

Instruction Encoding - 2



Z. Jerry Shi

Department of Computer Science and Engineering
University of Connecticut

CSE3666: Introduction to Computer Architecture

RISC-V Core Instruction Format

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

imm in the table is the 32-bit immediate the processor uses in computation
It may not be the immediate we write in the source code, e.g., in LUI

Encoding of branches ([Section 2.10](#))

Jump far away

Review: LUI examples

- Load 32-bit 0x003D0500 into register x19
 - LUI set the higher 20 bits (adjusted for sign bit from the lower 12 bits)
 - ADDI set the lower 12 bits
 - Immediate is sign extended

```
lui x19, 0x3D0 # 0x003D0
```

x19:

0000 0000 0011 1101 0000	0000 0000 0000
--------------------------	----------------

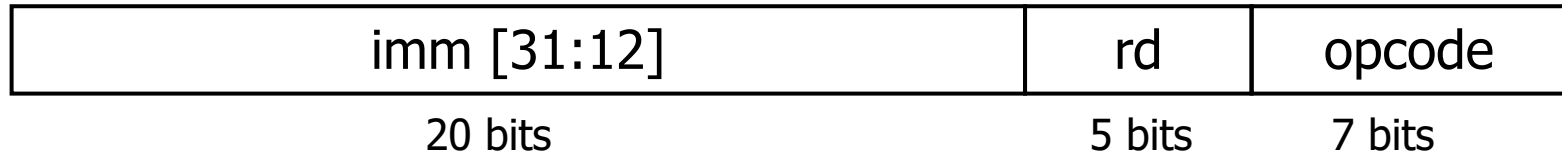
```
addi x19, x19, 0x500
```

x19:

0000 0000 0011 1101 0000	0101 0000 0000
--------------------------	----------------

We need allocate 20 bits in instruction word for immediate in LUI

Encoding of LUI: U-type Instructions



- Fields in U-type
 - opcode: operation code
 - rd: destination register number
 - imm[31:12]:
The higher 20 bits of the 32-bit immediate **the processor gets**
- Opcode:
LUI: 0b 011 0111

Exercise:

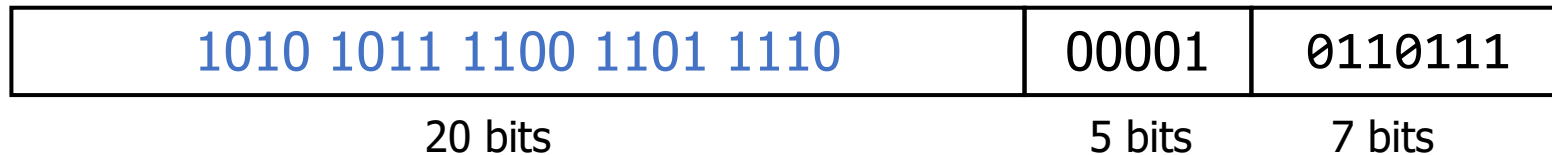
```
lui    x1, 0xABCDE    # the 32-bit immediate is 0xABCDE000
```

The immediate

the immediate written in assembly language

lui x1, 0xABCDE

The immediate field in the machine code

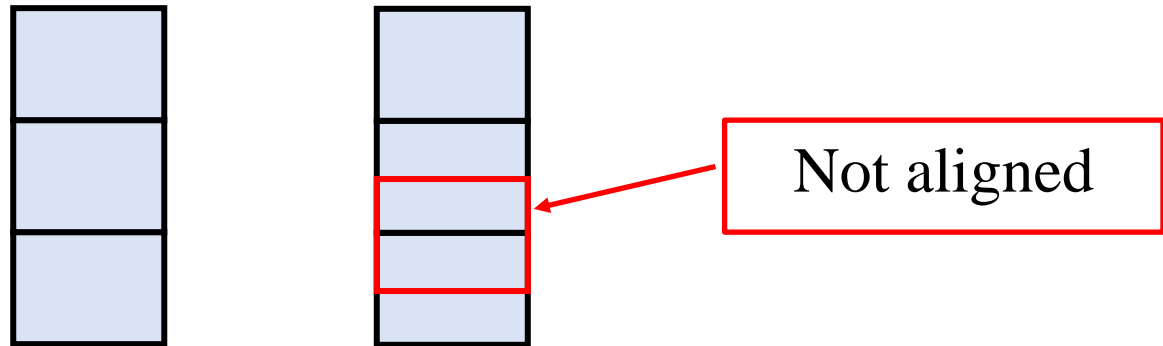


The 32-bit immediate the processor uses is 0xABCDE000



Address alignment

- Words and half-words have alignment issue
- When aligned,
 - Addresses of words must be a multiple of 4
 - Addresses of half-words must be a multiple of 2
- RISC-V does not require data to be aligned
- However, instructions must be aligned



Format for branches

beq rs1, rs2, Label

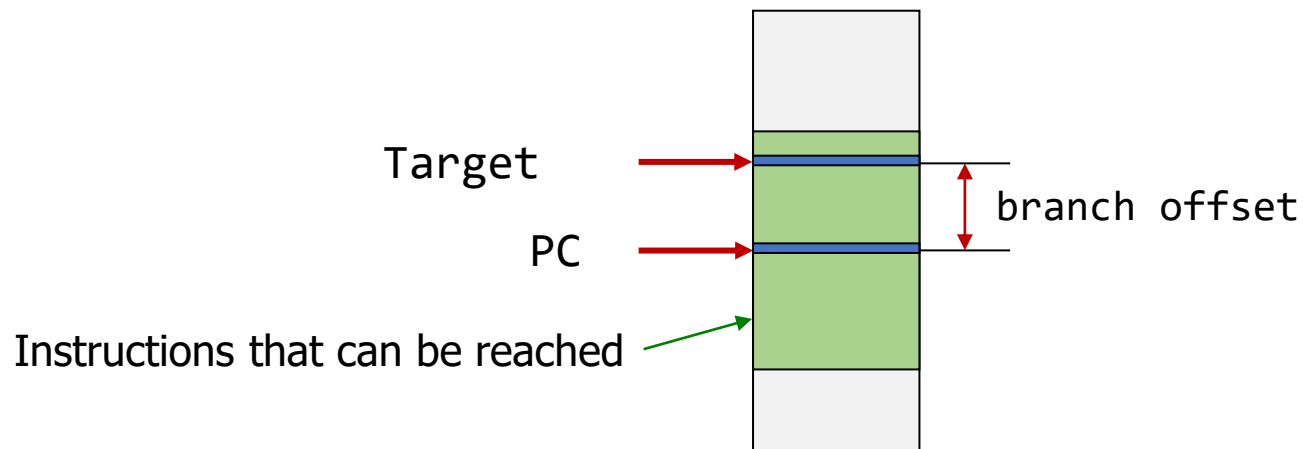
All branches are similar.

- Which format is the best for branches?
 - A. R-type
 - B. I-Type
 - C. S-Type
- What problem do we have?

R:	funct7	rs2	rs1	funct3	rd	opcode
I:	imm [11:0]		rs1	funct3	rd	opcode
S:	imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode

Offset in Branches

- The full address is 32 bits
 - Not that many bits in instruction!
 - There are only 12 bits to encode the target address
- Instead of the full address, we store **the offset**, a smaller number in the 12 bits
 - The offset is the distance between the target address and PC
 - The target address is relative to PC (**PC-relative addressing**)



Encoding and executing branches

- When encoding branches, assembler calculates the offset:

$$\text{branch offset} = \text{target address} - \text{PC}$$

> 0: jump forward
== 0: the branch itself
< 0: jump backward

- When processor executes the instruction, it
 - Forms a 32-bit immediate from the 12 bits in machine code
 - Calculates the target address as

$$\text{target address} = \text{PC} + \text{imm}$$



imm == branch offset

Instruction alignment

- All instructions are 4 bytes in RV32I, and
- Instructions must be aligned
- What can we say about the branch offset?

$$\text{branch offset} = \text{target address} - \text{PC}$$

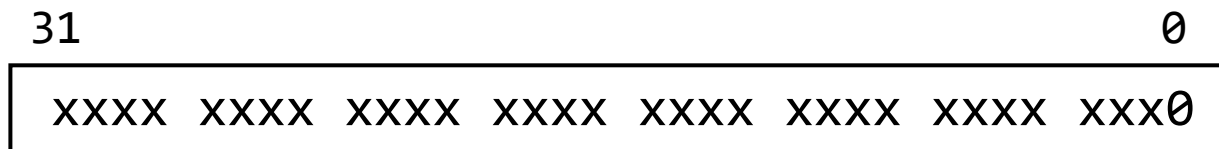
Example and question: offset in branch

What are the offsets in the following branches?

```
I1:      add      s0, s0, t0
I2:      bne      x8, x9, I8
I3:      NOP
I4:      NOP
I5:      NOP
I6:      NOP
I7:      beq      x0,  x0, I1
I8:      NOP
I9:      NOP
```

Supporting RISC-V C extension

- However, **compressed instructions (in C extension) are of 16 bits**
 - Although we are not using those instructions
- **They can be located at any even address**, not just a multiple of 4
 - For example, 2, 6, 0xFFCC02, ...
- RISC-V keeps the ability of branching to any even addresses
 - The offset is always even, but not necessarily a multiple of 4 in C ext.

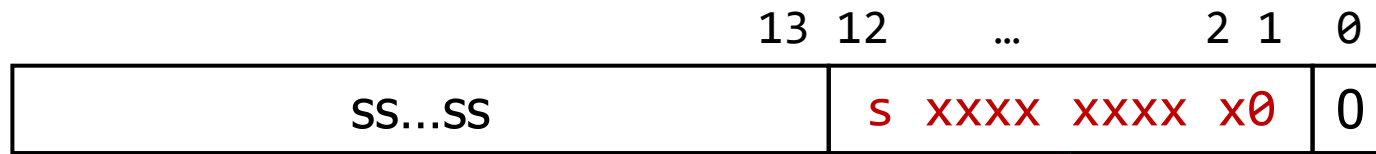


What bits should we keep in the machine code?

What about the bits in the higher end?

Offset bits keep in machine code

- The 12 bits going to the machine code are **offset[12:1]**
 - offset[0] is always 0, so do not keep it in the machine code



Higher 19 bits must be the sign

Bits 12 to 1 are kept in machine code
Bit 1 may not be 1 for C extension

Processor builds a 32-bit immediate imm
from these 12 bits, which is the same as
offset

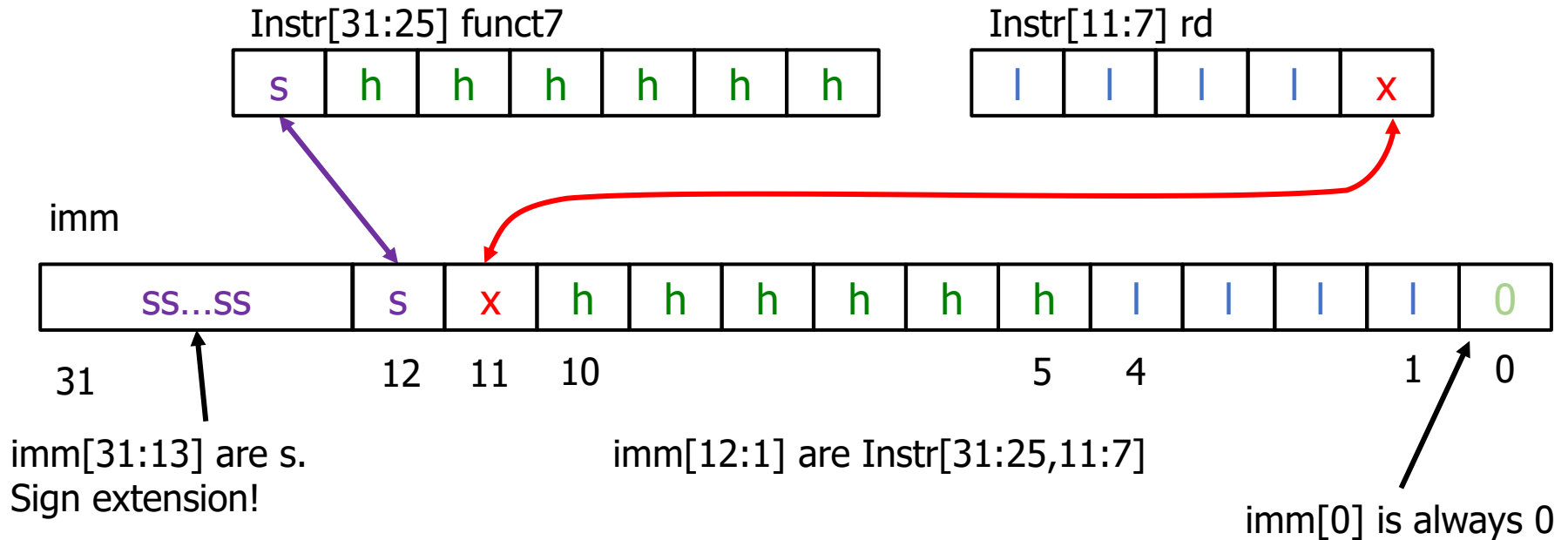
Now we have 12 bits to be placed in the machine code.

The placement of bits is unusual ! So a new type of format.

Placement of immediate bits in SB type

Instr is the 32-bit machine code.

imm == offset



RISC-V SB-type Instructions

imm[]	rs2	rs1	funct3	imm[]	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

branch-instruction rs1, rs2, L

- Fields in SB-type
 - opcode: operation code
 - rs1: first source register number
 - rs2: second source register number
 - immediate fields: funct7 and rd

Store 12 bits from the offset

Similar to S-type, but the placement of the bits is **unusual**

When a processor executes branches, it reconstructs a 32-bit immediate (which is the same as the offset) from the 12 bits in the immediate fields

Opcode and funct3 of branch instructions

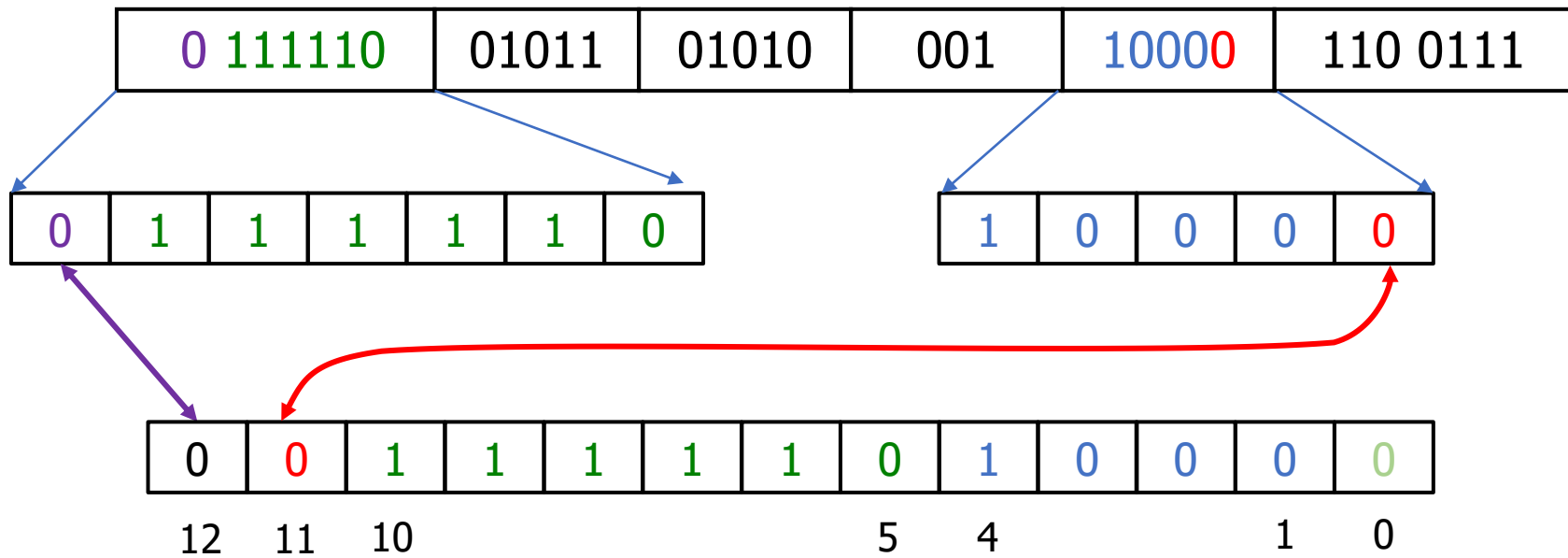
All branches have the same opcode.

beq	SB	1100011	000
bne	SB	1100011	001
blt	SB	1100011	100
bge	SB	1100011	101
bltu	SB	1100011	110
bgeu	SB	1100011	111

SB-type Example: BNE

imm[]	rs2	rs1	funct3	imm[]	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

```
bne x10, x11, PC+2000
```



2000 = 0b00..000 0111 1101 0000

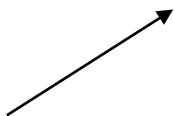
Discussions

- Effectively, the offset is a 13-bit 2's complement number
 - 12 bits from the machine code, plus a 0
- Can go to any even addresses in range:

from $PC - 2^{12}$ to $PC + 2^{12} - 2$

- Since the target is PC-relative, branches work correctly even if the code block is placed at a different address

instruction words
w/o C extension



Out of reach
I1023
...
I2
I1
I0: Branch
I_1
I_2
...
I_1024
Out of reach

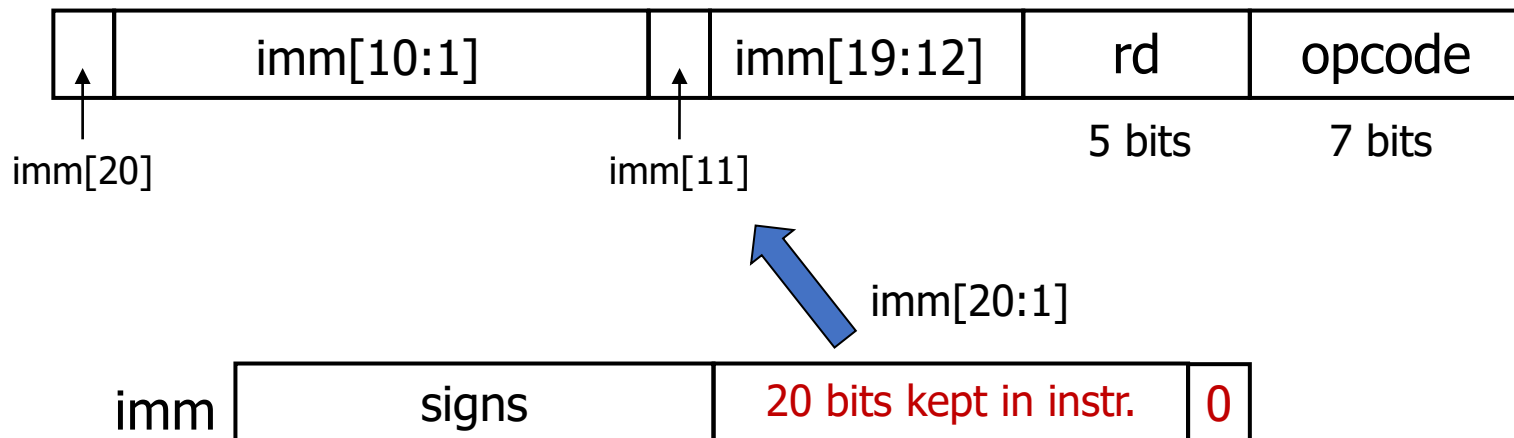
Limitation on branches

- It is fine for if-else and loop
 - A loop should not have 2000 instructions!
- We need to jump farther for function calls though
- U-type has a 20-bit immediate field!

JAL: UJ-type

JAL rd, Label

- JAL is based on U-type format, which has 20 bits for immediate
 - **imm[20:1]** are kept in the machine code, **imm[0]** is always 0
 - Can jump to any even addresses in $[PC - 2^{20}, PC + 2^{20} - 2]$
- The placement of bits are **unusual**
 - So it is a new format, called **UJ**
 - Can you figure out how the processor builds the 32-bit immediate?



Branch farther away

- If branch target is too far to encode in a branch instruction, assembler rewrites the code

Example:

```
        beq    s0, s1, L1
        ↓
        bne    s0, s1, L2
        jal    x0, L1
L2:      ...
```

Note that there is no special Jump instruction in RISC-V.
Pseudoinstruction J is just JAL with rd set to x0.

Can we go farther?

- 32-bit processors: two instructions can go to any location
 - Again, the lower 12 bits are sign extended in JALR
 - Add 1 to the higher 20 bits if the lower 12 bits are negative

```
lui    t0, HI20_ADDR      # load address[31:12] in t0
jalr   x0, LO12_ADDR(t0)  # add address[11:0] to t0
```

- 64-bit processors:
 - Anywhere in lower 4 GiB, or
 - 4GiB centered around PC
 - We will learn AUIPC next time
 - What if we want to go even farther?

JALR: I-type

imm[11:0]	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits

`jalr rd, offset(rs1)`

JALR writes PC+4 to rd, sets PC = rs1 + imm

The immediate is sign extended as in ADDI
imm[0] is included!

RISC-V Encoding Summary

- There are 4-types essentially: R, I, S, and U
- SB is similar to S, but bits from imm are placed differently
- UJ is similar to U, but bits from imm are placed differently

Name (Field Size)	Field						Comments
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

How does processor construct immediate?

- Why did they make it so confusing?
 - It is hard for compiler/assembler/us
 - It is easier for hardware

Compare bits in I and J, and those in S and B.

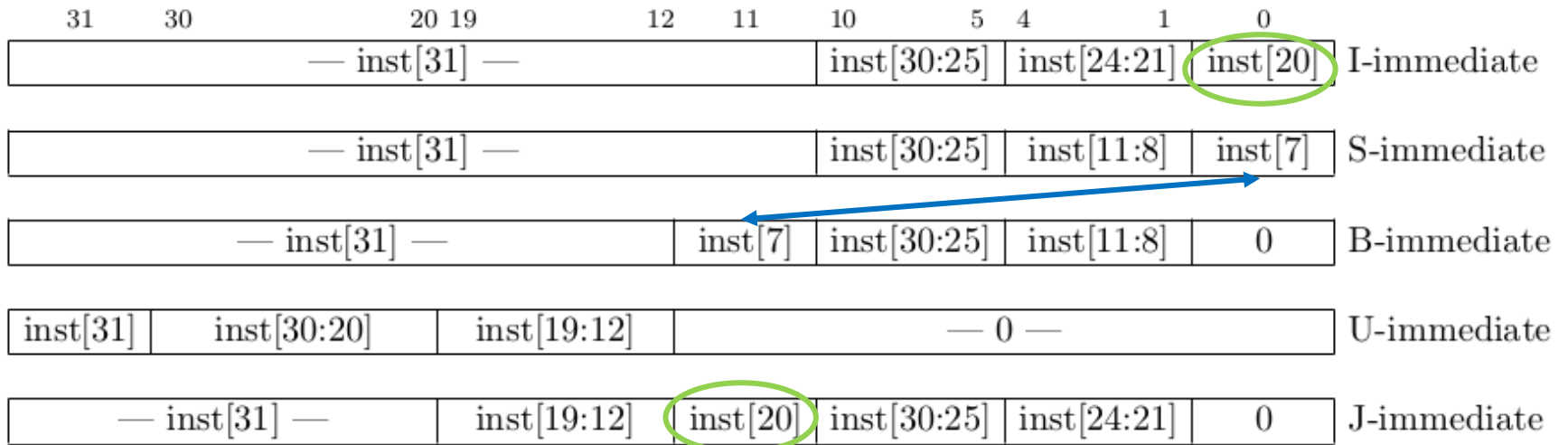
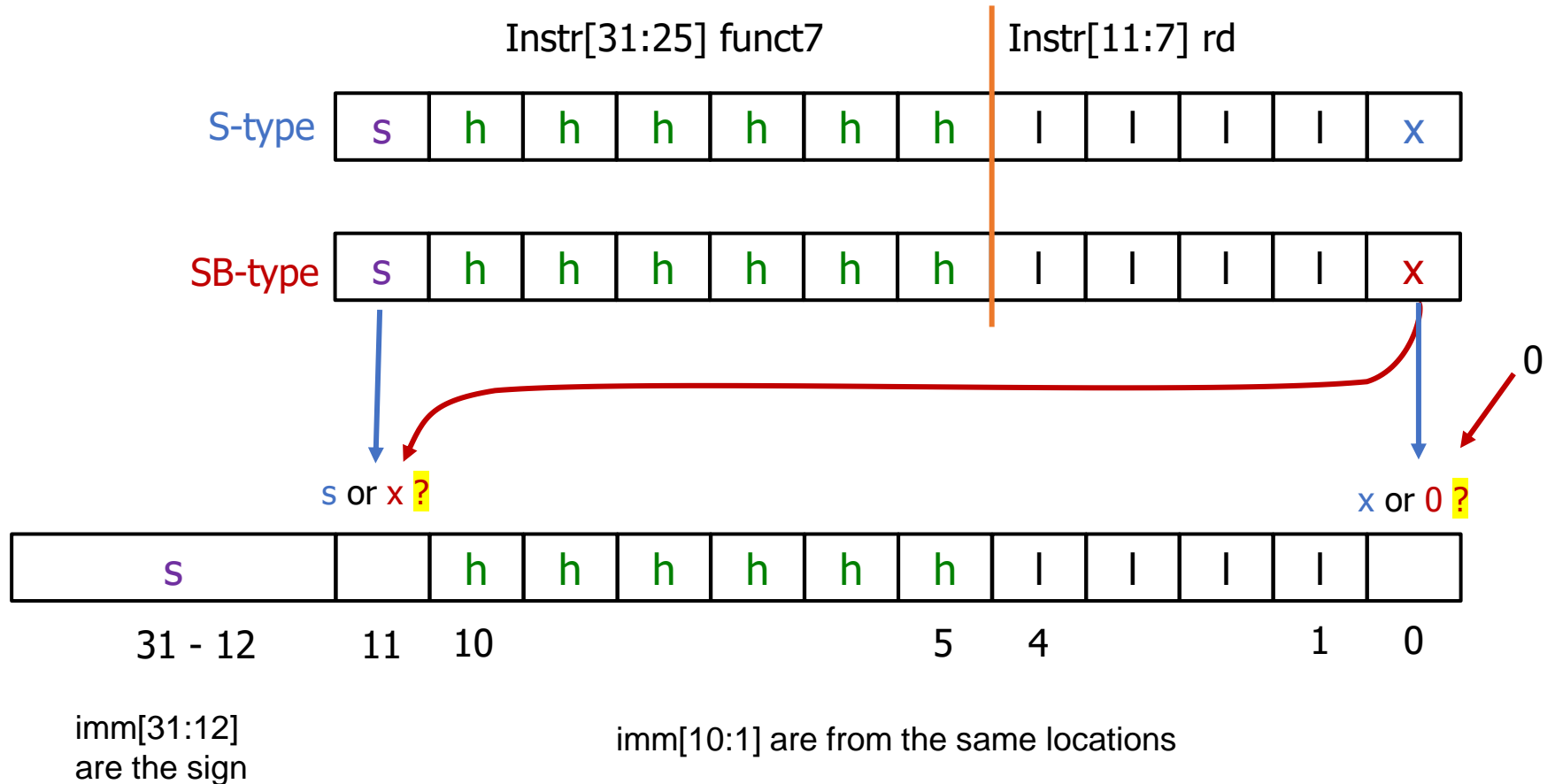


Figure 2.4: Types of immediate produced by RISC-V instructions. The fields are labeled with the instruction bits used to construct their value. Sign extension always uses inst[31].

Comparison of immediate in S and SB types



When processor constructs 32-bit imm, only imm[11] and imm[0] need special handling

Instruction Encoding Examples

The following code starts at location 80000 (**decimal**).

Fields are in hexadecimal.

				f7	rs2	rs1	f3	rd	op.
Loop:	slli	x10, x22, 2	80000	00	02	16	1	0A	13
	add	x10, x10, x25	80004	00	19	0A	0	0A	33
	lw	x9, 0(x10)	80008	00	00	0A	3	09	03
	bne	x9, x24, Exit	80012	00	18	09	1	0C	63
	addi	x22, x22, 1	80016	00	01	16	0	16	13
	beq	x0, x0, Loop	80020	7F	00	00	0	0D	63
Exit:	...		80024						

BNE: offset = 80024 - 80012 = 12

BEQ: offset = 80000 - 80020 = -20

Example: decoding BEQ

- The BEQ Instruction on the previous slide

Hex: FE0006E3

Bin: 1111 1110 0000 0000 0000 0110 1110 0011

Fields: 1111111 00000 00000 000 01101 1100011

Imm: 1111 1111 1111 1111 1111 1111111 01100



Assembler calculates branch offset and places bits in fields

Processor extracts bits from machine code and rebuilds the immediate

Study other instructions on the previous slides.

Question


- What is the immediate decoded from the following branch instruction?
- What is the target address?

0x0040 004C:



Address

0xFCB5 10E3



Machine code

Answer

0xFCB5 10E3

= 0b 1111 1100 1011 0101 0001 0000 1110 0011

= 0b 1111110 01011 01010 001 00001 1100011

imm[15:0]: 1111 1111 1100 0000 -64



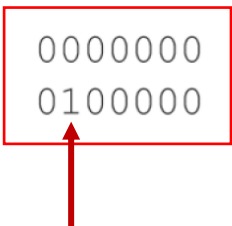
Target address is:

0x0040_004C + 0xFFFF_FFC0 = 0x0040_000C

Design Question

- Why is Instr[30] picked to differentiate SRLI and SRAI?
 - Why not Instr[31]?
 - Why not Instr[25] (the lowest bit in funct7)?

srli	I	0010011	101	0000000	funct7
srai	I	0010011	101	0100000	



Question

- What is the offset of the BEQ instruction in the following code?
- What are the bits in funct7 and rd in the machine code?
- How is the BEQ instruction encoded in the machine code?

I1: NOP

I2: NOP

I3: NOP

...

I10: NOP

I11: BEQ 0, 0, I1