CSC 3210 Computer organization and programming

Chunlan Gao

Review Lecture 9 - Lecture 10

Conditional: bne, beq, bge, blt (check the value and choose the other instruction depend on the result), are used for if….else, while loop, switch

Unconditional: jal, jalr are used for jump to a label and return.

Procedure: A stored subroutine that performs a specific task

based on the **parameters** with which it is provided.

Procedure call:

- jal x1, ProcedureLabel

Procedure return:

jump and link register:

- jalr x0, 0(x1)

Leaf-procedure

```
int leaf_example (int i, int j) {
                                        int  f = A[i – j];
                                                return f;  }
```

- Arguments i, j in x10, x11
- f in x20
- use temporaries x5, x6
- array A initial address save in x12
- Need to save x5, x6 on stack

Leaf-Procedure

```
leaf_example:

  addi sp, sp, -8 # Allocate 8 bytes stack space
  sw x5, 4(sp) # Save x5 to stack (offset 8)
  sw x6, 0(sp) # Save x6 to stack (offset 4)

  sub x5, x10, x11 # x5= i-j
  slli x5,x5, 2
  add x6, x12, x5
  lw    x20, 0(x6)

lw x6, 0(sp) # Restore x6 from stack
 lw x5, 4(sp) # Restore x5 from stack
 addi sp, sp, 8 # Free stack space
```

Byte/Halfword/Word Operations

- RISC-V byte/halfword/word load/store
  - Load byte/halfword/word: Sign extend to 64 bits in rd
    - `lb rd, offset(rs1)`
    - `lh rd, offset(rs1)`
    - `lw rd, offset(rs1)`
  - Load byte/halfword/word unsigned: Zero extend to 64 bits in rd
    - `lbu rd, offset(rs1)`
    - `lhu rd, offset(rs1)`
    - `lwu rd, offset(rs1)`
  - Store byte/halfword/word: Store rightmost 8/16/32 bits
    - `sb rs2, offset(rs1)`
    - `sh rs2, offset(rs1)`
    - `sw rs2, offset(rs1)`

32-bit Constants

- Most constants are small
  - 12-bit immediate is sufficient

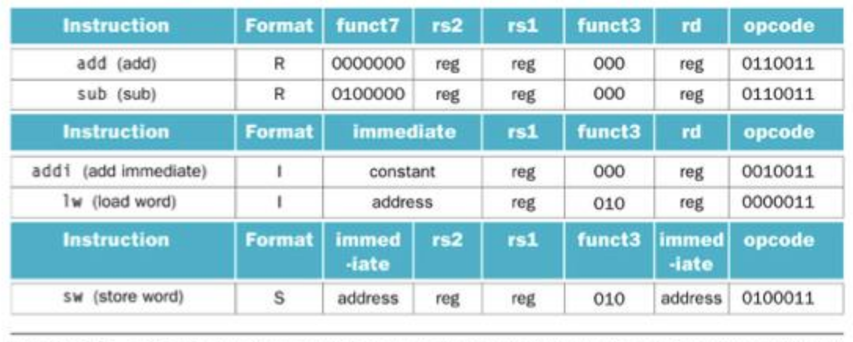
- For the occasional 32-bit constant

`lui rd, constant （Load upper immediate）`
  - Copies 20-bit constant to bits [31:12] of rd

- Clears bits [11:0] of rd to 0

Loading 32-niy constant into register X19(32-bits),initially 0x 00000000,
the constant is 0000 0000 0011 1101 0000 0101 0000 0000two

0000 0000 0011 1101 0000	0000 0000 0000
lui x19, 976	
0000 0000 0011 1101 0000	0101 0000 0000
addi x19, x19, 1280	



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

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

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

FIGURE 2.5 RISC-V instruction encoding. In the table above, "reg" means a register number between 0 and 31 and "address" means a 12-bit address or constant. The funct3 and funct7 fields act as additional opcode fields.

Hardware Software Interface

Key Points

1. **Immediate Field Size Restriction**
    1. In RISC-V, the **immediate field size** in instruction formats (such as **I-type, S-type**) is limited.
    2. For example, **I-type instructions (e.g., addi)** can only hold a **12-bit immediate (-2048 to 2047)**.
    3. If a larger constant (e.g., 0x12345678) is needed, it **cannot fit directly into a single instruction**.

   Breaking and Reassembling Large Constants
    1. The **compiler or assembler must split large constants into multiple parts** and **reassemble them into registers** using multiple instructions.
    2. In RISC-V, this is commonly done using the lui (**Load Upper Immediate**) and addi (**Add Immediate**) combination: lui x5, 0x12345 # Load upper 20 bits (0x12345000) addi x5, x5, 0x678 # Load lower 12 bits and add to x5
    3. As a result, x5 contains 0x12345678.

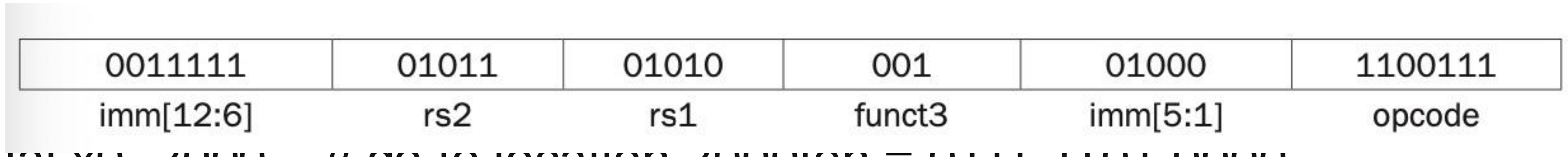
1. **Memory Addressing Faces Similar Limitations**
    1. **Load (lw) and store (sw) instructions** also face the **12-bit immediate limit**, meaning memory addresses often require **two-step computation**: lui x6, 0x10000 # Load upper 20 bits of the address lw x7, 0x200(x6) # Use lower 12-bit offset to load data
    2. This allows RISC-V to **access full 32/64-bit memory addresses while keeping instruction formats simple**.

2. **The Assembler Extends the Instruction Set**
    1. The **hardware itself does not support loading a full 32-bit constant in one instruction**, but the **assembler provides a more user-friendly symbolic representation**.
    2. For example, instead of manually using lui and addi, an assembler **pseudo-instruction** simplifies the process: li x5, 0x12345678 # Pseudo-instruction
    3. This li (Load Immediate) **is not a real RISC-V instruction**—instead, the **assembler translates it** into multiple actual instructions (lui + addi).

- **Hardware constraints limit instruction formats**, restricting immediate values and memory addressing.
- **The assembler enhances usability** by **providing pseudo-instructions** that translate into multiple valid RISC-V instructions.
- **Understanding computer architecture requires focusing on the raw hardware-supported instruction set**, but **assembly programming benefits from the assembler's enhancements** for improved readability and efficiency.

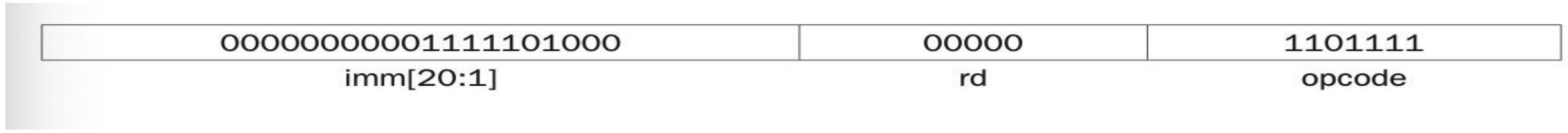
Branch and Jump Addressing

- bne x10, x11, 2000 // if x10 != x11, go to location 2000ten = 0111 1101 0000（absolute address）



0011111	01011	01010	001	01000	1100111
imm[12:6]	rs2	rs1	funct3	imm[5:1]	opcode

- jal x0, 2000 // go to location 2000ten = 0111 1101 0000



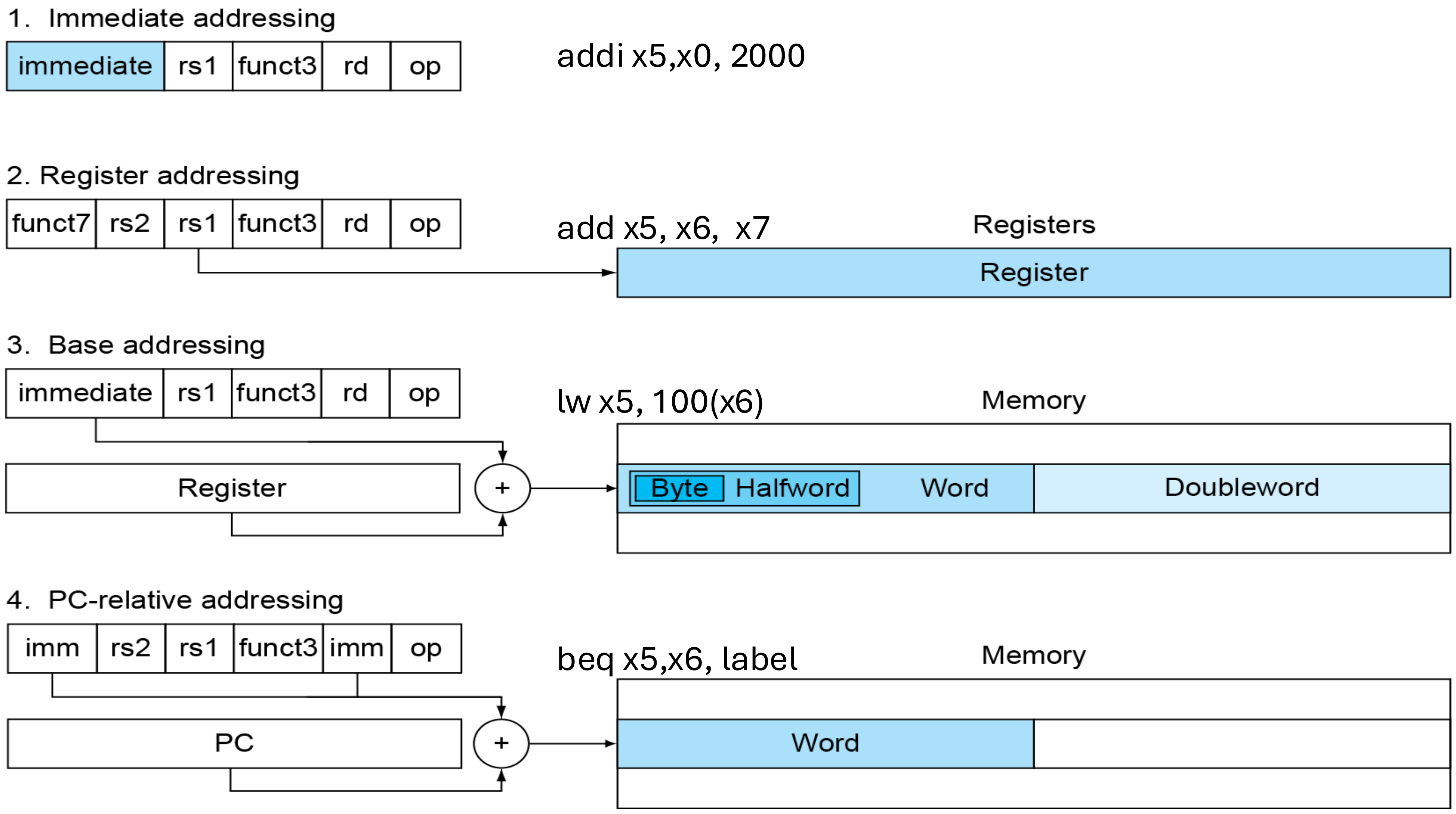
00000000001111101000	00000	1101111
imm[20:1]	rd	opcode

If addresses of the program had to fit in this 20-bit or 12-bit field, it would mean that no program could be bigger than 2^{20}, which is far too small to be a realistic option today. So, we use relative address

Program counter= Register + Branch offset

- Since the **Program Counter (PC)** holds the address of the current instruction, we can:
- **Branch within ±2^{10} words** of the current instruction.
- **Jump within ±2^{18} words** of the current instruction.
- For long jumps, eg, to 32-bit absolute address
  - lui: load address[31:12] to temp register
  - jalr: add address[11:0] and jump to target

RISC-V Addressing Summary



addi x5,x0, 2000

add x5, x6, x7

lw x5, 100(x6)

beq x5,x6, label

RISC-V Encoding Summary

RISC-V Encoding refers to the process of converting **assembly instructions into machine code** in the RISC-V architecture.

Core Concepts of RISC-V Encoding

1. **Instruction Format**
   RISC-V follows a **fixed 32-bit instruction length** (with extensions supporting 16-bit and 64-bit instructions). Each instruction is broken down into several fields:
   1. **Opcode**: Defines the instruction category, e.g., LOAD, STORE, ALU, BRANCH, etc.
   2. **rd (Destination Register)**: Specifies the destination register.
   3. **funct3 (Function Code 3)**: Differentiates instruction variants, such as ADD vs. SUB.
   4. **rs1, rs2 (Source Registers)**: Hold operands for operations.
   5. **funct7 (Function Code 7)**: Further distinguishes operations like ADD and SUB.
   6. **Immediate**: An embedded value in the instruction.

RISC-V Encoding Summary

2. Instruction Formats (Types) RISC-V classifies instructions into the following formats:

- **R-Type (Register-to-Register Operations)**
  Example: ADD rd, rs1, rs2
- **I-Type (Immediate & Load Instructions)**
  Example: ADDI rd, rs1, imm
- **S-Type (Store Instructions)**
  Example: SW rs2, imm(rs1)
- **B-Type (Branch Instructions)**
  Example: BEQ rs1, rs2, offset
- **U-Type (Upper Immediate Load Instructions)**
  Example: LUI rd, imm
- **J-Type (Jump Instructions)**
  Example: JAL rd, offset

Format	Instruction	Opcode	Funct3	Funct6/7
R-type	add	0110011	000	0000000
	sub	0110011	000	0100000
	sll	0110011	001	0000000
	xor	0110011	100	0000000
	srl	0110011	101	0000000
	sra	0110011	101	0000000
	or	0110011	110	0000000
	and	0110011	111	0000000
	lr.d	0110011	011	0001000
	sc.d	0110011	011	0001100
I-type	lb	0000011	000	n.a.
	lh	0000011	001	n.a.
	lw	0000011	010	n.a.
	lbu	0000011	100	n.a.
	lhu	0000011	101	n.a.
	addi	0010011	000	n.a.
	slli	0010011	001	000000
	xori	0010011	100	n.a.
	srli	0010011	101	000000
	srai	0010011	101	010000
	ori	0010011	110	n.a.
	andi	0010011	111	n.a.
	jalr	1100111	000	n.a.
S-type	sb	0100011	000	n.a.
	sh	0100011	001	n.a.
	sw	0100011	010	n.a.
SB-type	beq	1100111	000	n.a.
	bne	1100111	001	n.a.
	blt	1100111	100	n.a.
	bge	1100111	101	n.a.
	bltu	1100111	110	n.a.
	bgeu	1100111	111	n.a.
U-type	lui	0110111	n.a.	n.a.
UJ-type	jal	1101111	n.a.	n.a.

FIGURE 2.18 RISC-V instruction encoding. All instructions have an opcode field, and all formats

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

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

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

FIGURE 2.5 RISC-V instruction encoding. In the table above, "reg" means a register number between 0 and 31 and "address" means a 12-bit address or constant. The funct3 and funct7 fields act as additional opcode fields.

Example:

addi x5, x0, 10 # x5 = 0 + 10

opcode = 0010011 (I-Type instruction)

rd   = 00101   (Destination register x5)

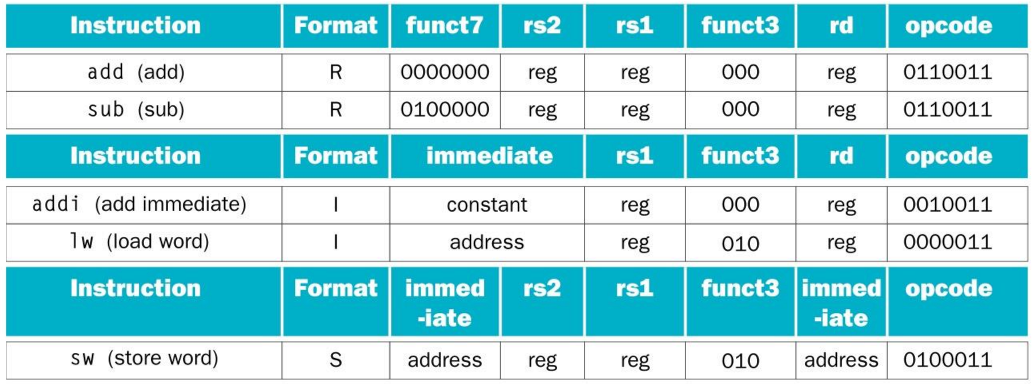
funct3 = 000   (ADDI instruction)

rs1   = 00000   (Source register x0)

imm   = 000000001010 (Immediate value 10)

- Binary:000000001010 00000 000 00101 0010011
- The final **32-bit machine code** in **hexadecimal:**

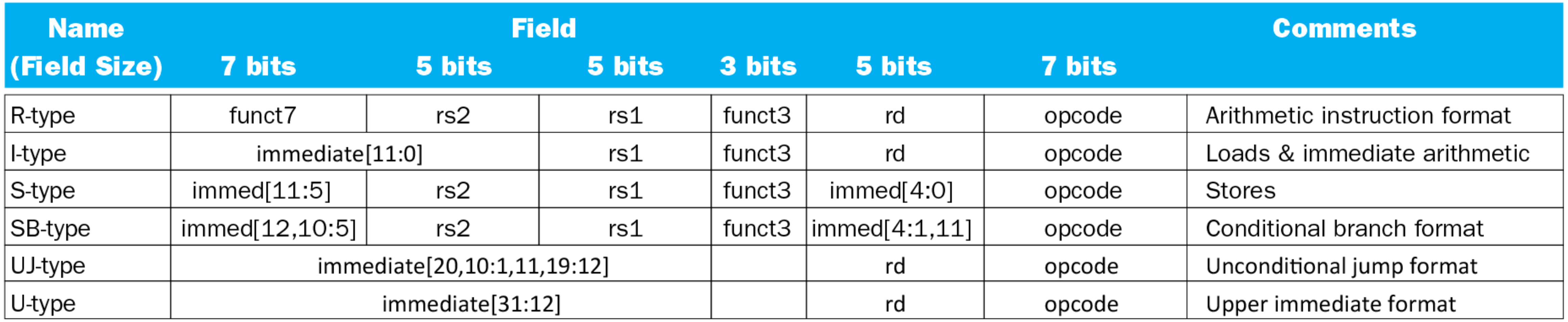
0x00A00293



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

FIGURE 2.5 RISC-V instruction encoding. In the table above, "reg" means a register number between 0 and 31 and "address" means a 12-bit address or constant. The funct3 and funct7 fields act as additional opcode fields.

RISC-V Encoding Summary



Name	Field						Comments
(Field Size)	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

Example for decoding

- What is the assembly language statement corresponding to this machine instruction?

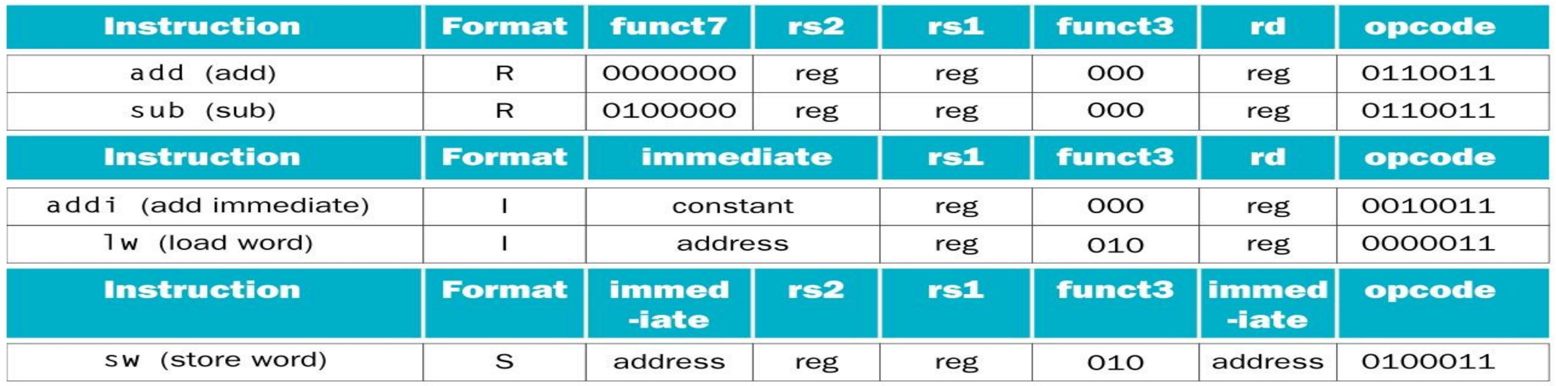
00578833hex

- The first step is converting hexadecimal to binary:

opcode

funct7

0000 0000 0101 0111 1000 1000 0011 0011



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
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
Instruction	**Format**	**immed-iate**	**rs2**	**rs1**	**funct3**	**immed-iate**	**opcode**
sw (store word)	S	address	reg	reg	010	address	0100011

FIGURE 2.5 RISC-V instruction encoding. In the table above, “reg” means a register number between 0 and 31 and “address” means a 12-bit address or constant. The funct3 and funct7 fields act as additional opcode fields.

We have other scenarios

- **Multi-Threading within a Single Process**

Voice and video calls require multiple threads to handle different data streams, such as:

- **One thread for audio capture (Microphone Input).**
- **One thread for audio playback (Speaker Output).**
- **One thread for video stream processing (Camera Input & Display).**
- **One thread for network transmission (Sending & Receiving Data).**

- **Multi-Process Synchronization**

when you open Chrome and Discord together, it involves multi-processing

- **Multi-Core Processing**

You are using a video editing software (e.g., Adobe Premiere, HandBrake, FFmpeg) to encode a large 4K video file into a smaller format (e.g., H.264 or HEVC).(Code Example: Multi-Core Video Encoding Using FFmpeg)

```
ffmpeg -i input.mp4 -c:v libx264 -preset fast -threads 8 output.mp4
```

-threads 8 → Uses **8 CPU cores** to encode the video in parallel.

- **Distributed Computing & Cloud Systems:**

When you search for something on Google, millions of users are sending search queries at the same time. A single server cannot handle this enormous load, so Google uses distributed computing across thousands of machines.

Synchronization in RISC-V

- Load reserved: `lr.w rd,(rs1)`
  - Load from address in rs1 to rd
  - Place reservation on memory address
- Store conditional: `sc.w rd,(rs1),rs2`
  - Store from rs2 to address in rs1
  - Succeeds if location not changed since the `lr.w`
    - Returns 0 in rd
  - Fails if location is changed
    - Returns non-zero value in rd

Synchronization in RISC-V

- Example 1: atomic swap (to test/set lock variable)

```
again:  lr.w x10,(x20)
        sc.w x11,(x20),x23 // X11 = status
        bne  x11,x0,again  // branch if store failed
        addi x23,x10,0     // X23 = loaded value
```

- Example 2: lock

```
        addi x12,x0,1          // copy locked value
again:  lr.w x10,(x20)         // read lock
        bne  x10,x0,again      // check if it is 0 yet
        sc.w x11,(x20),x12     // attempt to store
        bne  x11,x0,again      // branch if fails
```

- Unlock:

```
        sd   x0,0(x20)         // free lock
```