UC Berkeley
Teaching Professor
Dan Garcia

CS61C

Great Ideas in Computer Architecture

(a.k.a. Machine Structures)

UC Berkeley
Teaching Professor
Lisa Yan

RISC-V Instruction Formats, Part II

Disassembling the S-Format

(Last lecture's Slide 33)

- **What is the corresponding assembly instruction?**

Store opcode

31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
0000001	01110	00010	010	00100	0100011

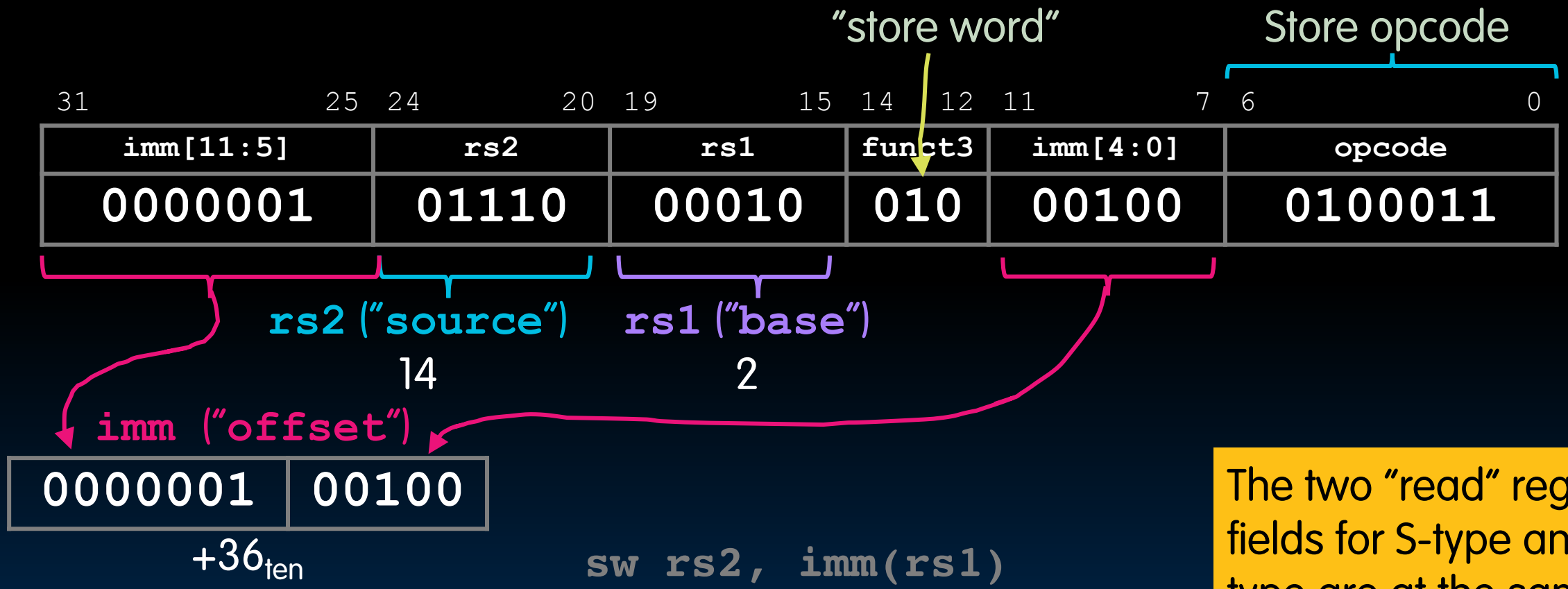
S-Format Lookup table:

			funct3		opcode	
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	sb
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	sh
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	sw

Disassembling the S-Format

(Last lecture's Slide 33)

- What is the corresponding assembly instruction?




```
sw rs2, imm(rs1)
```

```
sw x14, 36(x2)
```

The two "read" reg. fields for S-type and R-type are at the same bit positions.

VIEW FROM THE TOP

Conversations with Global Leaders

Driving Moore's Law:

A Fireside Chat with Ann Kelleher and Dean Tsu-Jae King Liu

ANN KELLEHER

EXECUTIVE VICE PRESIDENT &
GENERAL MANAGER OF TECHNOLOGY DEVELOPMENT
INTEL

Wednesday, Sept. 21 | 12-1 p.m.
Banatao Auditorium, Sutardja Dai Hall

Event is free and open to the Berkeley community.
Lunch will be provided.

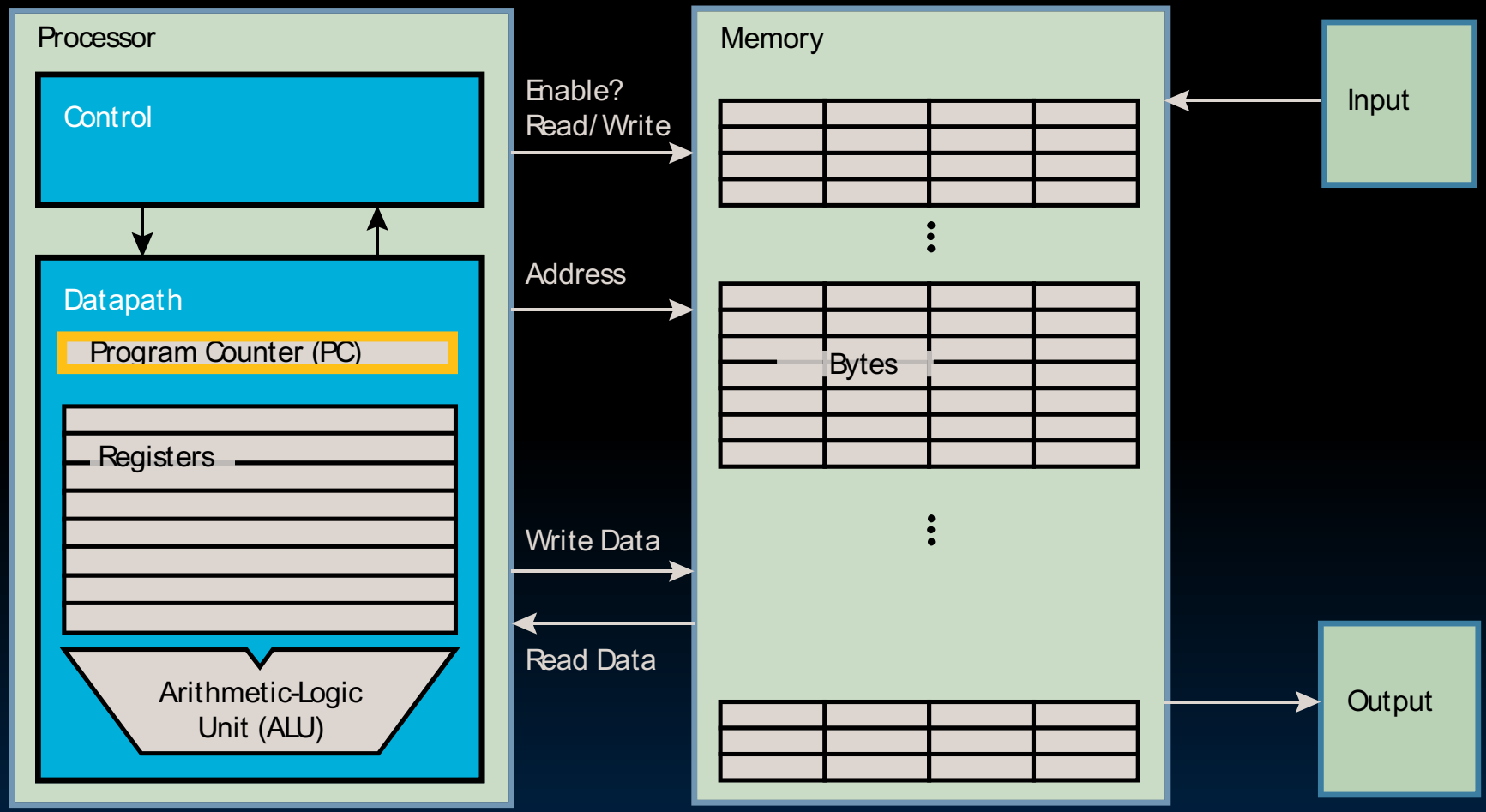
Co-hosted with the Society of Women Engineers
engineering.berkeley.edu/events

Berkeley Engineering

Updating the Program Counter (PC)

- Updating the Program Counter (PC)
- B-Format Layout
- J-Format Layout
- U-Format: Long Immediates
- `jalr`: I-Format

The Program Counter (PC) 程序计数器



The Program Counter (PC) is a special internal register inside processor holding byte address of the instruction being executed.

It is separate from the 32-registers Register File!

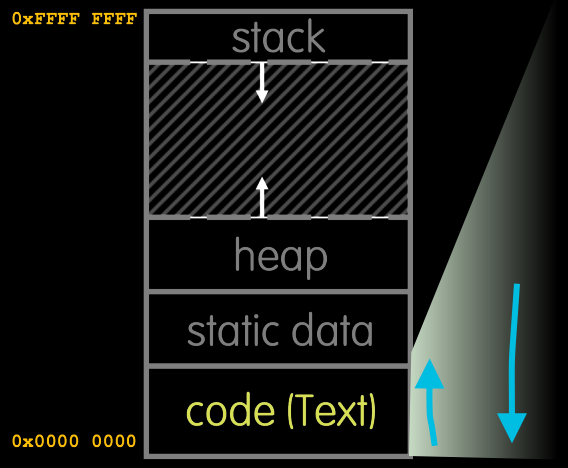
处理器内存接口

Processor-Memory Interface

I/O-Memory Interfaces

I/O内存接口

PC = PC + 4 for most instructions



Program Address Space

程序地址空间

Address	Machine Code	Original Assembly Code	
0x0c	00A98863	Loop: beq x19,x10,End	
0x10	00A90933	add x18,x18,x10	← PC
0x14	FFF98993	addi x19,x19,-1	
0x18	FF5FF06F	j Loop	
0x1c	00A98863	End: …	

machine code correspond to assembly code

Address	Machine Code	Original Assembly Code	
0x0c	00A98863	Loop: beq x19,x10,End	
0x10	00A90933	add x18,x18,x10	
0x14	FFF98993	addi x19,x19,-1	← PC
0x18	FF5FF06F	j Loop	
0x1c	00A98863	End: …	

Registers before execution

PC: 0000 0010

RegFile	
x10	0000 0001
x18	0000 0000
x19	0000 0002

+4

PC: 0000 0014

RegFile	
x10	0000 0001
x18	0000 0001
x19	0000 0002

Branches Update PC to "Jump"

Kris Kross - Jump (Official Video) -

Unconditional Branches

```
0x0c  Loop:  beq x19,x10,End
0x10         add x18 x18,x10
0x14         addi x19,x19,-1
0x18         j Loop
0x1c  End:   …
```

PC = 0000 0018

RegFile	
x10	0000 0001
x18	0000 0001
x19	0000 0001

-12

```
0x0c  Loop:  beq x19,x10,End
0x10         add x18,x18,x10
0x14         addi x19,x19,-1
0x18         j Loop
0x1c  End:   …
```

PC = 0000 000c

RegFile	
x10	0000 0001
x18	0000 0001
x19	0000 0001

Conditional Branch (if branch taken)

```
0x0c  Loop:  beq x19,x10,End
0x10         add x18,x18,x10
0x14         addi x19,x19,-1
0x18         j Loop
0x1c  End:   …
```

PC = 0000 000c

RegFile	
x10	0000 0001
x18	0000 0001
x19	0000 0001

+16

```
0x0c  Loop:  beq x19,x10,End
0x10         add x18,x18,x10
0x14         addi x19,x19,-1
0x18         j Loop
0x1c  End:   …
```

PC = 0000 001c

RegFile	
x10	0000 0001
x18	0000 0001
x19	0000 0001

If branch not taken, then PC = PC + 4

PC-Relative Addressing

相对地址

- PC-Relative Addressing: **Supply a signed offset to update PC.**
  - `PC = PC + byte_offset`
  - "*Position-Independent Code*": If all of code moves, *relative offsets* don't change!

PC = PC + byte-offset

```
0x0c  Loop:  beq x19,x10,End
0x10         add x18, x18,x10
0x14         addi x19, x19,-1
0x18         j Loop
0x1c  End:   …
```

Take branch:
PC = PC + 16
Don't take branch:
PC = PC + 4

```
0x400  Loop:  beq x19,x10,End
0x404         add x18, x18,x10
0x408         addi x19, x19,-1
0x40c         j Loop
0x410  End:   …
```

  - Branches generally change the PC by a small amount, therefore in RISC-V instructions we encode relative offsets as *signed immediates*.

- **Contrast with:** Absolute Addressing

绝对地址

  - Supply new address to overwrite PC. `PC = new_address`

覆盖

  - Use sparingly: Brittle to code movement/need to build 32-bit immediate (more later)

B-Format Layout

- Updating the Program Counter (PC)
- B-Format Layout
- J-Format Layout
- U-Format: Long Immediates
- `jalr`: I-Format

RISC-V Conditional Branches

- **Example:** `beq rs1,rs2,Label`
- **Conditional branches need:**
  - Two read registers, no destination register (like S-Format)
  - Some way to encode label (i.e., where to jump to)
- **RISC-V uses PC-relative addressing to encode labels.**
  - If we *don't* branch: `PC = PC + 4` bytes
  - If we *do* branch: `PC = PC +` (relative byte offset to `Label`)
- **B-Format Fields:**
  - 7b (`opcode`) + 3b (`funct3`) + 5b (`rs1`) + 5b (`rs2`) = 20b
  - 12b immediate field to represent PC's relative offset.
    - Signed two's complement means we can represent ± 2^{11} "units" from the PC.

相对偏移 偏移量 二进制补码

Scaling the Offset to Maximize Branch Range

The 12-bit immediate field for conditional branches is:

A. In units of 1 byte.

- One branch reaches ± 2^{11} bytes from PC → ± 2^{9} x 32-bit instructions.
  - Never branch into middle of instruction.

B. In units of 2 bytes (16-bit "half-words").

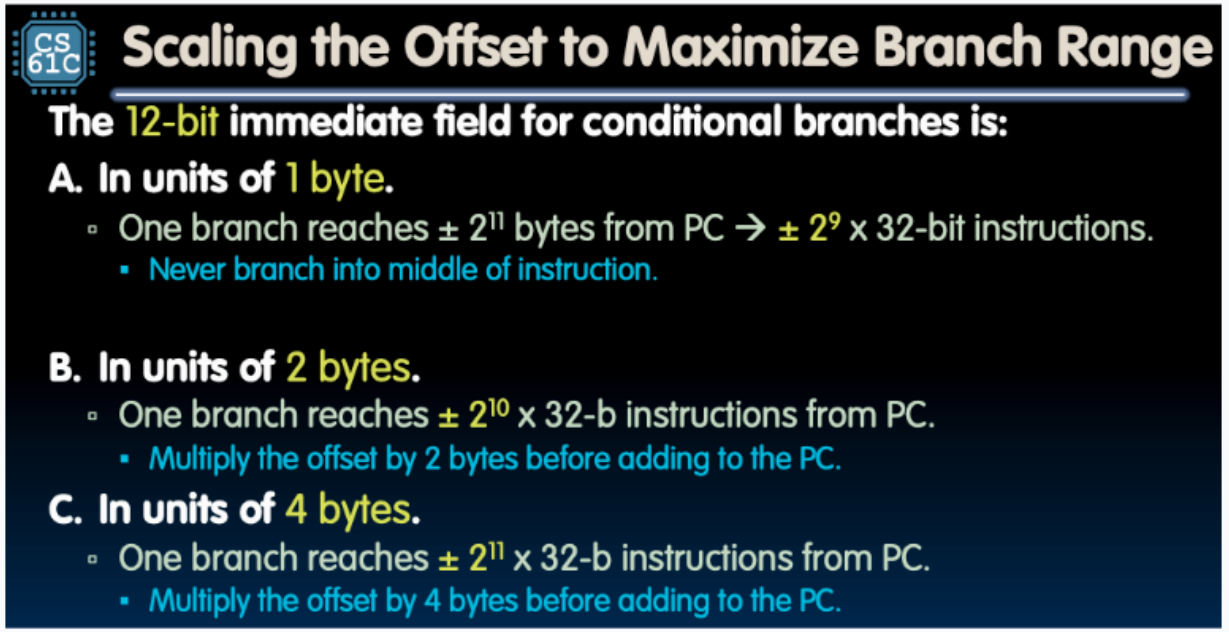
- One branch reaches ± 2^{10} x 32-b instructions from PC.
  - Multiply the offset by 2 before adding to the PC.

C. In units of 4 bytes (32-bit words).

- One branch reaches ± 2^{11} x 32-b instructions from PC.
  - Multiply the offset by 4 before adding to the PC.

pollev.com/yanl

How do branches encode PC relative offset?



A

B

C

None of the above

Scaling the Offset to Maximize Branch Range

The 12-bit immediate field for conditional branches is:

❌ **In units of 1 byte.**
- One branch reaches ± 2^{11} bytes from PC → ± 2^{9} x 32-bit instructions.
  - Never branch into middle of instruction.
- *Wasteful*! Lower 2 bits would always be wasted (always 00).

✅ **In units of 2 bytes (16-bit "half-words").** half word
- One branch reaches ± 2^{10} x 32-b instructions from PC.
  - Multiply the offset by 2 before adding to the PC.

❌ **In units of 4 bytes (32-bit words).**
- One branch reaches ± 2^{11} x 32-b instructions from
  - Multiply the offset by 4 before adding to the PC.
- Does not support 16-bit compressed instructions!

Branch immediate range [-2048, +2047] represents PC relative offsets [-4096, +4094] in 2-byte increments.

RISC-V Feature: 16-bit instructions

- **RISC-V Base ISA for RV32, RV64, RV128 all have 32-bit wide instructions, but it supports extensions:** 32位指令可拓展
  - 16b compressed instructions
  - Variable-length instructions that are multiples of 16b in length
- **RISC-V therefore *scales the branch offset by 2 bytes*, even when there are no 16b instructions!**
- **Implications for this class (which uses RISC-V processors that only support 32-bit instructions):**
  - Half of the possible branch targets will be errors.
  - RISC-V Conditional Branches can only reach $\pm 2^{10}$ 32-bit instructions on either side of PC.

B-Format Instruction Layout

- B-Format (textbook: SB-Type) close to S-Format:

```
opname  rs1,rs2,Label
```

All *conditional* branch instructions have opcode `1100011`.

Immediate represents relative offset in increments of 2 bytes ("half-words")

- To compute new PC: PC = PC + `byte_offset`.
- 12 immediate bits imply $\pm 2^{10}$ 32-bit instructions reachable:
  - 1 bit: 2's complement (allow +/– offset)
  - 1 bit: half-word/16-b instruction support

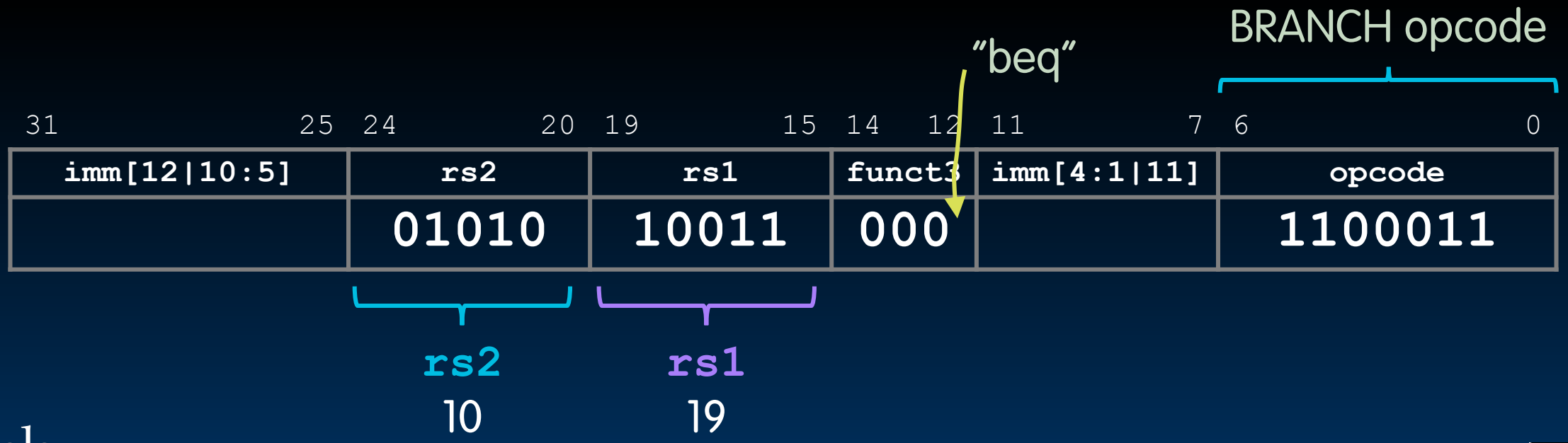
(Lowest bit of offset is always zero, so no need to store in instruction.)

B-Format Example (1/2)

```
Loop:  beq x19,x10,End
       add x18,x18,x10
       addi x19,x19,-1
       j Loop
End:   … # target instr
```

How do we encode the beq instruction in this RISC-V code?

1. Fill in fields (other than immediate).



B-Format Example (2/2)

```
Loop:  beq x19,x10,End
       add x18,x18,x10
       addi x19,x19,-1
       j Loop
End:   … # target instr
```

+4 instruct-ions

How do we encode the `beq` instruction in this RISC-V code?

2. Construct branch offset (in bytes), then encode immediate.

31 25	24 20	19 15	14 12	11 7	6 0
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode
0000000	01010	10011	000	10000	1100011

- +4 32-bit instructions = +16 byte offset

12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	0	0	0	0

13-bit signed `byte_offset` ("`imm`")

Why Swirl Immediate Bits Around? HW Design!

- **Recall: RISC-V register field positions consistent across instr. formats.**
- **RISC-V also tries to keep bit positions of immediates consistent:**

	31 25	24 20	19 15	14 12	11 7	6 0
I-type	imm[11:5]	imm[4:0]	rs1	funct3	rd	opcode
	yxxxxxx	wwwwv				
S-type	imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
	yxxxxxx				wwwwv	
B-type	imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode
	zxxxxxx				wwwwy	

- Instruction Bit 31 is always the *sign bit* (highest bit to sign extend in immediate)
- B-type has a *13-bit* immediate encoding b/c of implicit zero in Imm bit 0.

Between S and B, only two bits change meaning:
Instruction Bit 7 → S: Imm Bit 0; B: Imm Bit 11
Instruction Bit 31→ S: Imm Bit 11; B: Imm Bit 12

RISC-V immediate bit encoding is optimized to reduce hardware cost.

All six RV32 B-Format Instructions

			funct3		opcode	
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	beq
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	bne
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	blt
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	bge
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	bltu
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	bgeu

Conditionally Branching Even Further

- **Conditional Branches are typically used for if-else statements, for/while loops.**
  - Usually pretty small (<50 lines of C code) no more than 50 lines
- **B-Format has limited range:**
  - ± 2^{10} 32-bit instructions from current instruction
- **What if destination is further away?**
  - Enter the *unconditional jump*! jump

J-Format Layout

- Updating the Program Counter (PC)
- B-Format Layout
- J-Format Layout
- U-Format: Long Immediates
- `jalr`: I-Format

The `jal` instruction and `j` pseudoinstruction

- **`jal` does a Jump And Link (i.e., "link and jump"):** link and jump

```
jal rd, Label
```

  - Jump to `Label`
  - Write address of the *following* instruction to `rd`.

- **Two use cases:**

1. Call a function (and simultaneously save return address in named register)

```
jal ra, FuncLabel
```

2. Unconditional branch
   - `j` pseudoinstruction: jump and discard return address

```
j Label
```

⇕

```
jal x0, Label
```

jump discards the return address

The `jal` instruction and `j` pseudoinstruction

- **`jal` does a** Jump And Link **(i.e., "link and jump"):**

```
jal rd, Label
```

  - Jump to **`Label`** *(i.e., add relative offset to PC)*
  - Write address of the *following* instruction to **`rd`**.

```
PC = PC + offset
rd = PC + 4
```

- **Two use cases:**

1. Call a function (and simultaneously save return address in named register)

```
jal ra, FuncLabel
```

2. Unconditional branch
   - `j` pseudoinstruction: jump and discard return address
   - Often used to conditionally branch further than B-Types

```
j Label
```

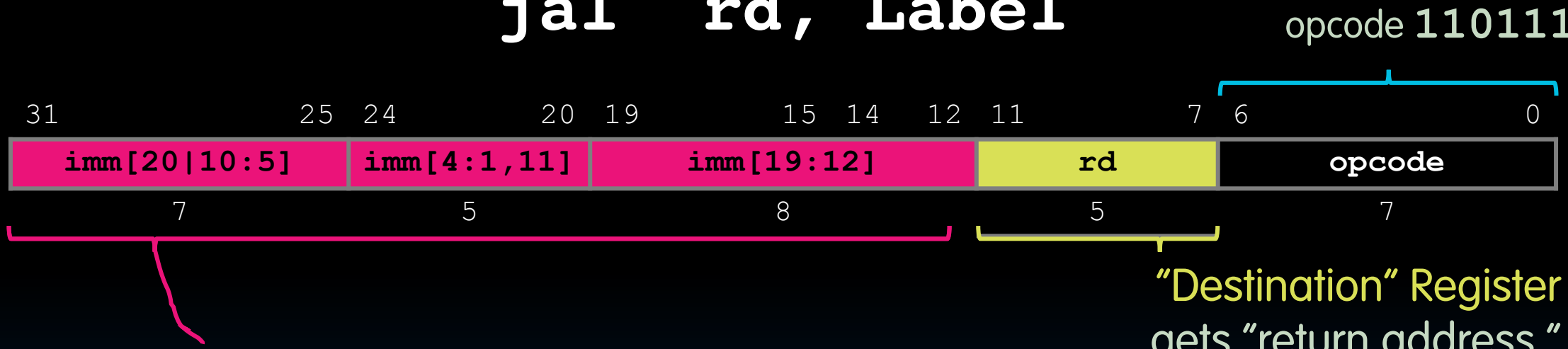
⇕

```
jal x0, Label
```

J-Format Instruction Layout

- J-Format (textbook: UJ-Type) used only for `jal`:

```
jal   rd, Label
```



"Destination" Register gets "return address."

`rd = PC + 4`

Immediate represents relative offset in increments of 2 bytes.

- To compute new PC: PC = PC + **`byte_offset`**.
- 20 immediate bits imply $\pm 2^{18}$ 32-bit instructions reachable:
  - 1 bit: 2's complement (allow +/– offset), 1 bit: half-word/16-b instruction support
- Immediate bit encoding optimized to reduce HW cost

What about jumping *further*? Next: How to load 32-bit immediate into a register!

U-Format: Long Immediates

- Updating the Program Counter (PC)
- B-Format Layout
- J-Format Layout
- U-Format: Long Immediates
- `jalr`: I-Format

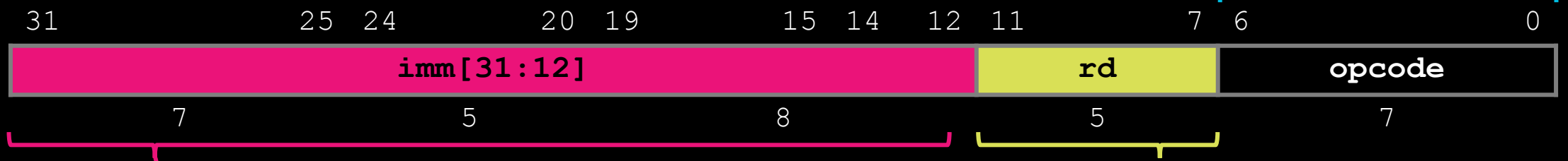
U-Format Instruction Layout

- "Upper Immediate" **instructions:**

```
opname   rd,immed
```

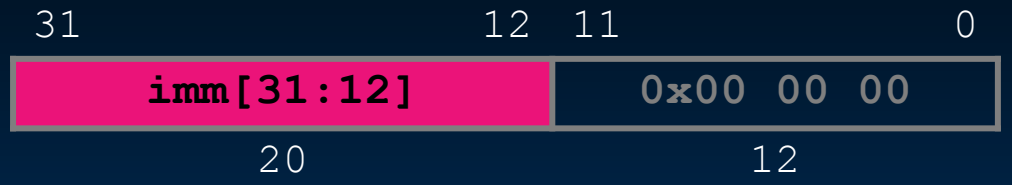
U-Format opcodes:

```
lui   0110111
auipc 0010111
```



Immediate represents *upper 20 bits* of a 32-bit immediate operand `imm = immed << 12`.

"Destination" Register
Saves result of operation.



Long Immediate ("`imm`")

`lui` and `addi` creates Long Immediates

`lui rd,immed`

- **The `lui` instruction,** Load Upper Immediate:
  - Write a 20-bit immediate value into the upper 20-bits of register rd.
  - Clear the lower 12 bits.

`rd = immed << 12`

li = lui + addi

- **`lui` together with an `addi` (to set lower 12 bits) can create any 32-bit value in a register:**

```
lui  x10, 0x87654      # x10 = 0x87654000
addi x10, x10, 0x321   # x10 = 0x87654321
```

The **`li`** *pseudoinstruction* (*Load Immediate*) resolves to **`lui` + `addi`** as needed, e.g., **`li x10,0x87654321`**.

`lui` Edge Case: `addi` Extends Sign

- **How should we set the immediate `0xB0BACAFE`?**
  - Unfortunately:

```
lui x10, 0xB0BAC      # x10 = 0xB0BAC000
addi x10, x10, 0xAFE  # x10 = 0xB0BABAFE
```

  - Recall: **`addi`** sign-extends the 12-bit immediate.
  - If "sign bit" set, subtracts 1 from the upper 20-bits!

```
  0xB0BAC000
+ 0xFFFFFAFE
```

```
  0xB0BAC          0x000
+ 0xFFFFF -1     + 0xAFE
  0xB0BAB          0xAFE
```

concat

0xB0BABAFE

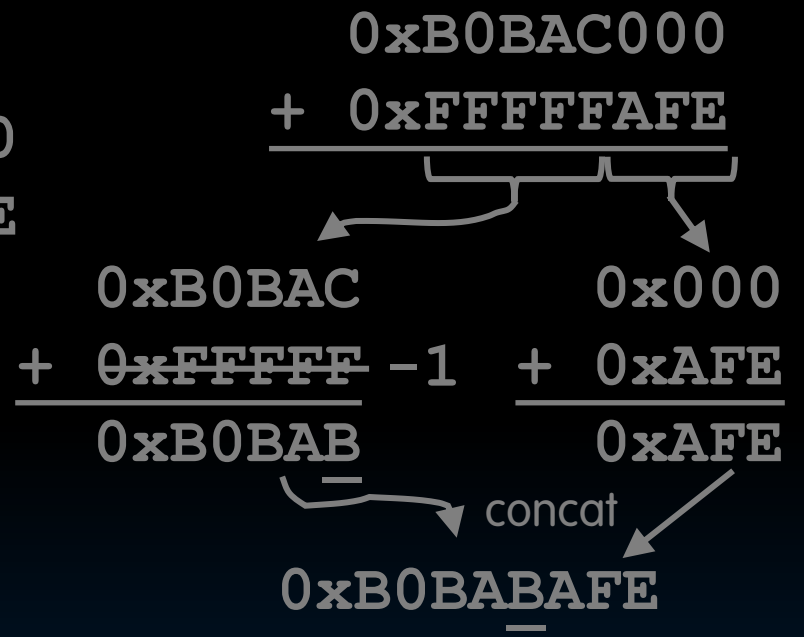
0x B0BA C AFE

`lui` Edge Case: `addi` Extends Sign

- **How should we set the immediate `0xB0BACAFE`?**
  - ⚠ Unfortunately:

```
lui  x10, 0xB0BAC      # x10 = 0xB0BAC000
addi x10, x10, 0xAFE   # x10 = 0xB0BABAFE
```

    - Recall: **`addi`** sign-extends the 12-bit immediate.
    - If "sign bit" set, subtracts 1 from the upper 20-bits!



lower down a bit

✅ **Solution: If 12-bit immediate is negative, add 1 to the upper 20-bit load.**

```
lui  x10, 0xB0BAD      # x10 = 0xB0BAD000
addi x10, x10, 0xAFE   # x10 = 0xB0BACAFE
```

Use pseudoinstructions! `li` automatically handles this edge case.

`auipc` loads the PC into the Register File

`auipc rd,immed`

- **The `auipc` instruction**
  Adds an Upper Immediate **to the** PC.

`rd = PC + (immed << 12)`

- **Example:**

```
auipc x5, 0xABCDE     # x5 = PC + 0xABCDE
```

- **In Practice:**

```
Label: auipc x5, 0    # puts address of Label in x5
```

  - Loads the PC into a register accessible by other instructions.

`auipc` is most often used together with **`jalr`** to do PC-relative addressing with super large offsets.

`jalr`: I-Format

- Updating the Program Counter (PC)
- B-Format Layout
- J-Format Layout
- U-Format: Long Immediates
- **`jalr`**: I-Format

The `jalr` instruction and `jr` pseudoinstruction

- **`jalr` does a Jump And Link Register:**

```
jalr rd,rs1,imm
```

  - Jump to `rs1 + imm`.
  - Write address of the *following* instruction to `rd`.

```
PC = rs1 + imm
rd = PC + 4
```

- **Use cases (so far):**

1. Return to caller

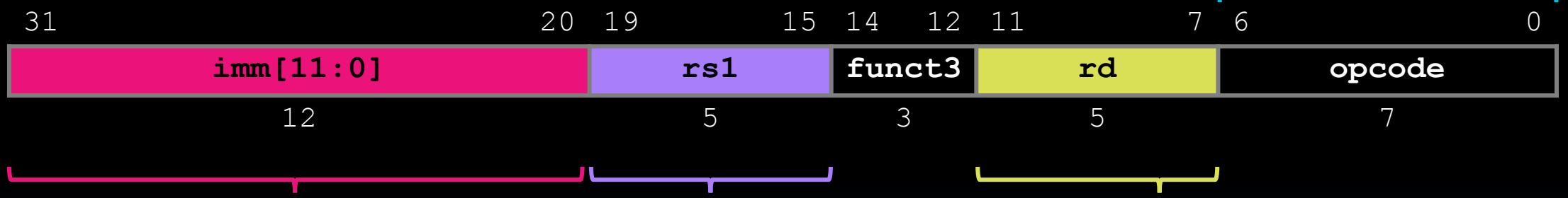
```
jr ra   ⇔   jalr x0, ra, 0
```

I-Format Instruction Layout: `jalr`

- **`jalr` uses I-Format:**

`jalr rd,rs1,imm`

opcode `1100111`



Immediate

Source Register

"Destination" Register gets "return address."

`rd = PC + 4`

`imm` and **`rs1`** are added together to update PC.

`PC = rs1 + imm`

⚠ Note I-Type!
Unlike B-type, J-type, **`jalr`** will not multiply **`imm`** by 2.

- Immediate offset therefore must be written in *units of bytes*.

The `jalr` instruction and `jr` pseudoinstruction

- **`jalr` does a** Jump And Link Register:

```
jalr rd,rs1,imm
```

  - Jump to **`rs1 + imm`**.
  - Write address of the *following* instruction to **`rd`**.

```
PC = rs1 + imm
rd = PC + 4
```

- **Use cases:**

1. Return to caller

```
jr ra   ⟺   jalr x0, ra, 0
```

Unlike **`jal`** (relative to PC), **`jalr`** addresses are relative to **`rs1`**, which is modifiable by arithmetic instructions. We can do bigger jumps!

2. Call a function (and save return address) at any 32-bit *absolute address*.

```
lui x1 <hi20bits>
jalr ra, x1, <lo12bits>
```

3. Jump *PC-relative* with a *32-bit offset*.

```
auipc x1 <hi20bits>
jalr x0, x1, <lo12bits>
```

"And in Conclusion…"

- We've covered (almost) the entire RV32 ISA!
- Practice assembling and disassembling!

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI

0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
0000	pred	succ	00000	000	00000	0001111	FENCE
0000	0000	0000	00000	001	00000	0001111	FENCE.I
000000000000			00000	000	00000	1110011	ECALL
000000000001			00000	000	00000	1110011	EBREAK
csr			rs1	001	rd	1110011	CSRRW
csr			rs1	010	rd	1110011	CSRRS
csr			rs1	011	rd	1110011	CSRRC
csr			zimm	101	rd	1110011	CSRRWI
csr			zimm	110	rd	1110011	CSRRSI
csr			zimm	111	rd	1110011	CSRRCI

Not in CS61C

https://riscv.org/technical/specifications/