CSC 3210 Computer organization and programming

Final-Review

Chunlan Gao

Attendance number

- 9865

Final

Look at the study guide

Final_Exam_Study_Guide_11am

Chapter 2: Instructions – Language of the Computer

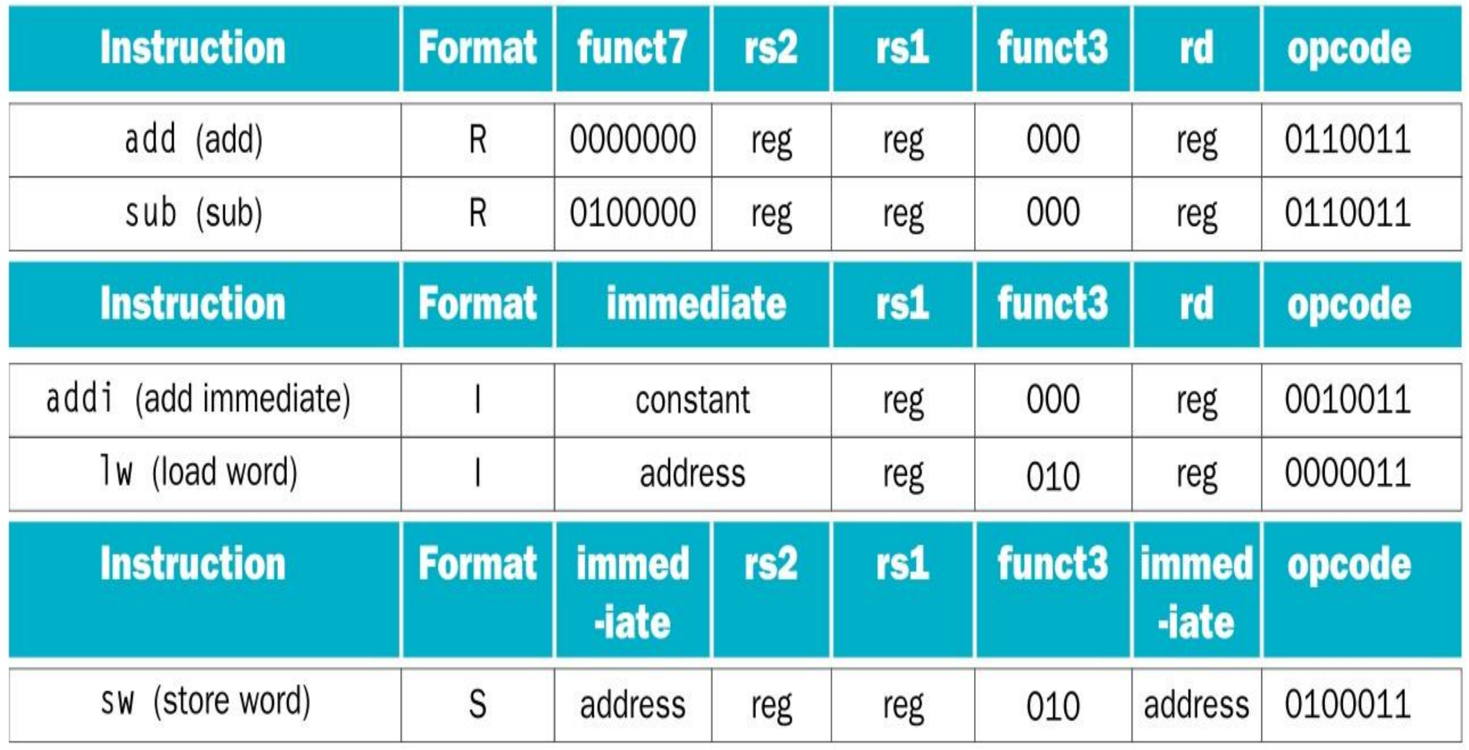
2.5 Representing Instructions:

- RISC-V instruction formats
- Instruction binary encoding

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.

Example



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.

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 = 0000000000001010 (Immediate value 10)
Binary:0000000000001010 00000 000 00101 0010011
The final **32-bit machine code** in **hexadecimal**:
0x00A00293

2.7 Making Decisions

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)

2.8 Supporting Procedures in Hardware:

- jal, jalr instructions and stack usage

A **leaf procedure** is a function that **does not call any other functions**

```
leaf_proc:
   add a0, a0, a1      # Some computation
   jalr x0, 0(ra)      # Return to caller (no need to save ra)
```

A **non-leaf procedure calls other functions**,

```
non_leaf_proc:
   addi sp, sp, -16    # Allocate stack space
   sw ra, 12(sp)       # Save return address before call

   jal ra, helper_func  # Call another function

   lw ra, 12(sp)       # Restore return address
   addi sp, sp, 16     # Deallocate stack space
   jalr x0, 0(ra)      # Return to caller
```

Synchronization in RISC-V

- Load reserved: `lr.w rd,(rs1)`
  - Load value 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

Translating Programs:

1. **Compilation (High-Level to Assembly)**
    1. The **C source code** (x.c) is first **compiled** into an **assembly language program** (x.s).
    2. Example: int main() { return 42; } Compiles to: _main: movl $42, %eax ret
2. **Assembly (Assembly to Machine Code)**
    1. The **assembler** converts the **assembly language file** (x.s) into an **object module** (x.o) containing **machine language**.
    2. This object file is **not yet executable** because it may have **unresolved references** (e.g., calling printf() but not defining it).
3. **Linking (Combining Multiple Object Files)**
    1. The **linker** takes multiple x.o object files and combines them into an executable.
    2. It also **links external libraries** (e.g., printf() from libc.a or libc.so).
    3. The output is usually an **executable file** (a.out in UNIX, .exe in Windows).
4. **Loading (Placing Code in Memory for Execution)**
    1. The **loader** loads the executable file into **memory** and prepares it for execution.
    2. It may also perform **dynamic linking** if the program depends on shared libraries (x.so in UNIX, .DLL in Windows).

Appendix A: Basics of Logic Design

- Gates and Truth Tables, and Boolean Algebra(+, ▪)

-ROMs

64X16 ROM, How many address linesare needed?

- Sequential (state)Logic blocks: Register file,
- Combinational Logic blocks: multiplexers, adders

Chapter 3: Arithmetic for Computers

3.1–3.2 Addition and Subtraction:

- - 2’s complement, overflow detection

For **n-bit signed integers**, the maximum is:

$$2^{n-1} - 1$$

- 8-bit maximum: 0111 1111 –> 127

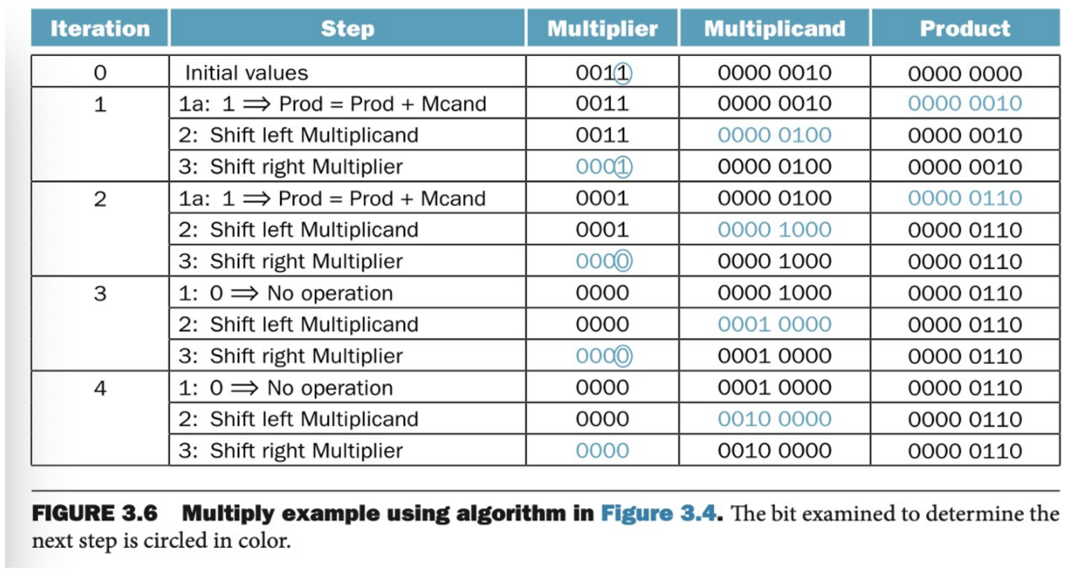
  minimum： 1111 1111-> -128

- addition: A>0, B>0, A+B

3.3 Multiplication:

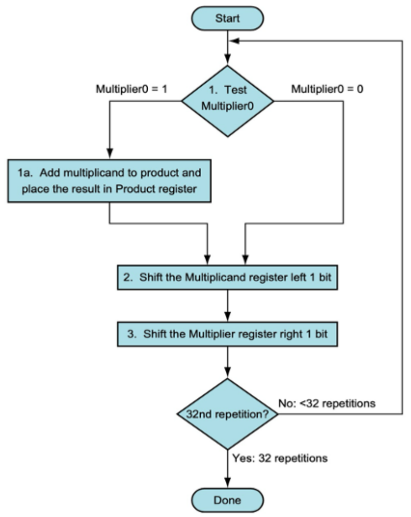
multiplication algorithm

Using 4-bit numbers to save space, multiply 2ten × 3ten, or 0010two × 0011two

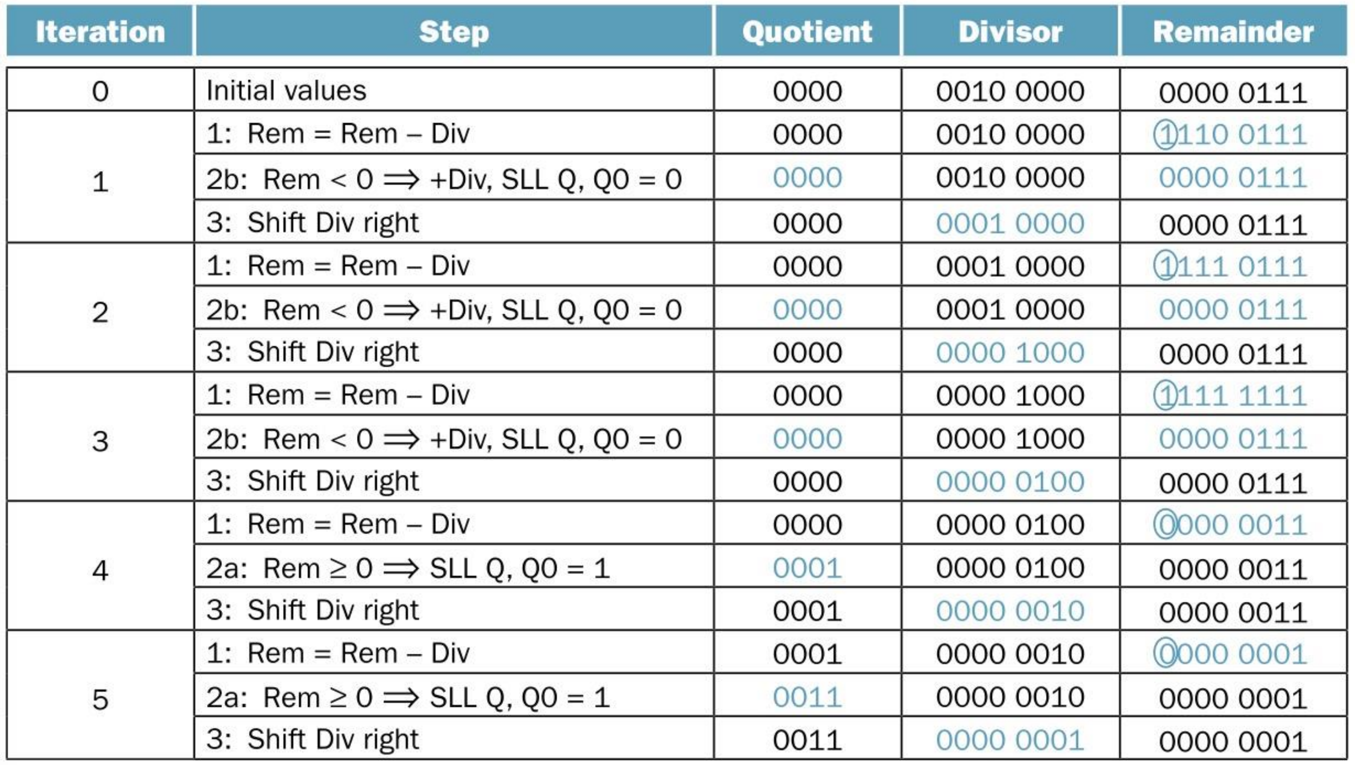


Iteration	Step	Multiplier	Multiplicand	Product
0	Initial values	0011	0000 0010	0000 0000
1	1a: 1 ⟹ Prod = Prod + Mcand	0011	0000 0010	0000 0010
	2: Shift left Multiplicand	0011	0000 0100	0000 0010
	3: Shift right Multiplier	0001	0000 0100	0000 0010
2	1a: 1 ⟹ Prod = Prod + Mcand	0001	0000 0100	0000 0110
	2: Shift left Multiplicand	0001	0000 1000	0000 0110
	3: Shift right Multiplier	0000	0000 1000	0000 0110
3	1: 0 ⟹ No operation	0000	0000 1000	0000 0110
	2: Shift left Multiplicand	0000	0001 0000	0000 0110
	3: Shift right Multiplier	0000	0001 0000	0000 0110
4	1: 0 ⟹ No operation	0000	0001 0000	0000 0110
	2: Shift left Multiplicand	0000	0010 0000	0000 0110
	3: Shift right Multiplier	0000	0010 0000	0000 0110

FIGURE 3.6 Multiply example using algorithm in Figure 3.4. The bit examined to determine the next step is circled in color.


Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
No: <32 repetitions
Yes: 32 repetitions
Done

3.4 Division:



Iteration	Step	Quotient	Divisor	Remainder
0	Initial values	0000	0010 0000	0000 0111
1	1: Rem = Rem – Div	0000	0010 0000	①110 0111
	2b: Rem < 0 ⟹ +Div, SLL Q, Q0 = 0	0000	0010 0000	0000 0111
	3: Shift Div right	0000	0001 0000	0000 0111
2	1: Rem = Rem – Div	0000	0001 0000	①111 0111
	2b: Rem < 0 ⟹ +Div, SLL Q, Q0 = 0	0000	0001 0000	0000 0111
	3: Shift Div right	0000	0000 1000	0000 0111
3	1: Rem = Rem – Div	0000	0000 1000	①111 1111
	2b: Rem < 0 ⟹ +Div, SLL Q, Q0 = 0	0000	0000 1000	0000 0111
	3: Shift Div right	0000	0000 0100	0000 0111
4	1: Rem = Rem – Div	0000	0000 0100	⓪000 0011
	2a: Rem ≥ 0 ⟹ SLL Q, Q0 = 1	0001	0000 0100	0000 0011
	3: Shift Div right	0001	0000 0010	0000 0011
5	1: Rem = Rem – Div	0001	0000 0010	⓪000 0001
	2a: Rem ≥ 0 ⟹ SLL Q, Q0 = 1	0011	0000 0010	0000 0001
	3: Shift Div right	0011	0000 0001	0000 0001

FIGURE 3.10 Division example using the algorithm in Figure 3.9. The bit examined to determine the next step is circled in color.

3.5 Floating Point:

- - IEEE 754, normalized, special values (NaN, ±∞), Multiplication

$$(-1)^S \times (1 + \text{Fraction}) \times 2^E$$

Component	Single Precision (32-bit)	Double Precision (64-bit)	Description
Sign	1 bit	1 bit	Indicates the sign: 0 = positive, 1 = negative
Exponent	8 bits	11 bits	Biased exponent (bias = 127 for single, 1023 for double)
Fraction	23 bits	52 bits	Also called mantissa; stores the fractional part (with an implicit leading 1)
Total Size	32 bits	64 bits	Total size in memory

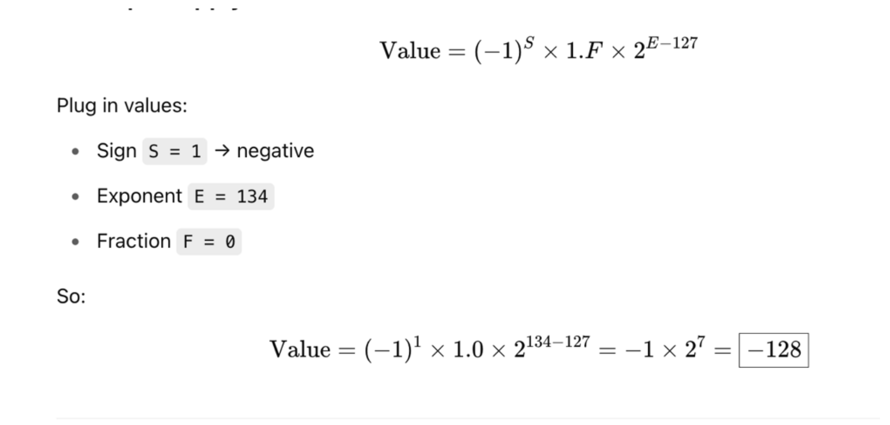
Example

- Represent –0.75
  - $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
  - S = 1
  - Fraction = $1000\ldots00_2$
  - Exponent = –1 + Bias
    - Single: $-1 + 127 = 126 = 01111110_2$
    - Double: $-1 + 1023 = 1022 = 01111111110_2$
- Single: 1011111101000…00
- Double: 1011111111101000…00

Example

- 0xC3000000 in IEEE single precision
- hex-binary
  1100 0011 0000 0000 0000 0000 0000 0000

Field	Value
Sign	1 (negative)
Exponent	10000110 = 134
Fraction (Mantissa)	00000000000000000000000



$$\text{Value} = (-1)^S \times 1.F \times 2^{E-127}$$

Plug in values:

- Sign `S = 1` → negative
- Exponent `E = 134`
- Fraction `F = 0`

So:

$$\text{Value} = (-1)^1 \times 1.0 \times 2^{134-127} = -1 \times 2^7 = \boxed{-128}$$

Example

0.75 × 0.25 using **IEEE 754 single-precision format (32-bit)** step-by-step.

Step 1: Convert Decimal to IEEE 754 Binary

1. 0.75 in IEEE 754 (32-bit)

- Decimal: 0.75
- Binary: 0.11
- Normalize: 1.1×2^{-1}
- Sign bit: 0
- Exponent: 127 - 1 = 126 → 01111110
- Mantissa: 10000000000000000000000

So:

0.75 = 0 01111110 10000000000000000000000 sign exponent fraction Hex: 0x3F400000

2. 0.25 in IEEE 754

- Decimal: 0.25
- Binary: 0.01
- Normalize: 1.0×2^{-2}
- Sign: 0
- Exponent: 127 - 2 = 125 → 01111101
- Mantissa: 00000000000000000000000

So:

0.25 = 0 01111101 00000000000000000000000 📜 Hex: 0x3E800000

Example

Step 2: Multiply in IEEE 754

To multiply two IEEE 754 floats:

Formula:

$$(-1)^{s_1} \times 1.f_1 \times 2^{e_1-127} \quad \times \quad (-1)^{s_2} \times 1.f_2 \times 2^{e_2-127}$$

Then:

$$\text{Result} = (-1)^{s_1 \oplus s_2} \times (1.f_1 \times 1.f_2) \times 2^{(e_1+e_2-127)}$$

Apply to our numbers:

- Sign: `0 ⊕ 0 = 0`
- Mantissa:

$$1.1 \times 1.0 = 1.1 = 1.5 \text{ (in decimal)}$$

- Exponent:

$$126 + 125 - 127 = 124$$

So result is:

$$\text{Value} = (+1) \times 1.5 \times 2^{124-127} = 1.5 \times 2^{-3} = \boxed{0.1875}$$

Example

Final IEEE 754 Result:

- Value: 0.1875
- Binary: 0.0011 → Normalize: 1.5×2^{-3}
- Sign = 0
- Exponent = 127 - 3 = 124 → 01111100
- Mantissa = .5 → 10000000000000000000000

$$\text{IEEE 754: } \boxed{0\ 01111100\ 10000000000000000000000}$$

Final Answer:

- **Decimal:** 0.75 × 0.25 = 0.1875
- **IEEE 754 Hex:** 0x3E400000
- Let me know if you want me to walk you through this on a visual chart or need the 64-bit double version!

Chapter 4: The Processor

4.1–4.4 Datapath and Control:

- - ALU, registers, control signals, execution cycle

5 pipeline stages:

$$\text{Instruction Memory} \rightarrow \text{Register File (Read)} \rightarrow \text{ALU} \rightarrow \text{Data Memory} \rightarrow \text{Register File (Write)}$$

A register file consists of a set of registers that can be read and written by supplying a register number to be accessed.

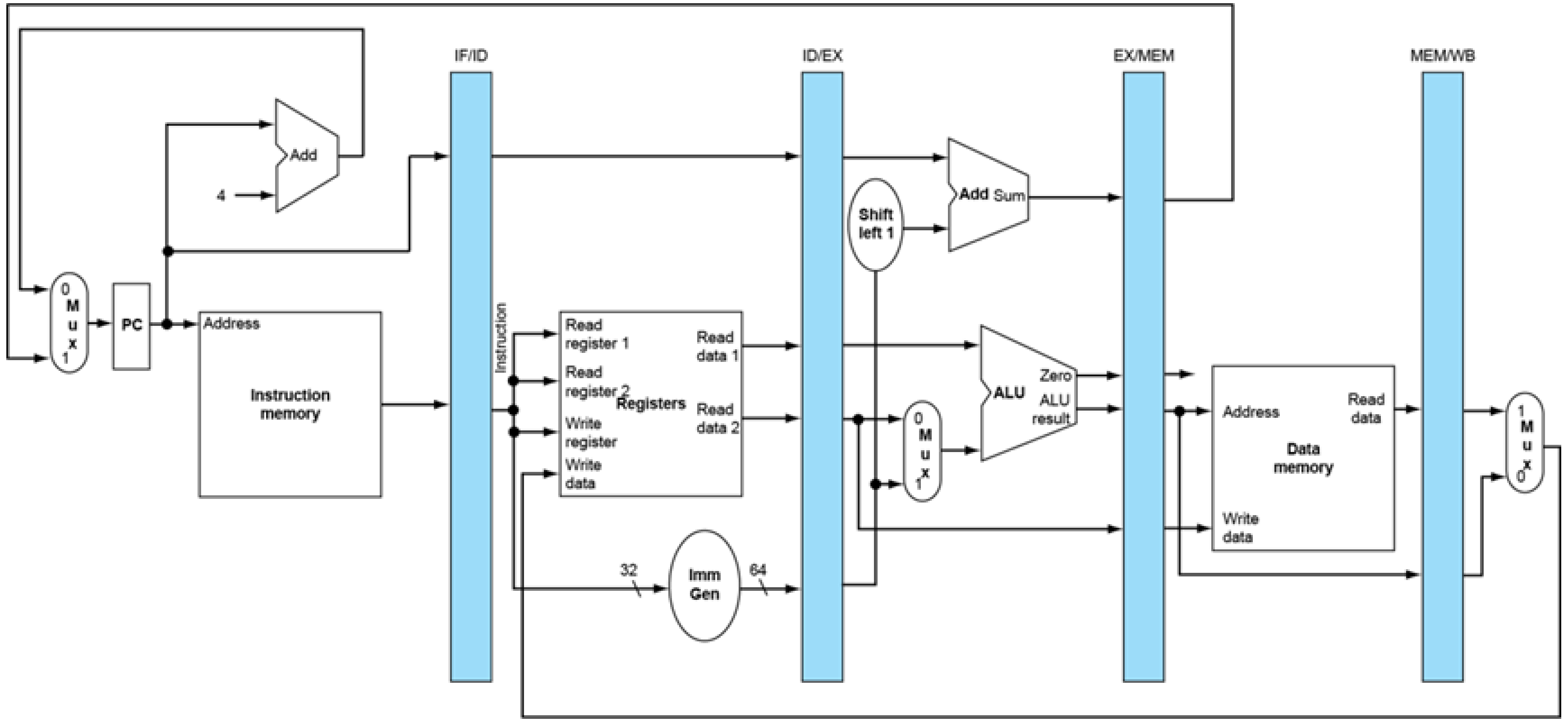
4.6–4.7 Pipelining:

- 5 stages (IF, ID, EX, MEM, WB)

1. **IF (Instruction Fetch):** Fetch the instruction from memory.
2. **ID (Instruction Decode):** Decode the instruction and read registers.
3. **EX (Execute):** Perform ALU operations or compute memory address.
4. **MEM (Memory Access):** Access data memory (for load/store instructions).
5. **WB (Write Back):** Write the result back to the destination register.

Pipeline Register

- **Purpose:**
  - Maintain **correct data flow** across stages
  - Prevent mixing of instructions
  - Enable **parallel execution** of multiple instructions



4.8 Data Hazards:

- - Forwarding, stalling, load-use

```
and   x12,x2,x5
sub   x2, x1,x3
or    x13,x6,x2
add   x14,x2,x2
sd    x15,100(x2)
```

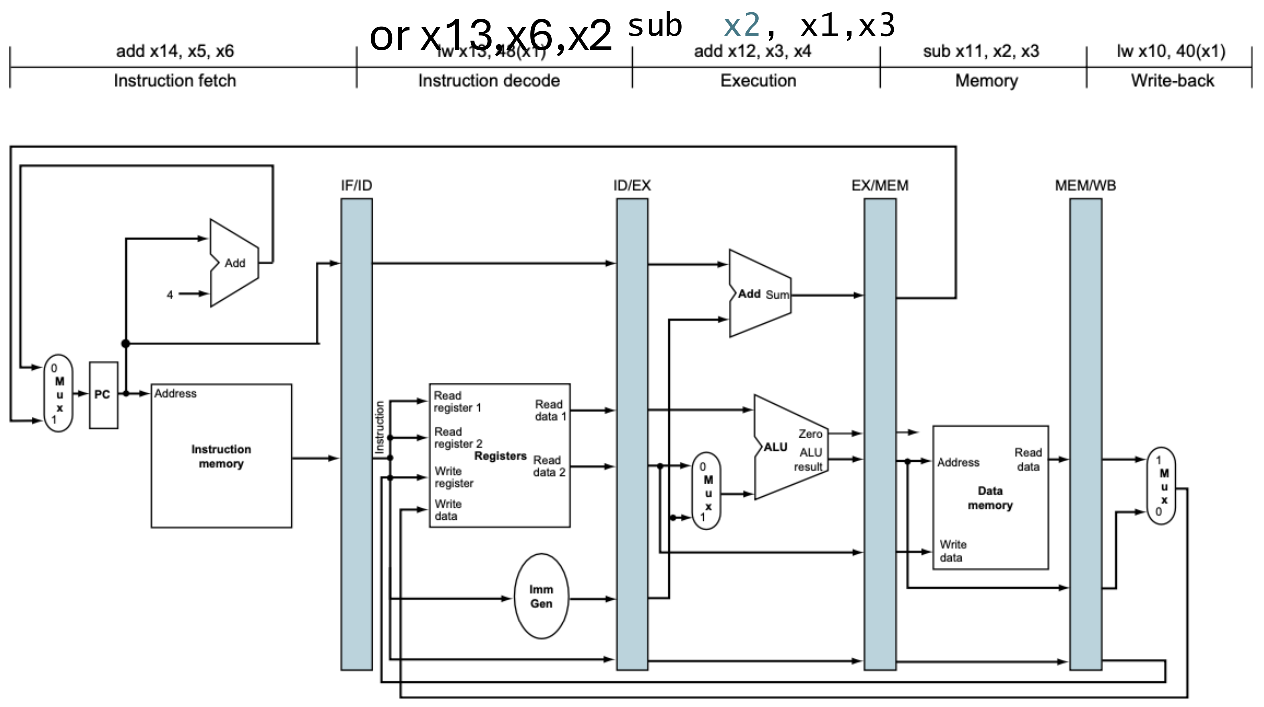


FIGURE 4.47 The single-clock-cycle diagram corresponding to clock cycle 5 of the pipeline in Figures 4.45 and 4.46. As you can see, a single-clock-cycle figure is a vertical slice through a multiple-clock-cycle diagram.

A. When the instruction (or x13, x2, x5) reaches the ID stage, does it depend on any previous instruction? Yes

B. Which instruction does it depend on? sub

C. Which register is causing the data dependency? x2

D. From which pipeline register should the result be forwarded? MEM/WB

E. Which forwarding condition (e.g., EX/MEM.RegisterRd == ID/EX.RegisterRsX) will be triggered?

MEM/WB.RegisterRd ==ID/EX.RegisterRs2 =x2

EX/MEM.RegisterRd ==ID/EX.RegisterRs2 =x2

4.10 Exceptions:

4.10 Exceptions:

- - Interrupts and exceptions

Principle of Locality

- Temporal locality
  - Items accessed recently are likely to be accessed again soon
  - e.g., instructions in a loop, induction variables
- Spatial locality
  - Items near those accessed recently are likely to be accessed soon
  - E.g., sequential instruction access, array data

Study Tips

- Review lecture slides and notes
- Practice encoding/decoding instructions
- Understand datapath and control signal flow
- Memorize pipeline stages and hazards
- Know floating-point binary format

• - Solve previous quizzes and assignments