Digital Design with Verilog

RISC-V : Instruction Formats

Lecture 28 : RISC-V Part 3

Disclaimer

I do not own all the slides which I am going to present. The content is mostly from

- GIAN course titled "**Next-Generation Semiconductors: RISC-V, AI, TL-Verilog**" by Steeve Hoover. https://github.com/silicon-vlsi/gian-course-2024-IITBBS
- Introduction to Computer Architecture by Hasan Baig, https://www.hasanbaig.com
  - Most of the slides are copied/directly from his course content, please visit his site for reference.
- Digital Design and Computer Architecture, RISC-V Edition by Sarah L. Harris and David Money Harris.
- Other online publicly available sources. I tried my best to acknowledge the source wherever possible. I too might have missed some sources too ☺

How to represent Instructions ?

- Everything in computer is represented with binary and so the **Instructions** as well
  - Called machine code
- How many bits should we use to encode instructions?
- Are we using the same number of bits to encode all instructions?
  - Do all instructions have the same length?

```
add         rd, rs1, rs2
addi        rd, rs1, immd
lw          rd, offset(rs1)
sw          rs2, offset(rs1)
beq         rs1, rs2, LABEL
```

How to represent Instructions ?

Design Principle 4*:* Good design demands good compromises

Keep formats as similar as possible

instruction $\boxed{\text{32 bits}}$

- RISC-V instructions
  - Encoded as 32-bit instruction words
  - Small number of formats, encoding operation code (opcode), register numbers, …
  - Regularity!

RISC-V Instruction Format

- The layout of bits in instruction words is instruction format
  - How do we use 32-bit bits to specify operation code (opcode), registers, immediate, offset, etc? How many bits for each?

- RISC-V has six instruction formats
  - R, I, S, SB, U, and UJ

R-Format

R-Format

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

- For instructions that have 3 registers as operands
- Fields in R-type

• opcode:	7-bit operation code
• rd:	destination register number
• rs1:	first source register number
• rs2:	second source register number
• funct3:	additional function code
• funct7:	even more function code

R-Format

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

```
a = b + c

add x9,x20,x21
```

0	21	20	0	9	51

0000000	10101	10100	000	01001	0110011

$0000000\_10101\_10100\_000\_01001\_0110011_2$

$0000\ 0001\ 0101\ 1010\ 0000\ 0100\ 1011\ 0011_2$

$015A04B3_{16}$

R-Format

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

Any instruction like:

```
instruction     rd, rs1, rs2
```

can be represented with R-Format.

R-Format: Knowledge Check

	Funct7
ADD	0b 000 0000
SUB	0b 010 0000

- What RISC-V instruction does this represent? Choose from one of the four options below.
  - The numbers are in decimal

funct7	rs2	rs1	funct3	rd	opcode
32	9	10	0	11	51

```
A.  sub    x9,   x10,   x11
B.  add    x11,   x9,   x10
C.  sub    x11,   x10,   x9
D.  sub    x11, x9, x10
```

R-Format

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

	type	opcode	funct3	funct7
add	R	0110011	000	0000000
sub	R	0110011	000	0100000
sll	R	0110011	001	0000000
slt	R	0110011	010	0000000
sltu	R	0110011	011	0000000
xor	R	0110011	100	0000000
srl	R	0110011	101	0000000
sra	R	0110011	101	0100000
or	R	0110011	110	0000000
and	R	0110011	111	0000000

I-Format

I-Format

immediate	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits

- Immediate arithmetic and load instructions
  - rs1: source or base address register number
  - immediate: constant operand, or offset added to base address
    - 2’s-complement, sign extended

- Since only 12 bits are kept in machine code, the immediate must be in $[-2^{11}, +2^{11} - 1]$ or $[-2048, 2047]$

I-Format

immediate	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits

```
addi x1, x2, 32
```

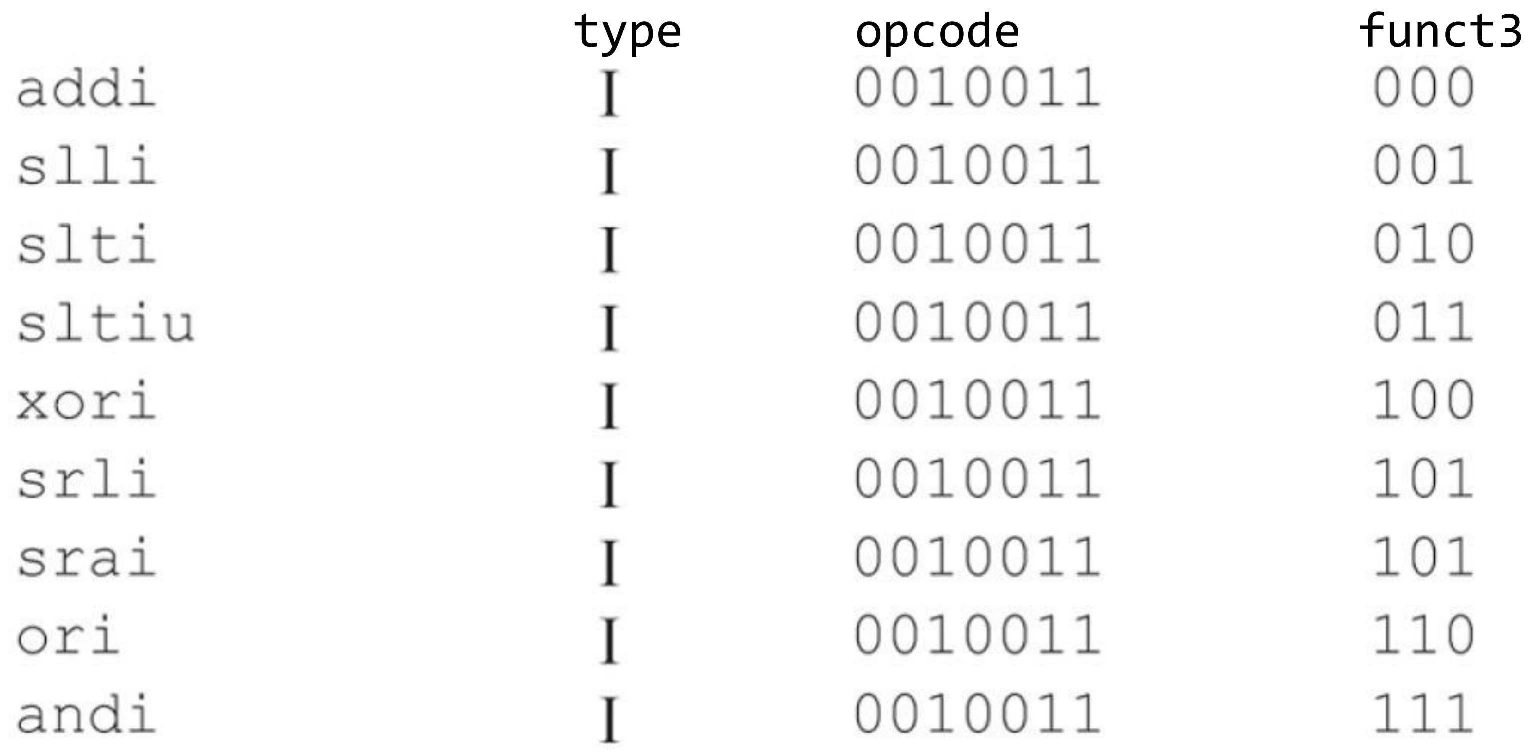
0000 0010 0000	00010	000	00001	0010011

When executing the instruction, processor builds a 32-bit immediate imm by extending the sign bit

	imm[31:12] sign	imm[11:0]
imm	0000 0000 0000 0000 0000	0000 0010 0000

I-Format

immediate	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits



	type	opcode	funct3
addi	I	0010011	000
slli	I	0010011	001
slti	I	0010011	010
sltiu	I	0010011	011
xori	I	0010011	100
srli	I	0010011	101
srai	I	0010011	101
ori	I	0010011	110
andi	I	0010011	111

In slli, srli, and srai, only lower 5 bits of the immediate are used for shift amount. 5 bits are enough for 32-bit registers!

I-Format Instructions: srli and srai

immediate	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits

```
srli x1, x2, 16
```

More than 31 times shift is not needed, therefore the maximum immediate value, i.e. 31, can be represented in 5 bits.

Remaining 7 bits can be repurposed as an additional funct7 field

funct7	imm	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

I-Format Instructions: srli and srai

funct7	imm	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

`srli x1, x2, 16`

0000 000	1 0000	00010	101	00001	0010011

`srai x1, x2, 16`

0100 000	1 0000	00010	101	00001	0010011

srli	I	0010011	101	0000000
srai	I	0010011	101	0100000

funct7

I-Format Instructions: load

- Load instructions

```
    lw  rd, offset(rs1)
```

A. R-format

B. I-format

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

I-Format Instructions: load

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

```
lw  x1, 32(x2)
```

0000 0010 0000	00010	010	00001	000 0011

	type	opcode	funct3
lb	I	0000011	000
lh	I	0000011	001
lw	I	0000011	010
ld	I	0000011	011
lbu	I	0000011	100
lhu	I	0000011	101
lwu	I	0000011	110

Do you see the pattern in funct3?

How about store?

RISC-V Instruction Formats

- Store instructions have rs2, but not rd

```
sw  rs2,offset(rs1)
```

Can any of the following two formats be used for store instructions?

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

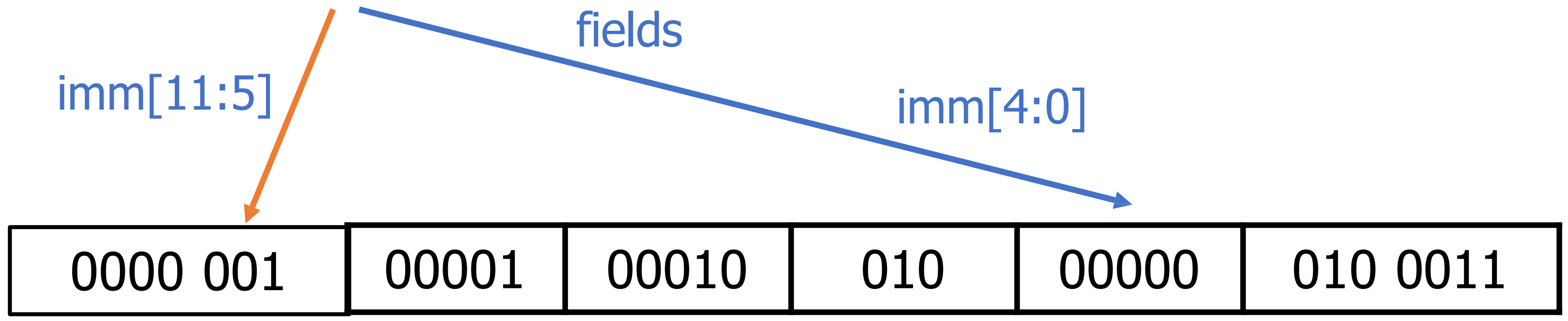
S-Format

S-Format

imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

- Different immediate format for store instructions
  - rs1: base address register number
  - rs2: source operand register number
  - immediate: offset added to base address
    - Split so that rs1 and rs2 fields always remain in the same place

`sw x1, 32(x2)` The 12 bits of the immediate are saved in two fields



R/I/S-type Instructions Summary

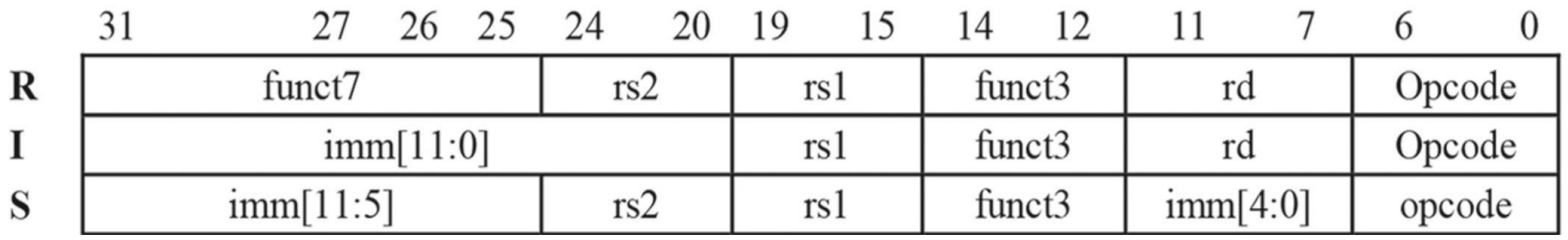
Instruction	Format	funct7	rs2	rs1	funct3	rd	opcode
`add` (add)	R	0000000	reg	reg	000	reg	0110011
`sub` (sub)	R	0100000	reg	reg	000	reg	0110011
		7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

Instruction	Format	immediate	rs1	funct3	rd	opcode
`addi` (add immediate)	I	constant	reg	000	reg	0010011
`lw` (load word)	I	address	reg	010	reg	0000011
		12 bits	5 bits	3 bits	5 bits	7 bits

Instruction	Format	immed -iate	rs2	rs1	funct3	immed -iate	opcode
`sw` (store word)	S	address	reg	reg	010	address	0100011
		7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

U-Format

RISC-V 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

U-Format

U-Format

- Recall 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

U-Format

imm [31:12]	rd	opcode
20 bits	5 bits	7 bits

- 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**

Example:

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

U-Format

```
# the immediate written in assembly language
lui   x1, 0xABCDE
```

The immediate field in the machine code

1010 1011 1100 1101 1110	00001	0110111
20 bits	5 bits	7 bits

The 32-bit immediate the processor uses is 0xABCDE000

imm	1010 1011 1100 1101 1110 0000 0000 0000

SB-Format

RISC-C Instruction Format

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
U:	imm [31:12]				rd	opcode

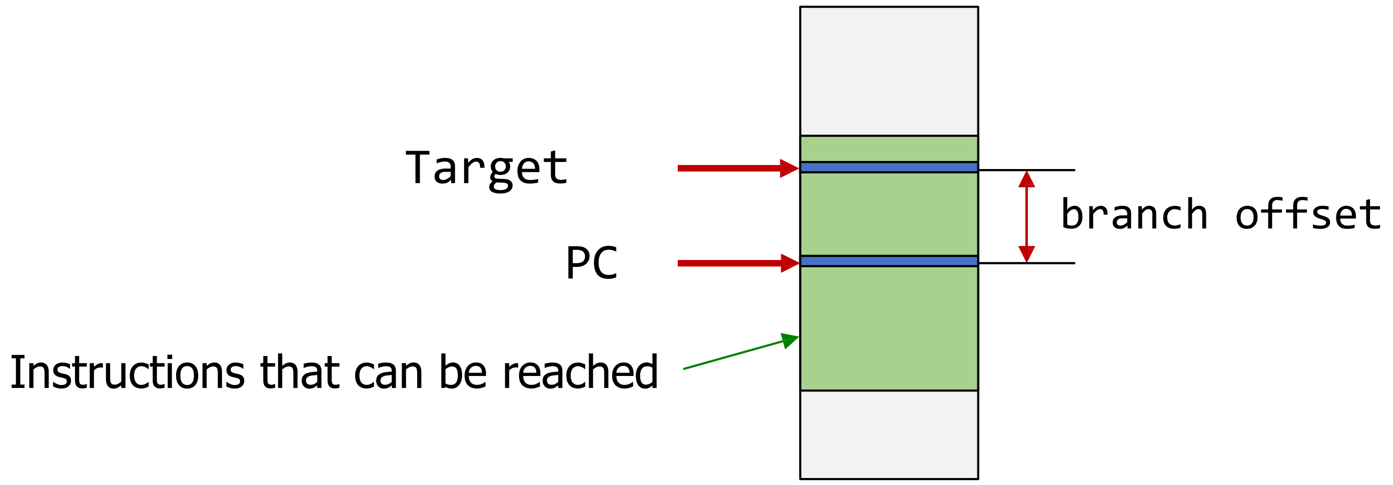
beq rs1, rs2, label

All branches are similar.

- Which format is the best for branches?
- What problem do we have?

Offset in Branches

- The full instruction address is 32 bits
  - 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)



Offset in Branches

- When encoding branches, assembler calculates the offset:

branch offset = target address – PC

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

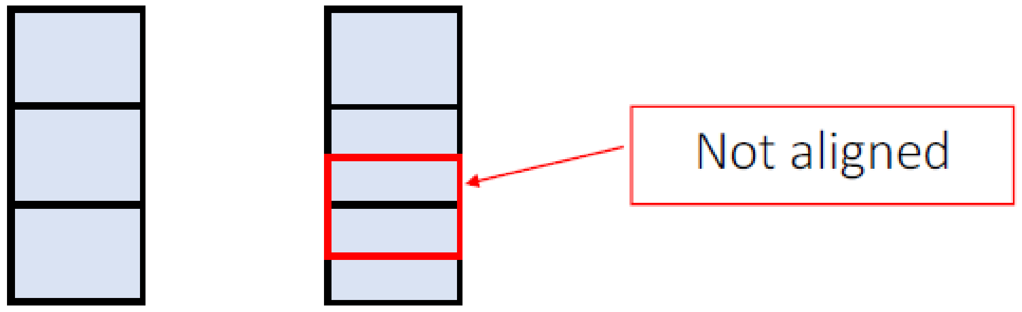
- When processor executes the instruction, it
  - Forms a 32-bit immediate from the 12 bits in machine code
  - Calculates the target address as

target address = PC + imm

imm == branch offset

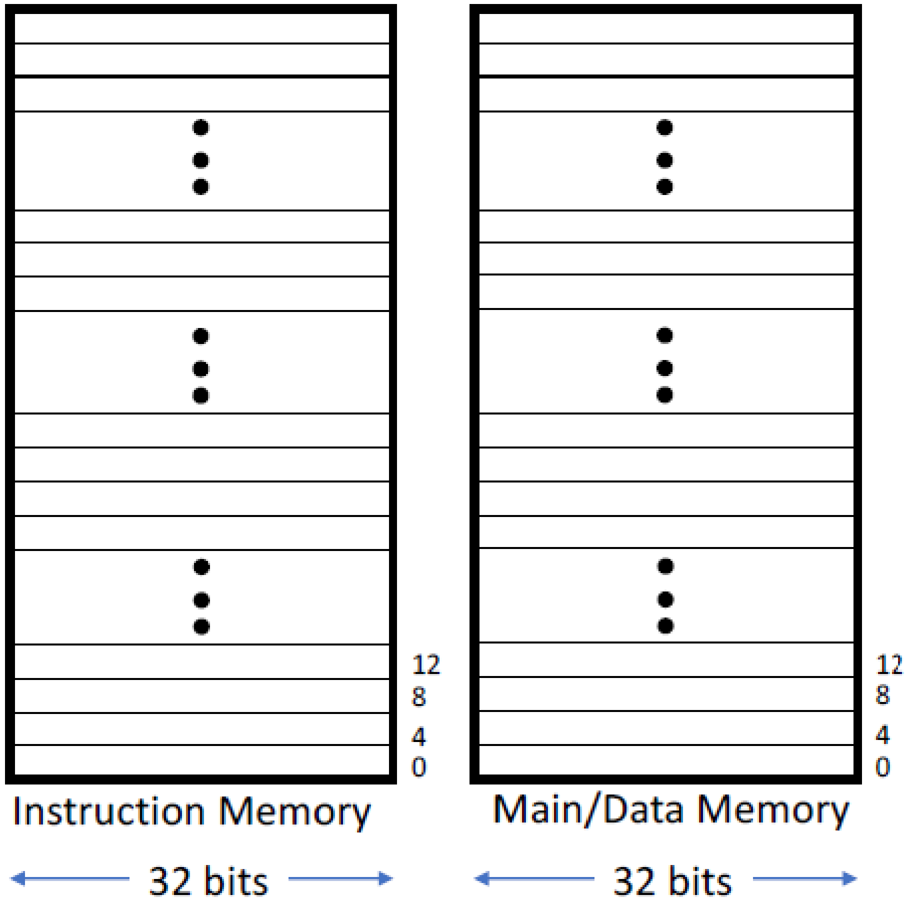
Recalling Address Alignment

- 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

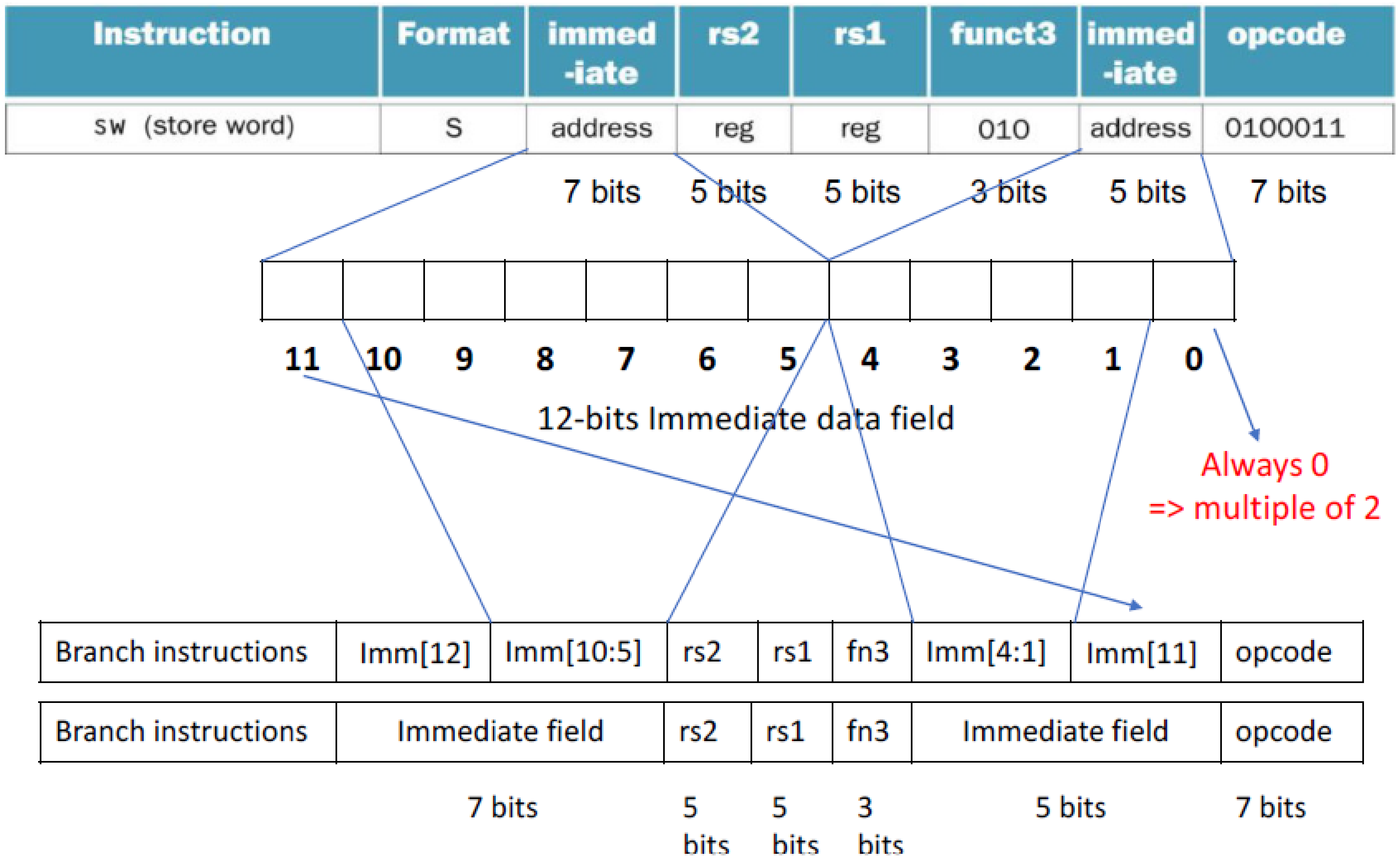


Offset Address Boundary

- Each instruction is 32 bits wide
  - → offset of 4 bytes
- Sometimes the instructions are 16-bits wide (in C extension)
  - → Can be located at any even addresses
  - → offset of 2 bytes
- RISC-V keeps the ability of branching to any even addresses
- Because of the insignificance of right most bit (i.e., always zero), the Immediate data field can be represented in **13 bits**
- Instead of -2048 to 2046, we can now access -4096 to 4094 in multiples of 2



SB-Format



Instruction	Format	immed -iate	rs2	rs1	funct3	immed -iate	opcode
sw (store word)	S	address	reg	reg	010	address	0100011
		7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

Branch instructions	Imm[12]	Imm[10:5]	rs2	rs1	fn3	Imm[4:1]	Imm[11]	opcode

Branch instructions	Immediate field	rs2	rs1	fn3	Immediate field	opcode
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

SB-Format

bne x10, x11, 2000 // if x10 != x11, goto location address 2000

2000 = 0 0111 1101 0000

0	111 110	01011	01010	001	1000	0	1100011
Imm[12]	Imm[10:5]	rs2	rs1	fn3	Imm[4:1]	Imm[11]	opcode

Branch ranges for 13 bits = -2^{12} to $2^{12}-1$ = -4096 to 4094

Represent -4096 offset in the immediate format of branch instruction

RISC-V Instruction Summary

Simple subset that shows most aspects

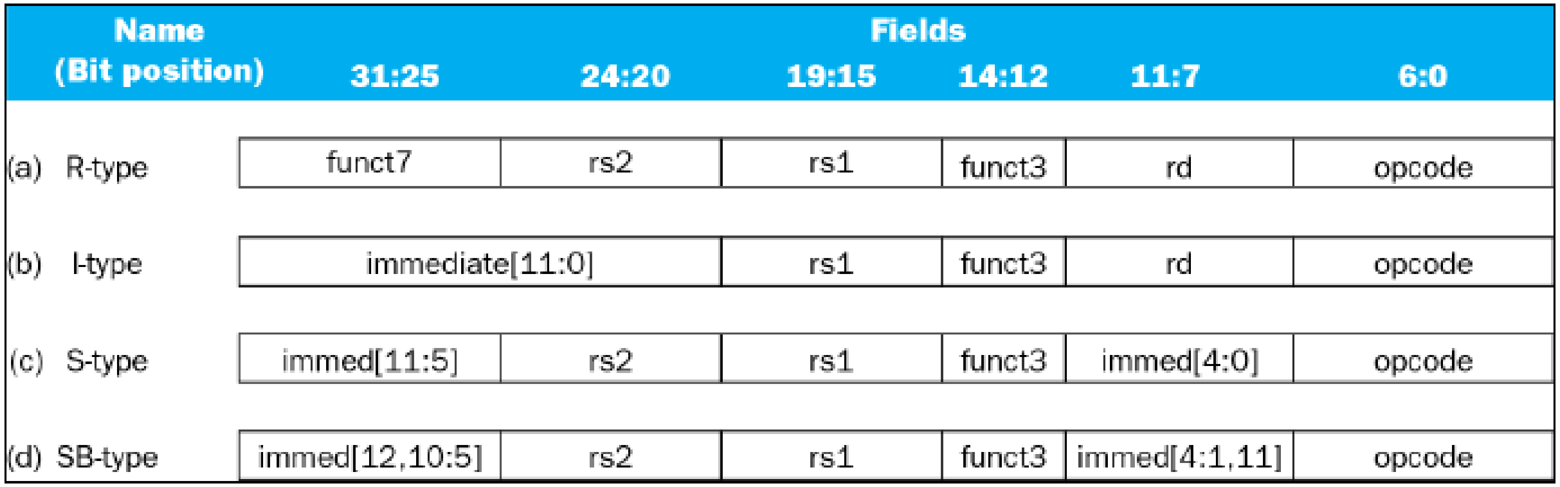
- Arithmetic/logical: add, sub, and, or
- Memory reference: lw, sw
- Branch: beq

Type	Instruction	Opcode	Funct3	Funct7
R-Type	add	011 0011	000	000 0000
R-Type	sub	011 0011	000	010 0000
R-Type	and	011 0011	111	000 0000
R-Type	or	011 0011	110	000 0000
I-Type	lw	000 0011		
S-Type	sw	010 0011		
SB-Type	beq	110 0011		

RISC-V Instruction Summary

Fields are at the same location in all encoding formats

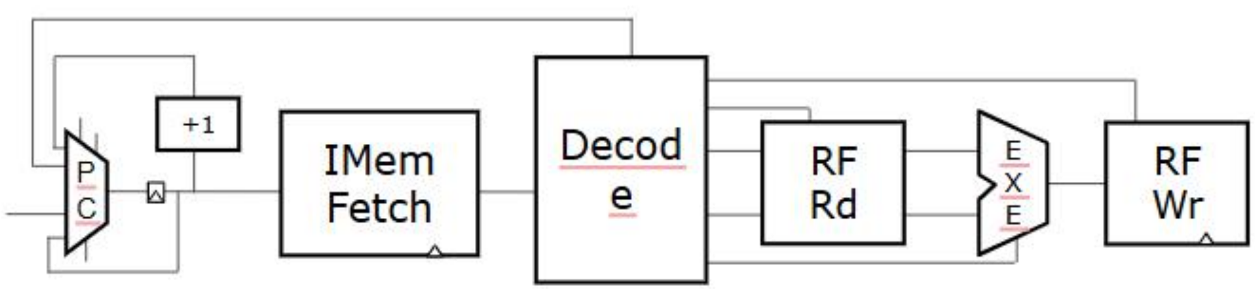
opcode, funct3, rs1, rs2, rd



Name	Fields					
(Bit position)	31:25	24:20	19:15	14:12	11:7	6:0
(a) R-type	funct7	rs2	rs1	funct3	rd	opcode
(b) I-type	immediate[11:0]		rs1	funct3	rd	opcode
(c) S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode
(d) SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode

How are the instructions executed?

Human CPU



1. Step through the execution of the program, one instruction at a time.
   - **Next PC:** Adjust PC to the next instruction (sequential or branch target) (initially to 0)
   - **Fetch**: Identify the binary instruction at the PC
   - **Decode**: Flip the Instruction Memory card to find the decoded (assembly) instruction
   - **RF Read**: "Get" the source register values from the register file
   - **Execute**: Determine the result of the operation
   - **RF Write**: Update (in pencil) the value in the destination register
   - Repeat
2. What does the program do?
3. Clean up.

Thank you