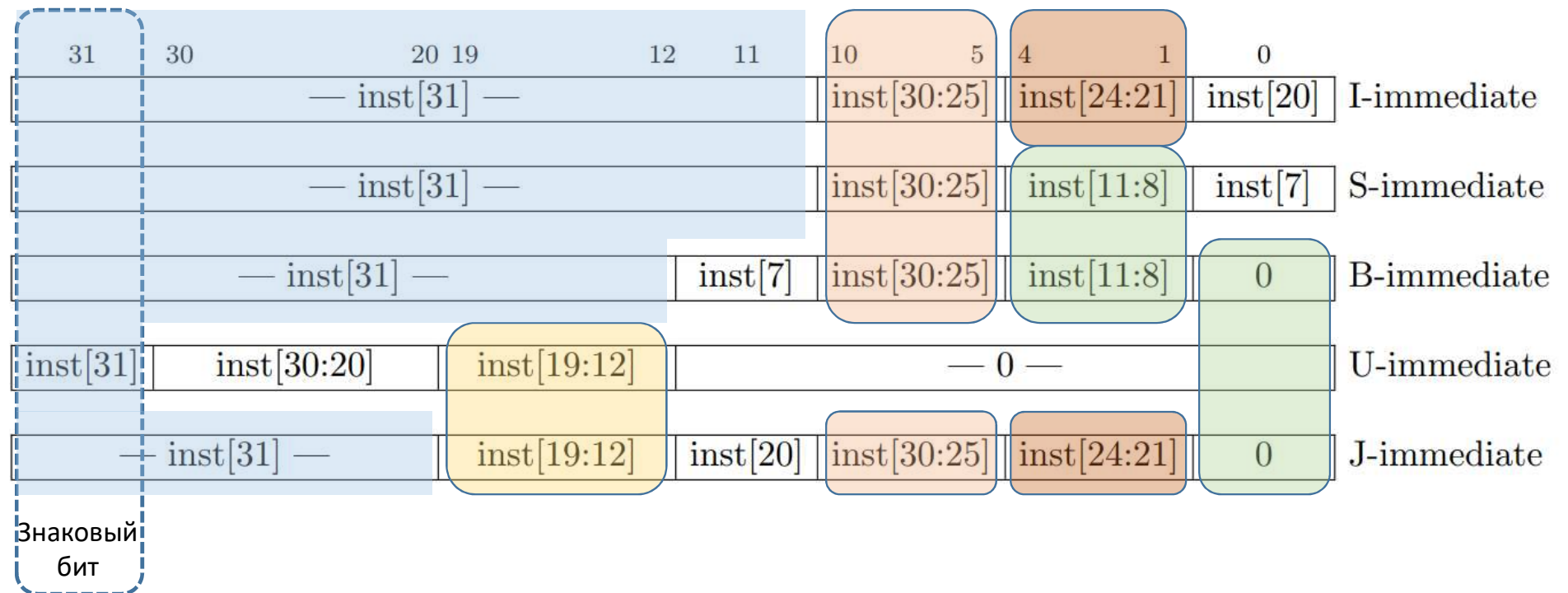


# Дополнительные форматы инструкций

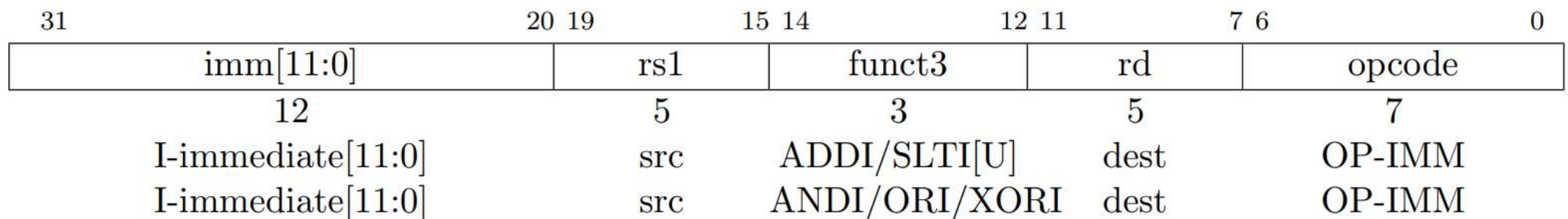
31	30	25	24	21	20	19	15	14	12	11	8	7	6	0				
funct7				rs2			rs1		funct3		rd			opcode		R-type		
imm[11:0]						rs1		funct3		rd			opcode		I-type			
imm[11:5]				rs2			rs1		funct3		imm[4:0]			opcode		S-type		
imm[12]		imm[10:5]			rs2			rs1		funct3		imm[4:1]		imm[11]		opcode		B-type
imm[31:12]										rd			opcode			U-type		
imm[20]		imm[10:1]				imm[11]		imm[19:12]				rd			opcode		J-type	

Зачем нужно такое странное расположение imm-полей?

# Декодирование чисел



# Целочисленные инструкции формата I



- `imm` - знаково-расширенное число (в т.ч. для `SLTIU`)
- `SLTI/SLTIU` – записывают 1 в `rd` если значение `rs1` меньше `imm`, иначе записывают 0.
- В 64-х битном режиме добавляется инструкция `ADDIW` с опкодом `OP-IMM-32`
- Почему нет инструкции вычитания?

# Инструкции сдвига в RV32I

31	25 24	20 19	15 14	12 11	7 6	0
imm[11:5]	imm[4:0]	rs1	funct3	rd	opcode	
7	5	5	3	5	7	
0000000	shamt[4:0]	src	SLLI	dest	OP-IMM	
0000000	shamt[4:0]	src	SRLI	dest	OP-IMM	
0100000	shamt[4:0]	src	SRAI	dest	OP-IMM	

- SLLI – логический сдвиг влево
- SRLI – логический сдвиг вправо
- SRAI – арифметический сдвиг вправо

# Инструкции сдвига в RV64I

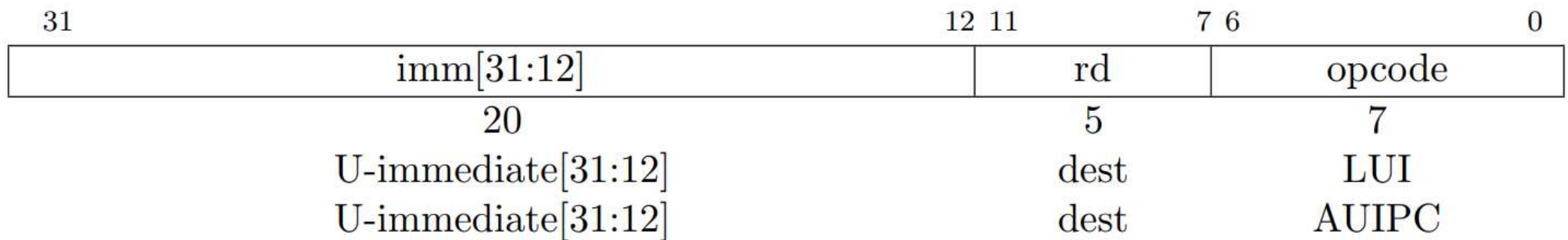
31		26	25	24	20 19		15 14		12 11		7 6		0
imm[11:6]		imm[5]		imm[4:0]		rs1		funct3		rd		opcode	
6		1		5		5		3		5		7	
000000		shamt[5]		shamt[4:0]		src		SLLI		dest		OP-IMM	
000000		shamt[5]		shamt[4:0]		src		SRLI		dest		OP-IMM	
010000		shamt[5]		shamt[4:0]		src		SRAI		dest		OP-IMM	
000000		0		shamt[4:0]		src		SLLIW		dest		OP-IMM-32	
000000		0		shamt[4:0]		src		SRLIW		dest		OP-IMM-32	
010000		0		shamt[4:0]		src		SRAIW		dest		OP-IMM-32	

# Псевдо-инструкции формата I

- Некоторые псевдо-инструкции могут быть оптимизированы в аппаратных реализациях.
- Канонические псевдо-инструкции удобны для написания кода и отладки (дизассемблирования).
- Псевдо-инструкции ассемблера для формата I

Псевдо-инструкция	Настоящая инструкция
<code>MV rd, rs1</code>	<code>ADDI rd, rs1, 0</code>
<code>NOT rd, rs</code>	<code>XOR rd, rs1, -1</code>
<code>NOP</code>	<code>ADDI x0, x0, 0</code>

# Целочисленные инструкции формата U



- LUI (load upper immediate) – загружает биты  $INST[31:12]$  в  $rd[31:12]$ , зануляет  $rd[11:0]$ :  $rd = se(INST[31:12] \ll 12)$
- AUIPC (add upper-immediate to pc) – добавляет PC к  $imm \ll 12$  и записывает результат в rd. Не рекомендуется использовать JAL для данной цели.  $rd = PC + se(INST[31:12] \ll 12)$ .
- В RV64I  $imm$  расширяется знаково.

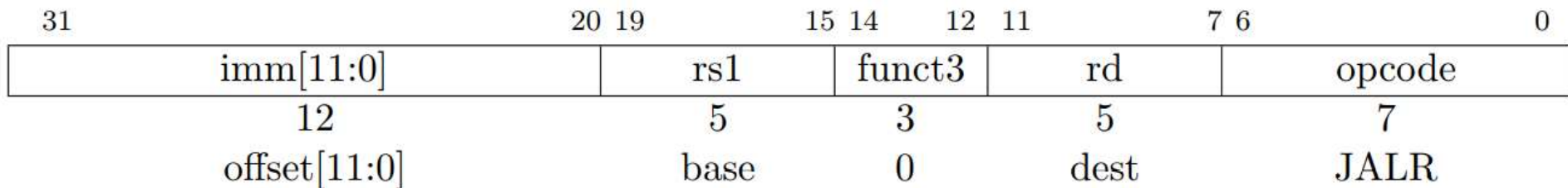
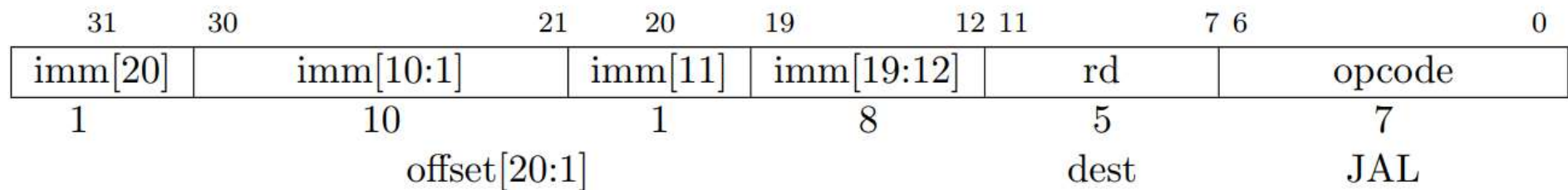
# Целочисленные инструкции формата R

31	25 24	20 19	15 14	12 11	7 6	0
funct7	rs2	rs1	funct3	rd	opcode	
7	5	5	3	5	7	
0000000	src2	src1	ADD/SLT/SLTU	dest	OP	
0000000	src2	src1	AND/OR/XOR	dest	OP	
0000000	src2	src1	SLL/SRL	dest	OP	
0100000	src2	src1	SUB/SRA	dest	OP	

- SLL/SRL/SRA используют для значения сдвига только младшие 5 бит в RV32I и 6 в RV64I.
- В RV64I добавляются дополнительные инструкции ADDW/SLLW/SRLW/SUBW/SRAW с OP-32.
- ADDW/SUBW расширяют знаково результат до 64-х бит.



# Инструкции безусловного перехода



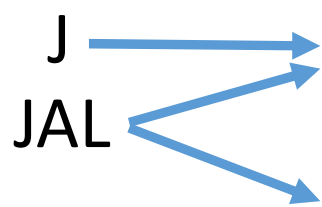
- JAL - инструкция прямого в диапазоне  $\pm 1$ МиБ.
  - $rd = PC + 4; PC = PC + se(imm \ll 1)$
- JALR – инструкция косвенного перехода.
  - $rd = PC + 4; PC = (rs1[XLEN-1:1] + imm[11:1]) \ll 1$

# Инструкции безусловного перехода

- Данные инструкции можно использовать для вызова процедур.
- Стандартный регистр возврата `x1`, альтернативный `x5`
- Простой переход – псевдоинструкция `J` – это `JAL x0, ....`
- Комбинация `LUI` и `JALR` позволяет реализовать любой абсолютный переход в RV32I.
- Комбинация `AUIPC` и `JALR` позволяет реализовать любой относительный переход в 32-хбитном диапазоне ( $\pm 2$ Гиб).
- Младший бит PC игнорируется в `imm` и в `rs1` у `JALR`. Это удобно для системного ПО, т.к. позволяет иметь 1-битный тег в указателе.
- Адрес перехода должен быть выровнен на 4, если нет RVC.

# Инструкции безусловного перехода

- Поддержка аппаратного ускорения стека вызовов/возвратов (RAS). *link* – это либо *x1*, либо *x5*.



<i>rd</i>	<i>rs1</i>	<i>rs1=rd</i>	RAS action
<i>!link</i>	<i>!link</i>	-	none
<i>!link</i>	<i>link</i>	-	pop
<i>link</i>	<i>!link</i>	-	push
<i>link</i>	<i>link</i>	0	push and pop
<i>link</i>	<i>link</i>	1	push

# Инструкции условного перехода

31	30	25 24	20 19	15 14	12 11	8	7	6	0
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]			opcode
1	6	5	5	3	4	1			7
offset[12,10:5]		src2	src1	BEQ/BNE	offset[11,4:1]				BRANCH
offset[12,10:5]		src2	src1	BLT[U]	offset[11,4:1]				BRANCH
offset[12,10:5]		src2	src1	BGE[U]	offset[11,4:1]				BRANCH

- Диапазон переходов  $\pm 4\text{КиБ}$ .
- Инструкции сравнивают значения регистров `rs1` и `rs2`.
- Условия: равенство, неравенство, меньше, больше-равно.
- `BGT/BGTU/BLE/BLEU` – псевдо-инструкции ассемблера с обратным порядком операндов для `BLT/BLTU/BGE/BGEU` соответственно.

## Псевдокод инструкций перехода

```

if (cond(reg[src1], reg[src2]))
    PC = PC + se(offset[12:1] << 1)
else
    PC = PC + 4
    
```

# Инструкции условного перехода

Рекомендации по оптимизации программного и аппаратного обеспечения:

- Частота «взятых» условных переходов должна быть ниже частоты не взятых.
- Переходы назад по умолчанию предсказываются «взятыми», переходы вперёд – «не взятыми».
- Для безусловных переходов нужно использовать  $\mathbb{J}$ , а не условный переход с постоянно выполненным условием.

# Примеры использования инструкций условного перехода

Пример на СИ	Возможный вариант реализации на RISC-V ассемблере (упрощенный)
<pre>int s = 0; for (int x = 0; x &lt; 5; x++) {     s += x; }</pre>	<pre>add x10, x0, x0 ; x10 = s add x11, x0, x0 ; x11 = x addi x12, x0, 5; x12 = 5 loop: add x10, x10, x11 addi x11, x11, 1 blt x11, x12, loop</pre>
<pre>// x = x &gt; 105 ? 105 : x; if (x &gt; 105) {     x = 105; }</pre>	<pre>; x = x16, 105 = x15 addi x15, x0, 105 blt x15, x16, false_cond mv x16, x15 false_cond: ...</pre>

# Проверка на переполнение

- **Беззнаковая:**

```
add t0, t1, t2
bltu t0, t1, overflow
```

- **Знаковая**

```
add t0, t1, t2
slti t3, t2, 0
slt t4, t0, t1
bne t3, t4, overflow
```

- **Оптимизированная знаковая**

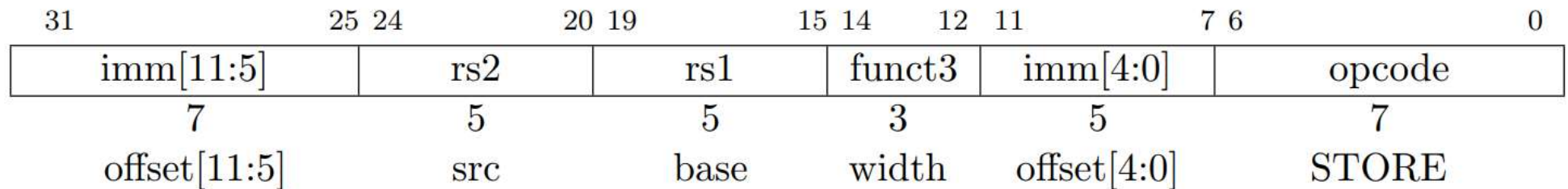
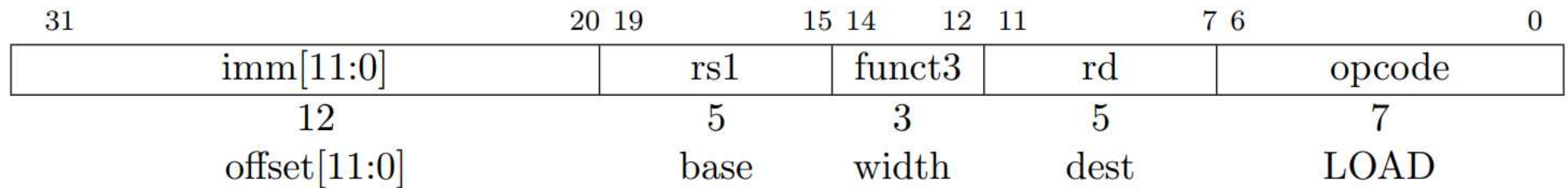
Для положительного imm:

```
addi t0, t1, +imm
blt t0, t1, overflow
```

- **Для 32-хбитных чисел в RV64I:**

```
add t0, t1, t2
addw t3, t1, t2
bne t0, t3, overflow
```

# Инструкции работы с памятью



- `funct3[1:0]` - кодирует размер обращения.
- `funct3[2]` – кодирует знаковое(0) или беззнаковое расширение (1).



# Инструкции работы с памятью

- RV32I:
  - `LB[U]/LH[U]/LW` читают 1/2/4 байта соответственно из памяти по адресу `rs1+imm[11:0]` и расширяют до `XLEN`.
  - `SB/SH/SW` записывают 1/2/4 байта соответственно.
- Изменения в RV64I:
  - `LD` – чтение 8 байт из памяти.
  - `LWU` – читает 4 байта и расширяет беззнаково до 64 бит.
  - `LW` – добавляется знаковое расширение.
  - `SD` – запись 8-ми байт в память.

# Инструкции работы с памятью

- Невыровненные обращения поддерживаются:
  - Не гарантируются атомарность.
  - Не гарантируется производительность.
  - Допускается реализация через обработчик исключения без поддержки на уровне АО.

# Примеры инструкций работы с памятью

Пример на СИ	Возможный вариант реализации на RISC-V ассемблере (упрощенный)
<pre>int s = 0; int arr[10] = ...; for (int i = 0; i &lt; 10; i++) {     s += arr[i]; }</pre>	<pre>... ; для простоты указатель на arr лежит в x20 - arr ; x21 - s      x22 - i      x23 - 10     add x21, x0, x0     add x22, x0, x0     addi x23, x0, \$10 loop: ; адрес i-го элемента arr: arr+i*sizeof(arr[0])=arr+i*4     slli x24, x22, 2          ; i*4 = (i &lt;&lt; 2)     add x24, x24, x20         ; arr+i*4     lw x24, \$0(x24)           ; regfile[24]=mem[regfile[24]]     add x21, x21, x24     addi x22, x22, \$1     blt x22, x23, loop ...</pre>

# Атомарность доступа

Код для Hart 0:

```
lw x5, 0x100(x2)
```

```
lw x6, 0x200(x2)
```

В каком порядке hart 1 увидит эти обращения в память? Зависит от:

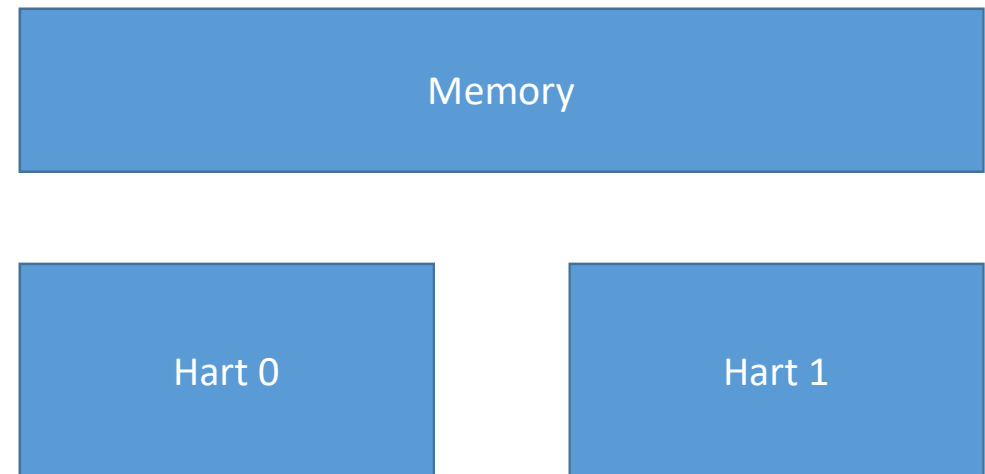
Модели памяти (популярные)

- TSO
- Relaxed model – **RISC-V**

Типа процессора:

- In-order – как правило TSO
- Out-of-order

Для гарантии порядка доступа есть несколько расширений системы команд



# Инструкции работы с системными регистрами (полноценные)

Инструкция	Алгоритм
CSRRW - Инструкция одновременного чтения-записи системного регистра	
csrrw rd, csr, rs1	<ol style="list-style-type: none"><li>1. regfile[rd]=sysregs[csr]</li><li>2. sysregs[csr]=regfile[rs1]</li></ol>
CSRRS - Инструкция установки отдельных битов в регистре	
csrrs rd, csr, rs1	<ol style="list-style-type: none"><li>1. regfile[rd]=sysregs[csr]</li><li>2. sysregs[csr]=sysregs[csr]   regfile[rs1]</li></ol>
CSRRC - Инструкция сброса отдельных битов в регистре	
csrrc rd, csr, rs1	<ol style="list-style-type: none"><li>1. regfile[rd]=sysregs[csr]</li><li>2. sysregs[csr]=sysregs[csr] &amp; ~regfile[rs1]</li></ol>

# Инструкции работы с системными регистрами - визуализация

csrrs rd, csr, rs1

[illegible]

```
csrrc rd, csr, rs1
```

[illegible]

# Расширение “М”

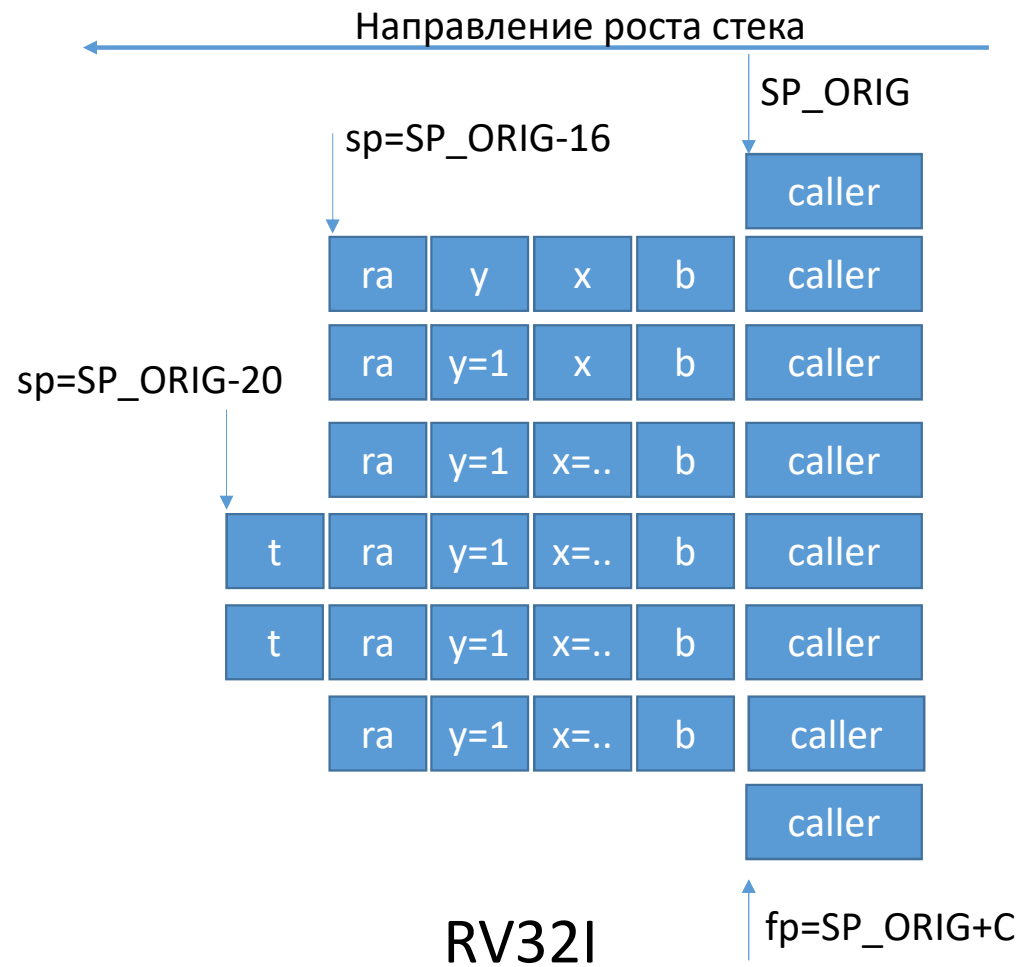
см. документацию - <https://riscv.org/technical/specifications/>

Вопрос - Почему умножение / деление не попадают в стандартную СК?

Алгоритм умножения на Си (скрытый ответ на вопрос)

```
unsigned x, y;  
unsigned s=0;  
for (int i = 0; i < 32; i++)  
{  
    if (y & (1 << i)) s += x << i;  
}
```

# Frame pointer



1. alloca
2. "int x[n];"

```
int foo(int b, int c)
{
    int x, y = 1;
    x = b * 2;
    {
        int t = x + b; x = y; y = tmp;
    }
}
```



# RVC

RISC-V “C” Standard Extension for Compressed Instructions.

# RVC: ключевые особенности

- Позволяет кодировать инструкции длиной в 16 бит.
- Не добавляет нового режима (как в ARM и MIPS).
- Можно смешивать с обычными инструкциями длиной 32 бита.
- JAL/JALR не вызывают исключения при невыровненном переходе.

# Оценки производительности

- 50-60% инструкций имеют короткие эквиваленты.
- Уменьшение размеров кода на 25-30%.

Исследование в Беркли показало:

- Снижение нагрузки на канал памяти на 25-30%.
- Снижение промахов кэш инструкций на 20-25% - эквивалентно увеличению кэша в 2 раза.

# Условие применимости коротких команд

- Поддерживаются только наиболее популярные инструкции.
- Выполнено одно из условий:
  - Кодировается короткое число (в поле imm)
  - Используется один из следующих регистров: x0(zero)/x1(link)/x2(sp).
  - Регистр назначения совпадает с первым регистром-источником.
  - Все регистры попадают в список 8ми наиболее популярных.

# Идеология RVC

- Инструкции RVC являются подмножеством базовой СК, что существенно упрощает валидацию.
- RVC не требует поддержки на уровне компилятора, но хороший компилятор может существенно улучшить качество кода.
- RVC не является частью базовой СК и обязательной к реализации.

# Архитектура RVC

- Большинство инструкций совпадает между RV32C, RV64C и RV128C.
- Для большинства инструкций с полем `imm` запрещены числа 0. С нулевым значением кодируется другая инструкция.
- В кодировках с 5-битными полями регистров местоположение `rd` такое же, как и в базовой СК.
- В кодировках с 3-битными полями регистров кодируются регистры `x8-x15`.

## 3-битная кодировка регистров

RVC Register Number

Integer Register Number

Integer Register ABI Name

Floating-Point Register Number

Floating-Point Register ABI Name

000	001	010	011	100	101	110	111
x8	x9	x10	x11	x12	x13	x14	x15
s0	s1	a0	a1	a2	a3	a4	a5
f8	f9	f10	f11	f12	f13	f14	f15
fs0	fs1	fa0	fa1	fa2	fa3	fa4	fa5

# Форматы инструкций

Format	Meaning	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
CR	Register	funct4				rd/rs1				rs2				op					
CI	Immediate	funct3		imm		rd/rs1				imm				op					
CSS	Stack-relative Store	funct3		imm						rs2				op					
CIW	Wide Immediate	funct3		imm								rd'		op					
CL	Load	funct3		imm				rs1'		imm		rd'		op					
CS	Store	funct3		imm				rs1'		imm		rs2'		op					
CB	Branch	funct3		offset				rs1'		offset				op					
CJ	Jump	funct3		jump target												op			



# Инструкции чтения-записи памяти

- Смещение расширяются беззнаково (imm/offset).
- Смещение умножается на размер обращений
- Инструкции делятся на 2 группы
  - Базовый регистр –  $sr(x2)$ , второй регистр – любой.
  - Оба регистра из RVC-подмножества.
- Причина появления отдельной кодировки для  $sr$  в том, что суммарно прологи/эпилоги функций составляют значительную долю кода (по мнению авторов RISC-V).

# Инструкции чтения данных со стека

Формат CI															
15	13	12	11								7	6			0
funct3			imm	rd					imm					op	
3			1	5					5					2	
C.LWSP			offset[5]	dest $\neq$ 0					offset[4:2 7:6]					C2	
C.LDSP			offset[5]	dest $\neq$ 0					offset[4:3 8:6]					C2	
C.LQSP			offset[5]	dest $\neq$ 0					offset[4 9:6]					C2	
C.FLWSP			offset[5]	dest					offset[4:2 7:6]					C2	
C.FLDSP			offset[5]	dest					offset[4:3 8:6]					C2	

Инструкция	Эквивалент	Вариант СК
C.LWSP	lw rd, offset[7:2] (x2)	RV32C/RV64C/RV128C
C.LDSP	ld rd, offset[8:3] (x2)	RV64C/RV128C
C.LQSP	lq rd, offset[9:4] (x2)	RV128C
C.FLWSP	flw rd, offset[7:2] (x2)	RV32FC
C.FLDSP	fld rd, offset[8:3] (x2)	RV32DC/RV64DC

# Инструкции записи данных в стек

15	13 12	7 6	2 1	0
funct3	imm	rs2	op	
3	6	5	2	
C.SWSP	offset[5:2 7:6]	src	C2	
C.SDSP	offset[5:3 8:6]	src	C2	
C.SQSP	offset[5:4 9:6]	src	C2	
C.FSWSP	offset[5:2 7:6]	src	C2	
C.FSDSP	offset[5:3 8:6]	src	C2	

Инструкция	Эквивалент	Вариант СК
C.SWSP	sw rs2, offset[7:2] (x2)	RV32C/RV64C/RV128C
C.SDSP	sd rs2, offset[8:3] (x2)	RV64C/RV128C
C.SQSP	sq rs2, offset[9:4] (x2)	RV128C
C.SLWSP	fsw rs2, offset[7:2] (x2)	RV32FC
C.FSDSP	fsd rs2, offset[8:3] (x2)	RV32DC/RV64DC

# Инструкции чтения данных

15	13 12	10 9	7 6	5 4	2 1	0
funct3	imm	rs1'	imm	rd'	op	
3	3	3	2	3	2	
C.LW	offset[5:3]	base	offset[2 6]	dest	C0	
C.LD	offset[5:3]	base	offset[7:6]	dest	C0	
C.LQ	offset[5 4 8]	base	offset[7:6]	dest	C0	
C.FLW	offset[5:3]	base	offset[2 6]	dest	C0	
C.FLD	offset[5:3]	base	offset[7:6]	dest	C0	

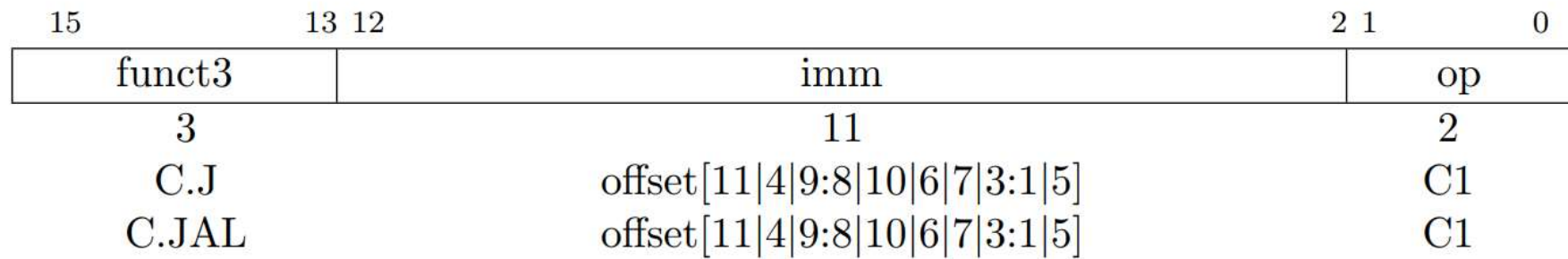
Инструкция	Эквивалент	Вариант СК
C.LW	lw rd', offset[6:2](rs1')	RV32C/RV64C/RV128C
C.LD	ld rd', offset[7:3](rs1')	RV64C/RV128C
C.LQ	lq rd', offset[8:4](rs1')	RV128C
C.FLW	flw rd', offset[6:2](rs1')	RV32FC
C.FLD	fld rd', offset[7:3](rs1')	RV32DC/RV64DC

# Инструкции записи данных

15	13 12	10 9	7 6	5 4	2 1	0
funct3	imm	rs1'	imm	rs2'	op	
3	3	3	2	3	2	
C.SW	offset[5:3]	base	offset[2 6]	src	C0	
C.SD	offset[5:3]	base	offset[7:6]	src	C0	
C.SQ	offset[5 4 8]	base	offset[7:6]	src	C0	
C.FSW	offset[5:3]	base	offset[2 6]	src	C0	
C.FSD	offset[5:3]	base	offset[7:6]	src	C0	

Инструкция	Эквивалент	Вариант СК
C.SW	sw rs2', offset[6:2] (rs1')	RV32C/RV64C/RV128C
C.SD	sd rs2', offset[7:3] (rs1')	RV64C/RV128C
C.SQ	sq rs2', offset[8:4] (rs1')	RV128C
C.FSW	fsw rs2', offset[6:2] (rs1')	RV32FC
C.FSD	fsd rs2', offset[7:3] (rs1')	RV32DC/RV64DC

# Инструкции управления



Инструкция	Эквивалент	Вариант СК	Диапазон
C.J	jal x0, offset[11:1]	RV32C/RV64C/RV128C	±2КиБ
C.JAL	jal x1, offset[11:1]	RV32C	±2КиБ

# Инструкции управления

15	12 11	7 6	2 1	0
funct4	rs1	rs2	op	
4	5	5	2	
C.JR	src≠0	0	C2	
C.JALR	src≠0	0	C2	

Инструкция	Эквивалент	Вариант СК
C.JR	jalr x0, rs1, 0	RV32C/RV64C/RV128C
C.JALR	jalr x1, rs1, 0	RV32C/RV64C/RV128C

# Инструкции управления

15	13 12	10 9	7 6	2 1	0
funct3	imm	rs1'	imm	op	
3	3	3	5	2	
C.BEQZ	offset[8 4:3]	src	offset[7:6 2:1 5]	C1	
C.BNEZ	offset[8 4:3]	src	offset[7:6 2:1 5]	C1	

Инструкция	Эквивалент	Диапазон
C.BEQZ	beq rs1, x0, offset[8:1]	$\pm 256$
C.BNEZ	bne rs1, x0, offset[8:1]	$\pm 256$



# Целочисленные инструкции

15	13	12	11	7	6	2	1	0
funct3	imm[5]	rd	imm[4:0]	op				
3	1	5	5	2				
C.LI	imm[5]	dest $\neq$ 0	imm[4:0]	C1				
C.LUI	nzuimm[17]	dest $\neq$ {0, 2}	nzuimm[16:12]	C1				

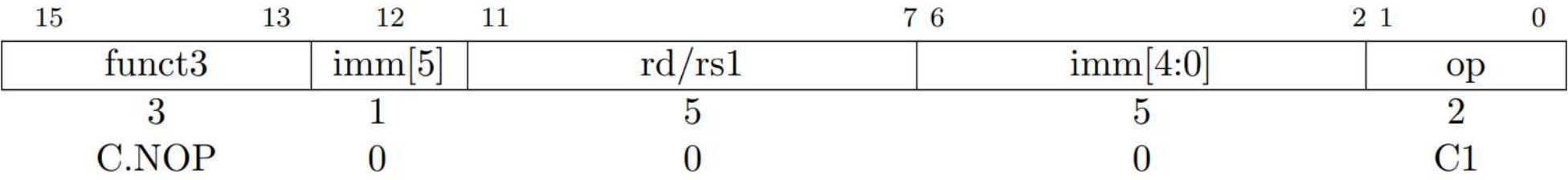
Инструкция	Эквивалент
C.LI	addi rd, x0, imm[5:0]
C.LUI	lui rd, nzuimm[17:12]

# Целочисленные инструкции

15	13	12	11	7	6	2	1	0
funct3	imm[5]	rd/rs1	imm[4:0]	op				
3	1	5	5	2				
C.ADDI	nzimm[5]	dest	nzimm[4:0]	C1				
C.ADDIW	imm[5]	dest≠0	imm[4:0]	C1				
C.ADDI16SP	nzimm[9]	2	nzimm[4 6 8:7 5]	C1				

Инструкция	Эквивалент	Вариант СК	Комментарий
C.ADDI	<code>addi rd, rd, nzimm[5:0]</code>	Все	
C.ADDIW	<code>addiw rd, rd, imm[5:0]</code>	RV64C/RV128C	Если imm=0 - sext.w rd
C.ADDI16SP	<code>addi x2, x2, nzimm[9:4]</code>	Все	

# NOP



# Пространство кодирования

[illegible]

# Список инструкций (1)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
000		0									0		00		<i>Illegal instruction</i>		
000		nzuimm[5:4 9:6 2 3]									rd'		00		C.ADDI4SPN ( <i>RES</i> , <i>nzuimm=0</i> )		
001		uimm[5:3]			rs1'			uimm[7:6]			rd'		00		C.FLD ( <i>RV32/64</i> )		
001		uimm[5:4 8]			rs1'			uimm[7:6]			rd'		00		C.LQ ( <i>RV128</i> )		
010		uimm[5:3]			rs1'			uimm[2 6]			rd'		00		C.LW		
011		uimm[5:3]			rs1'			uimm[2 6]			rd'		00		C.FLW ( <i>RV32</i> )		
011		uimm[5:3]			rs1'			uimm[7:6]			rd'		00		C.LD ( <i>RV64/128</i> )		
100		—												00		<i>Reserved</i>	
101		uimm[5:3]			rs1'			uimm[7:6]			rs2'		00		C.FSD ( <i>RV32/64</i> )		
101		uimm[5:4 8]			rs1'			uimm[7:6]			rs2'		00		C.SQ ( <i>RV128</i> )		
110		uimm[5:3]			rs1'			uimm[2 6]			rs2'		00		C.SW		
111		uimm[5:3]			rs1'			uimm[2 6]			rs2'		00		C.FSW ( <i>RV32</i> )		
111		uimm[5:3]			rs1'			uimm[7:6]			rs2'		00		C.SD ( <i>RV64/128</i> )		

## Список инструкций (2)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
000			0												01	C.NOP
000			nzimm[5]					rs1/rd≠0							01	C.ADDI ( <i>HINT</i> , <i>nzimm</i> =0)
001			imm[11 4 9:8 10 6 7 3:1 5]												01	C.JAL ( <i>RV32</i> )
001			imm[5]					rs1/rd≠0							01	C.ADDIW ( <i>RV64/128</i> ; <i>RES</i> , <i>rd</i> =0)
010			imm[5]					rd≠0							01	C.LI ( <i>HINT</i> , <i>rd</i> =0)
011			nzimm[9]					2							01	C.ADDI16SP ( <i>RES</i> , <i>nzimm</i> =0)
011			nzimm[17]					rd≠{0, 2}							01	C.LUI ( <i>RES</i> , <i>nzimm</i> =0; <i>HINT</i> , <i>rd</i> =0)
100			nzuimm[5]			00		rs1'/rd'							01	C.SRLI ( <i>RV32 NSE</i> , <i>nzuimm</i> [5]=1)
100			0			00		rs1'/rd'							01	C.SRLI64 ( <i>RV128</i> ; <i>RV32/64 HINT</i> )
100			nzuimm[5]			01		rs1'/rd'							01	C.SRAI ( <i>RV32 NSE</i> , <i>nzuimm</i> [5]=1)
100			0			01		rs1'/rd'							01	C.SRAI64 ( <i>RV128</i> ; <i>RV32/64 HINT</i> )
100			imm[5]			10		rs1'/rd'							01	C.ANDI
100			0			11		rs1'/rd'		00					01	C.SUB
100			0			11		rs1'/rd'		01					01	C.XOR

## Список инструкций (3)

100	0	11	rs1'/rd'	10	rs2'	01	C.OR
100	0	11	rs1'/rd'	11	rs2'	01	C.AND
100	1	11	rs1'/rd'	00	rs2'	01	C.SUBW <small>(RV64/128; RV32 RES)</small>
100	1	11	rs1'/rd'	01	rs2'	01	C.ADDW <small>(RV64/128; RV32 RES)</small>
100	1	11	—	10	—	01	<i>Reserved</i>
100	1	11	—	11	—	01	<i>Reserved</i>
101	imm[11 4 9:8 10 6 7 3:1 5]					01	C.J
110	imm[8 4:3]		rs1'	imm[7:6 2:1 5]		01	C.BEQZ
111	imm[8 4:3]		rs1'	imm[7:6 2:1 5]		01	C.BNEZ

# Недостатки RVC

- Проектирование СК с сжатым набором инструкций всегда является компромиссом, поэтому не стоит критиковать авторов, но:
- Декодирование RVC значительно сложнее, чем базовой СК.
  - Есть ли выигрыш по энергопотреблению??
- Инструкции более не выровнены:
  - Могут пересекать границы строки кэша/страницы/окна выборки.
  - Существенное усложнение реализации и валидации.
- Отсутствие варианта кодирования только RVC для простых микроконтроллеров без базовой СК.