Introduction to Computer Architecture

CPE2323: Computer Architecture and Organization
Capital University of Science and Technology, Islamabad
Department of Electrical and Computer Engineering

Course Instructor: Kamran Babar
Email: Kamran.babar@cust.edu.pk

About me

Name: Kamran Babar

Email: Kamran.babar@cust.edu.pk

Experience:

About 19 years of industrial experience in FPGAs, Embedded Systems, Signal Processing and Software Programming.

Education:

• BE Electrical Engineering (Telecom)	2007	Air University
• MS Electrical Engineering (Digital Systems and Signal Processing)	2012	NUST SEECS

Research Interests:

- FPGAs and Real time Systems
- Machine learning
- Digital Image Processing

- Name
- Interests
- Your goal in life



Course Information

- Course Code: CPE2323 – Computer Architecture and Organization
- Pre-requisite: CPE2313 – Digital Logic Design
- Credit Hours: 3
- Total Lectures: 48
- Recommended Book:
  - "Computer Organization and Design: The Hardware Software Interface (RISC-V Edition)" by Patterson & Hennessy, 2020
- Reference Book:
  - "Computer Organization and Embedded Systems" by Hamacher, Zaky, Manjikian, 6th Edition, 2011

Course Objectives

- Understand the basics of computer architecture
- Learn Assembly and Machine Language programming concepts
- Understand computer memory organization
- Develop comprehensive understanding of RISC-V microarchitecture and its ISA
- Design a single cycle RISC processor (data path and control-path)
- Learn about instruction set design
- Understand pipelining in computer architecture
- Understand memory hierarchy and cache organization

Course Learning Outcomes (CLOs)

CLO	Description	Taxonomy Level	PLO
CLO:1	Express the basic concepts of computer architecture and ISA with emphasis on RISC-V processor microarchitecture	C2	PLO:1 (Engineering Knowledge)
CLO:2	Apply concepts related to ISA, computer memory hierarchy, cache organization, and pipelining in processors	C3	PLO:1 (Engineering Knowledge)
CLO:3	Design of a single cycle RISC processor, its data path, control-path, and ISA	C5	PLO:3 (Design and Development of Solutions)

Course Discipline: Ground Rules

Classroom Etiquettes:

Students will be punctual for the class.

No attendance if you are late more than 5 mins.

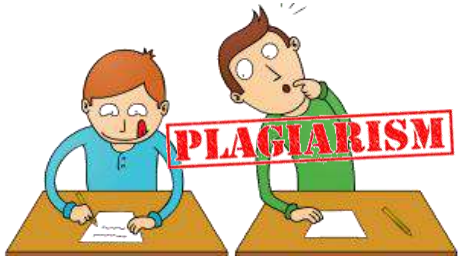
If a student decides to attend the class, he or she will not disrupt class by leaving before the lecture has ended.

All the cell phones must be switched OFF prior to entering the class room.

Academic Honesty and Plagiarism: [Both in Class and Labs]

Plagiarism is considered a serious offence by the university and severe penalties apply.

ZERO tolerance. Subject activity will be nullified.



Course Outline Overview

1. Introduction to Computer Architecture
2. Number Systems and Programming Paradigms
3. Overview of RISC-V Microarchitecture
4. RISC-V Instruction Set Architecture
5. Design of a Single Cycle Processor
6. Pipelining
7. Memory Organization and Cache Hierarchy
8. Introduction to Parallel Processing

Lecture 1 Topics

1. What is Computer Architecture?
2. Computer Organization vs. Computer Architecture
3. Components of a Computer
4. Components of a CPU
5. The Von Neumann Architecture
6. Instruction Set Architecture (ISA) – Introduction
7. Why Learn Computer Architecture?

What is Computer Architecture?

- Computer Architecture is the design and organization of a computer system, encompassing:
  - The instruction set the processor executes
  - The hardware components and their interconnections
  - The memory system organization
  - The input/output mechanisms
  - The data paths and control logic
- It defines what a computer does and how it does it at a functional level.

Computer Architecture vs. Computer Organization

Aspect	Computer Architecture	Computer Organization
Focus	What the computer does	How the computer does it
Level	Abstract / Logical	Physical / Implementation
Concerns	Instruction set, addressing modes, data types	Hardware components, timing, signals
Example	"Does it have a multiply instruction?"	"How is multiplication implemented?"
Visibility	Visible to programmer	Transparent to programmer

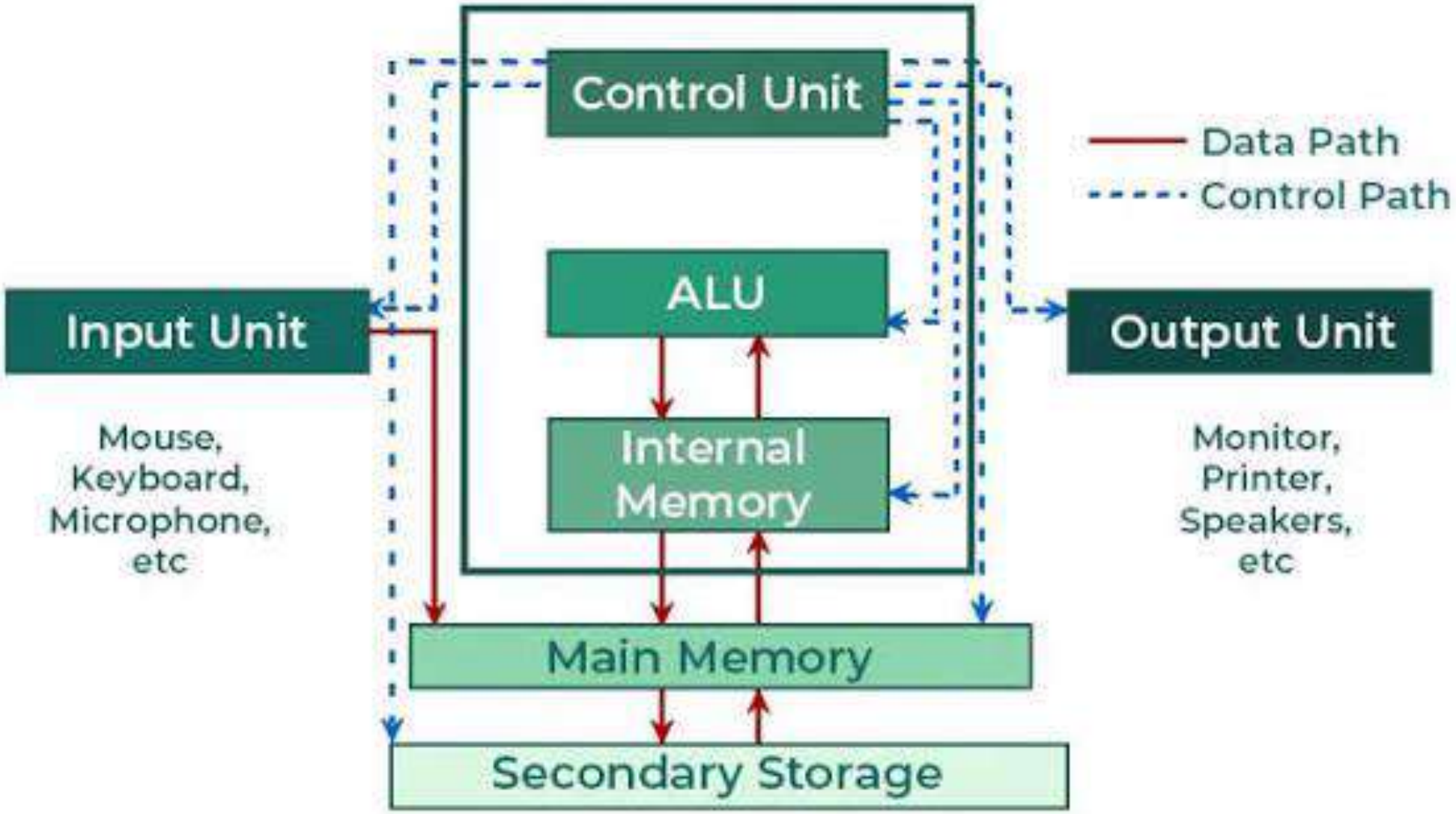
Architecture vs. Organization – Analogy

- Car Design Analogy:
  - Architecture: Does the car have automatic transmission? How many passengers can it carry?
  - Organization: What type of engine? How is the transmission built?
- In Computing:
  - Architecture: RISC-V ISA – has ADD, SUB, LW, SW instructions
  - Organization: How the ALU is built, how cache is organized, clock speed

Components of a Computer

- A modern computer consists of:
  - Processor (CPU) – Executes instructions
  - Memory – Stores data and instructions
  - Input Devices – Keyboard, mouse, sensors
  - Output Devices – Display, printer, actuators
  - Storage – Hard drives, SSDs
  - Interconnection Network – Buses, point-to-point links

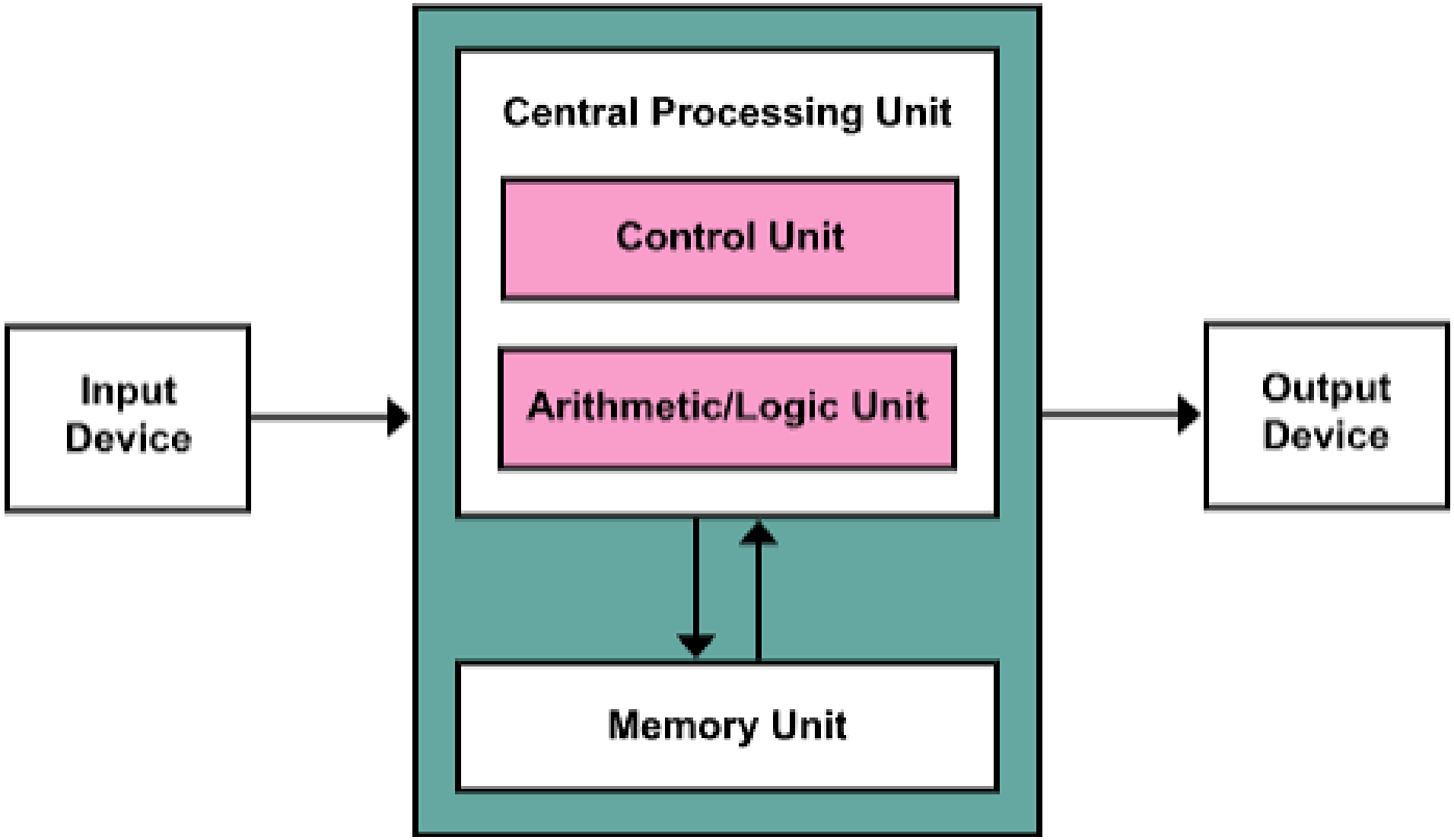
Simplified Computer Model



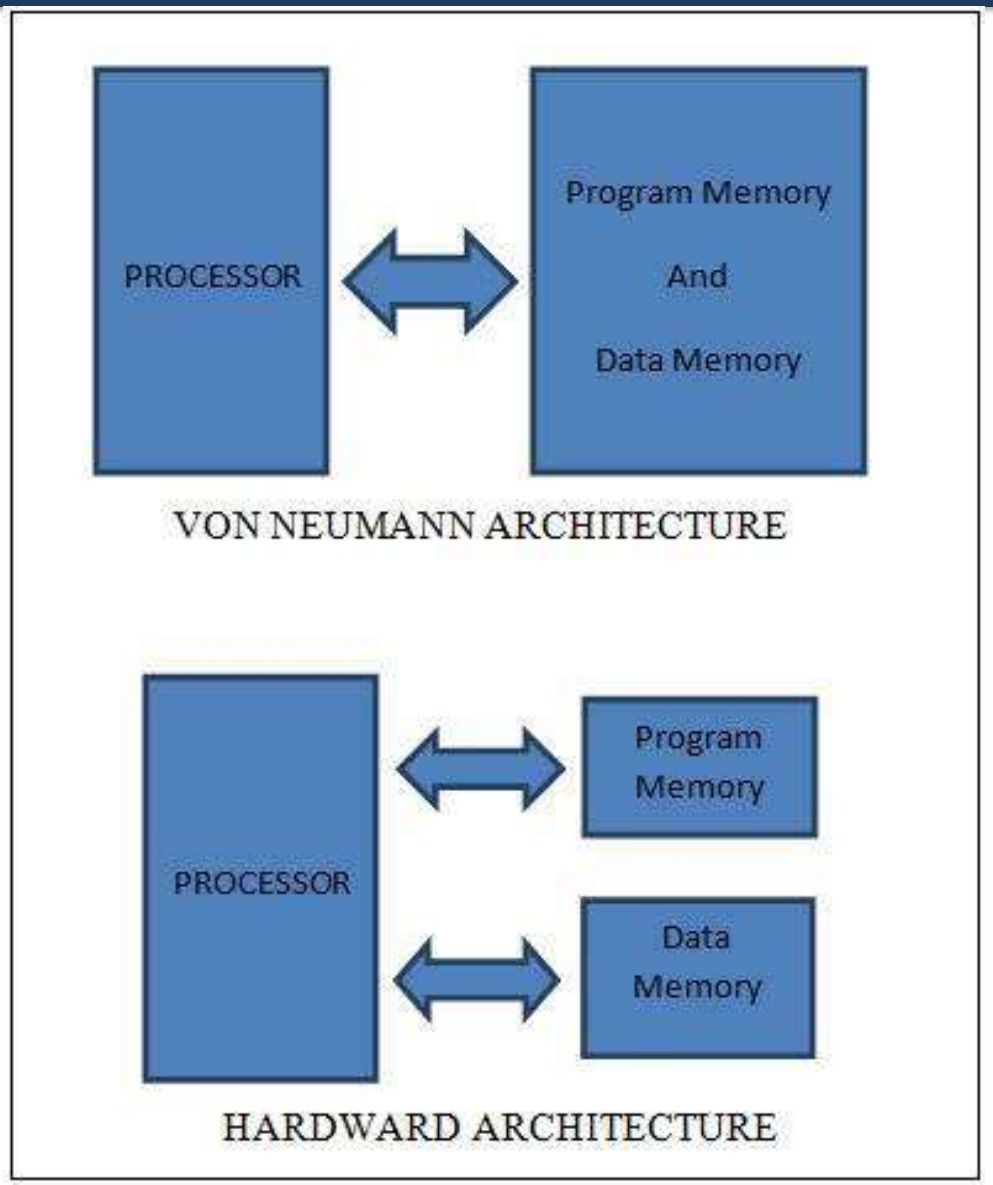
The Von Neumann Architecture

- Key Concepts:
  - Stored Program Concept: Instructions and data are stored in the same memory
  - Sequential Execution: Instructions are executed one after another
  - CPU fetches instructions from memory, decodes them, and executes them
  - Program Counter (PC) tracks the next instruction
- Components: Memory, ALU, Control Unit, Input/Output

Von Neumann Architecture Diagram



Von Neumann Architecture Diagram



Components of a CPU

The Central Processing Unit (CPU) consists of:

1. Control Unit (CU)
    - Directs operation of the processor
    - Decodes instructions
    - Generates control signals
2. Arithmetic Logic Unit (ALU)
    - Performs arithmetic operations (add, subtract, multiply, divide)
    - Performs logical operations (AND, OR, NOT, XOR)
    - Performs shift operations
3. Registers
    - High-speed storage inside the CPU
    - Hold data, instructions, and addresses

CPU Components (Continued)

4. Program Counter (PC) – Holds address of next instruction to execute
5. Instruction Register (IR) – Holds current instruction being executed
6. Memory Address Register (MAR) – Holds address of memory location to access
7. Memory Data Register (MDR) – Holds data being transferred to/from memory
8. Status/Flags Register – Holds condition codes (zero, negative, overflow, carry)

CPU Internal Structure

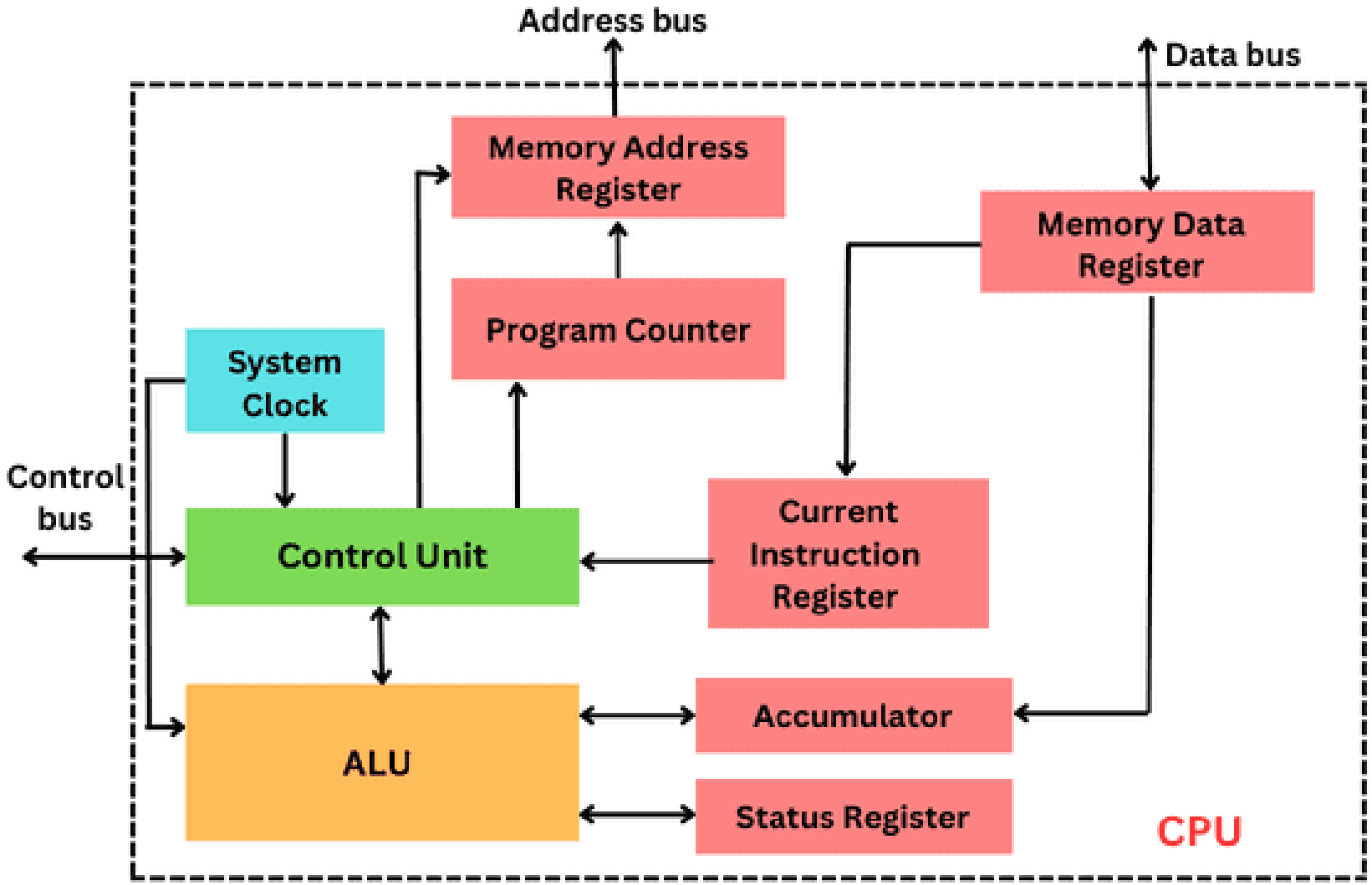
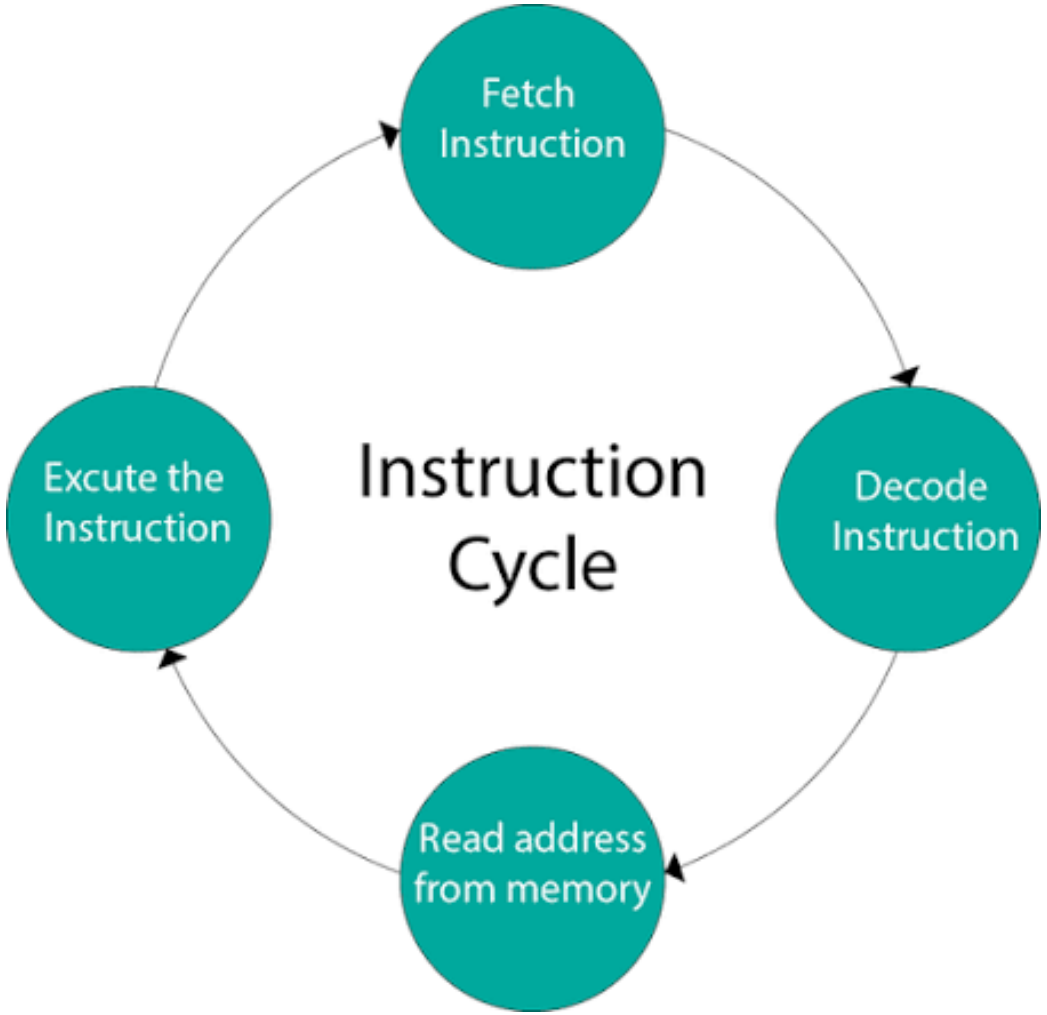


Fig: Basic components of Central Processing Unit

The Instruction Cycle

```
Basic Instruction Cycle (Fetch-Decode-Execute):

1. FETCH:   Get instruction from memory at
address in PC
2. DECODE:  Determine what operation to perform
3. EXECUTE: Perform the operation
4. STORE:   Write results back if needed
5. UPDATE PC: Move to next instruction
```



Instruction Set Architecture (ISA)

- Definition: The ISA is the interface between software and hardware.
- It defines:
  - Instruction set – Operations the processor can perform
  - Data types – Supported data formats (integer, floating-point)
  - Registers – Programmer-visible storage
  - Addressing modes – How operands are specified
  - Memory organization – Address space, endianness
  - I/O mechanisms – How I/O is performed

ISA Examples

ISA	Used In	Type
x86	Intel, AMD processors	CISC
ARM	Mobile devices, embedded	RISC
RISC-V	Research, education, emerging	RISC
MIPS	Education, embedded	RISC
PowerPC	Older Macs, embedded	RISC

RISC vs. CISC

Feature	RISC	CISC
Instruction Set	Small, simple	Large, complex
Instruction Length	Fixed	Variable
Addressing Modes	Few	Many
Memory Access	Load/Store only	Direct memory operations
Execution	Single cycle (mostly)	Multiple cycles
Pipelining	Easy	Difficult
Examples	RISC-V, ARM, MIPS	x86, VAX

Why Learn Computer Architecture?

1. Write better software – Understand performance implications
2. Optimize programs – Know how hardware executes code
3. Design new systems – Foundation for hardware design
4. Understand security – Hardware vulnerabilities (Spectre, Meltdown)
5. Career opportunities – CPU design, embedded systems, compilers
6. Appreciate technology – How computers really work

Real-World Applications

- Computer Architecture knowledge is used in:
  - Processor Design: Intel, AMD, ARM, Apple (M-series)
  - Embedded Systems: IoT devices, automotive, medical
  - Compilers: Optimizing code for specific architectures
  - Operating Systems: Memory management, scheduling
  - AI/ML Hardware: GPUs, TPUs, NPUs
  - Security: Hardware-based security features

Course Roadmap

```
Lecture 1: Introduction (Today)
    ↓
Lectures 2-4: Number Systems, Programming Paradigms
    ↓
Lectures 5-10: RISC-V Microarchitecture & ISA
    ↓
Lectures 11-20: Single Cycle Processor Design
    ↓
Lectures 21-30: Pipelining
    ↓
Lectures 31-40: Memory Hierarchy & Cache
    ↓
Lectures 41-48: Parallel Processing
```

Key Takeaways from Lecture 1

1. Computer Architecture defines what a computer does; Organization defines how
2. Von Neumann architecture stores both instructions and data in memory
3. CPU components: Control Unit, ALU, Registers, PC, IR, MAR, MDR, Flags
4. Instruction Cycle: Fetch → Decode → Execute → Store → Update PC
5. ISA is the interface between software and hardware
6. RISC vs. CISC: Different philosophies in instruction set design
7. This course focuses on RISC-V architecture and processor design

Reading Assignment

- Required Reading:
  - Patterson & Hennessy, Chapter 1: "Computer Abstractions and Technology"
    - Section 1.1: Introduction
    - Section 1.2: Eight Great Ideas in Computer Architecture
    - Section 1.3: Below Your Program
    - Section 1.4: Under the Covers
- Optional Reading:
  - Hamacher et al., Chapter 1: "Basic Structure of Computers"

Eight Great Ideas in Computer Architecture

Preview (from Patterson & Hennessy):

1. Design for Moore's Law
2. Use Abstraction to Simplify Design
3. Make the Common Case Fast
4. Performance via Parallelism
5. Performance via Pipelining
6. Performance via Prediction
7. Hierarchy of Memories
8. Dependability via Redundancy

We will explore these throughout the course.

Discussion Questions

1. What is the difference between computer architecture and computer organization?
2. Why is the Von Neumann architecture still dominant today?
3. What are the main components of a CPU and their functions?
4. How does the instruction cycle work?
5. Why has RISC architecture become increasingly popular?
6. What is the role of the ISA in computer design?

Lab/Assignment Preview

- Upcoming Lab:
  - Introduction to RISC-V simulator (RARS or Venus)
  - Writing simple assembly programs
  - Understanding registers and memory
- Assignment 1 (Preview):
  - Research a modern processor (e.g., Apple M3, Intel Core i9)
  - Identify its architecture, ISA, and key features
  - Compare with RISC-V

Summary

- In this lecture, we covered:
  - Definition of Computer Architecture and Organization
  - Components of a computer system
  - Von Neumann architecture and stored program concept
  - CPU components and their functions
  - The instruction cycle
  - Introduction to ISA
  - RISC vs. CISC comparison
  - Course roadmap and learning outcomes
- Next Lecture: Number Systems and Programming Paradigms

Questions?

Thank You!

Contact Information:
Course Instructor: Kamran Babar
Email: Kamran.babar@cust.edu.pk
Office Hours: Monday/Thursday (1000 AM to 1200PM)
Course Website: [URL]

Remember: Read Chapter 1 before next lecture.
Install RISC-V simulator for upcoming labs.

Topic 2: Number Systems and Programming Paradigms

CPE2323: Computer Architecture and Organization
Number Representation • Arithmetic Operations
IEEE 754 Floating-Point • Programming Paradigms
Capital University of Science and Technology, Islamabad

Topic 2 Overview

- Four main subtopics covered in this topic:
  1. Number Representation in Computers
     - Binary, hex, signed/unsigned, two's complement
  2. Addition, Subtraction, and Multiplication Operations
     - Hardware algorithms and their implementation
  3. Single-Precision Floating-Point Numbers
     - IEEE 754 format, encoding, special values
  4. High-Level vs. Low-Level Programming Paradigm
     - Compilation flow, assembler, linker, RISC-V assembly
- By the end you will understand how data and operations live inside the machine.

Section 1

Number Representation in Computers

Why Do Computers Use Binary?

- Digital circuits have two stable states:
  - Off / On, Low voltage / High voltage
  - Represented as 0 and 1
- Advantages of binary:
  - Simple, reliable hardware (two states easy to distinguish)
  - Noise immunity (clear threshold between 0 and 1)
  - Boolean algebra directly applies (AND, OR, NOT)
  - Cheap to manufacture at massive scale
- Historical alternatives:
  - Decimal computers (ENIAC), ternary (Setun) — abandoned

Positional Number Systems

System	Base	Digits Used	Example (value in decimal)
Binary	2	0, 1	$1011_2 = 11$
Octal	8	0–7	$17_8 = 15$
Decimal	10	0–9	$15_{10} = 15$
Hexadecimal	16	0–9, A–F	$B_{16} = 11$
Base 32	32	0–9, A–V	Rare (used in some encodings)
Base 64	64	A–Z, a–z, 0–9, +, /	Base64 encoding for data

Binary to Decimal Conversion

```
Method 1: Sum of powers of 2

Example: 1 0 1 1 0 1
Position:  5 4 3 2 1 0
Weight:   32 16 8 4 2 1

Value = 1×32 + 0×16 + 1×8 + 1×4 + 0×2 + 1×1
      = 32 + 8 + 4 + 1
      = 45

Method 2: Double-and-add (Horner's method)
  Start with 0; for each bit left to right: result = result × 2 + bit
  1 -> 1, 10 -> 2, 101 -> 5, 1011 -> 11, 10110 -> 22, 101101 -> 45
```

Decimal to Binary Conversion

```
Method 1: Repeated division by 2 (bottom-up)

Convert 45 to binary:
  45 / 2 = 22  R 1   <- LSB
  22 / 2 = 11  R 0
  11 / 2 =  5  R 1
   5 / 2 =  2  R 1
   2 / 2 =  1  R 0
   1 / 2 =  0  R 1   <- MSB

Result: 45 = 101101 (binary)

Method 2: Subtract largest power of 2
  45 - 32 = 13 -> set bit at 32
  13 -  8 =  5 -> set bit at 16
   5 -  4 =  1 -> set bit at  8
   1 -  1 =  0 -> set bits at 4, 2
  Result: 101101
```

Binary to Hexadecimal Conversion

```
Each hex digit represents exactly 4 binary bits (a 'nibble')

Hex to binary:
  Hex:     B    A    7    3
  Binary: 1011 1010 0111 0011

Binary to hex:
  Binary: 1100 1111 0001 1010
  Hex:    C    F    1    A

Common patterns:
  0 = 0000,  1 = 0001,  2 = 0010,  3 = 0011
  4 = 0100,  5 = 0101,  6 = 0110,  7 = 0111
  8 = 1000,  9 = 1001,  A = 1010,  B = 1011
  C = 1100,  D = 1101,  E = 1110,  F = 1111

32 bits = 8 hex digits; 64 bits = 16 hex digits
```

Unsigned Integers

- An n-bit unsigned integer represents values from 0 to 2^n - 1
- Common widths:
  - 8-bit: 0 to 255
  - 16-bit: 0 to 65,535
  - 32-bit: 0 to 4,294,967,295 (4 GiB - 1)
  - 64-bit: 0 to 18,446,744,073,709,551,615
- Every bit pattern represents a unique value
- Used for: addresses, sizes, counters, character codes, hashes
- Arithmetic wraps around modulo 2^n on overflow

Why Signed Numbers Are Tricky

- We need to represent negative numbers in binary
- Three historical schemes:
  1. Sign-Magnitude: MSB is sign bit
  2. One's Complement: negate = flip all bits
  3. Two's Complement: negate = flip bits, add 1
- Requirements for a good scheme:
  - Only one representation of zero
  - Simple hardware for add/subtract
  - Sign detectable from MSB
  - Extendable to wider widths

Two's Complement - The Rules

- Representation of an n-bit signed integer:
  - Value = -(b_n-1) * 2^(n-1) + sum(b_i * 2^i) for i=0 to n-2
- Range: -2^(n-1) to +2^(n-1) - 1
  - 8-bit: -128 to +127
  - 16-bit: -32,768 to +32,767
  - 32-bit: -2,147,483,648 to +2,147,483,647
- Negation: invert all bits, add 1
- Sign bit is MSB (1 = negative, 0 = non-negative)
- Only one zero (0000...0)
- Asymmetric range (one more negative than positive)

Two's Complement 4-bit Values

Binary	Unsigned Value	Two's Complement Value
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	-8
1001	9	-7
1010	10	-6
1011	11	-5
1100	12	-4
1101	13	-3
1110	14	-2
1111	15	-1

Two's Complement - Negation Examples

```
Rule: invert all bits, then add 1

Example 1: Find -5 in 8-bit two's complement
  +5       = 0000 0101
  Invert   = 1111 1010
  Add 1    = 1111 1011
  Check: 1111 1011 = -128 + 64+32+16+8+2+1 = -128 + 123 = -5 OK

Example 2: Find -1
  +1       = 0000 0001
  Invert   = 1111 1110
  Add 1    = 1111 1111  (all 1s = -1)

Example 3: Find -128
  +128 can't be represented in 8-bit signed!
  1000 0000 = -128 (special case)
```

Sign Extension

```
When widening a signed value, replicate the sign bit

Example: extend 8-bit -5 to 16 bits
  8-bit:   1111 1011
  16-bit:  1111 1111 1111 1011
           ^^^^^^^^^ new bits filled with sign (1)

Example: extend 8-bit +5 to 16 bits
  8-bit:   0000 0101
  16-bit:  0000 0000 0000 0101
           ^^^^^^^^^ new bits filled with 0

Rule: new high bits = old MSB
Zero-extension is for unsigned values (fill with 0)
RISC-V has lb/lh (sign-extend) and lbu/lhu (zero-extend)
```

Overflow in Signed Arithmetic

- Overflow: result cannot be represented in available bits
- Rule for two's complement addition:
  - Overflow occurs iff: operands have same sign AND result has different sign
- Examples (4-bit):
  - 0111 (+7) + 0001 (+1) = 1000 -> interpreted as -8 -> OVERFLOW
  - 1000 (-8) + 1111 (-1) = 0111 -> interpreted as +7 -> OVERFLOW
  - 0111 (+7) + 1111 (-1) = 0110 -> +6 -> no overflow
- Hardware detects overflow via XOR of carry-in/carry-out of MSB
- RISC-V: base add/sub do NOT trap on overflow

Unsigned vs. Signed Comparison

- The same bit pattern can represent different values

- Bit pattern: 1111 1111
- Unsigned: 255
- Signed: -1

- Comparison example: which is greater, 1000 0000 or 0111 1111?
  - Unsigned: 128 > 127 (both positive)
  - Signed: -128 < 127 (one is negative!)
- RISC-V has separate instructions:
  - blt, bge - signed comparison
  - bltu, bgeu - unsigned comparison
- Choosing the wrong one leads to subtle bugs

Section 1 Summary - Number Representation

- Binary is the natural language of digital circuits
- Positional systems: value depends on digit position
- Hex is a compact way to write binary (4 bits per digit)
- Unsigned n-bit integers: 0 to 2^n - 1
- Two's complement is the dominant signed representation
  - Range: -2^(n-1) to +2^(n-1) - 1
  - Negation: invert + add 1
  - Single zero, simple hardware
- Sign extension preserves value when widening
- Overflow detectable via carry-in XOR carry-out of MSB

Section 2

Addition, Subtraction, and Multiplication Operations

The Half Adder

Adds two 1-bit numbers (A, B)

Truth table:

A	B	SUM	CARRY
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

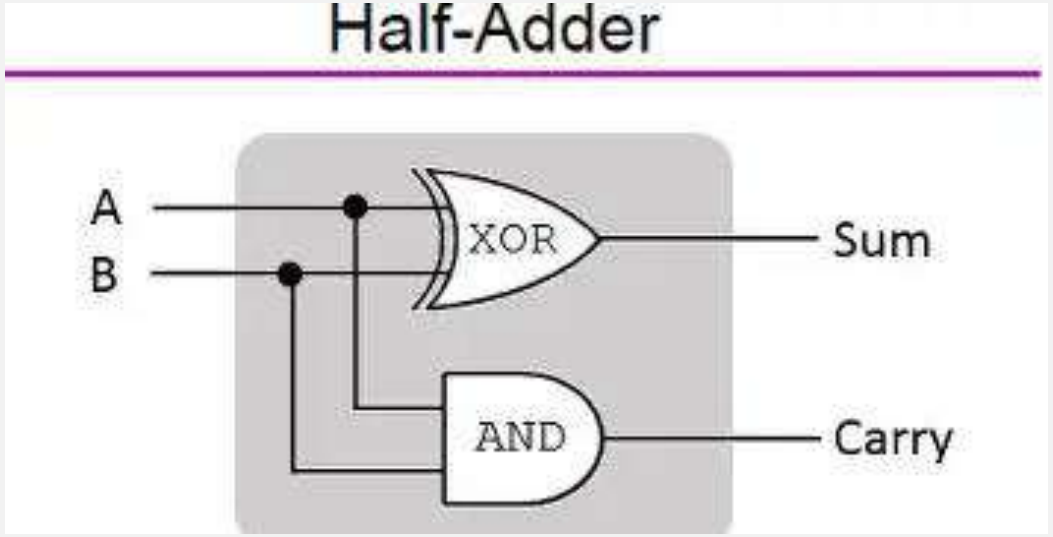
Boolean equations:

SUM = A XOR B

CARRY = A AND B

Limitation: cannot accept carry-in from a lower bit

Used only for the LSB position in a multi-bit adder



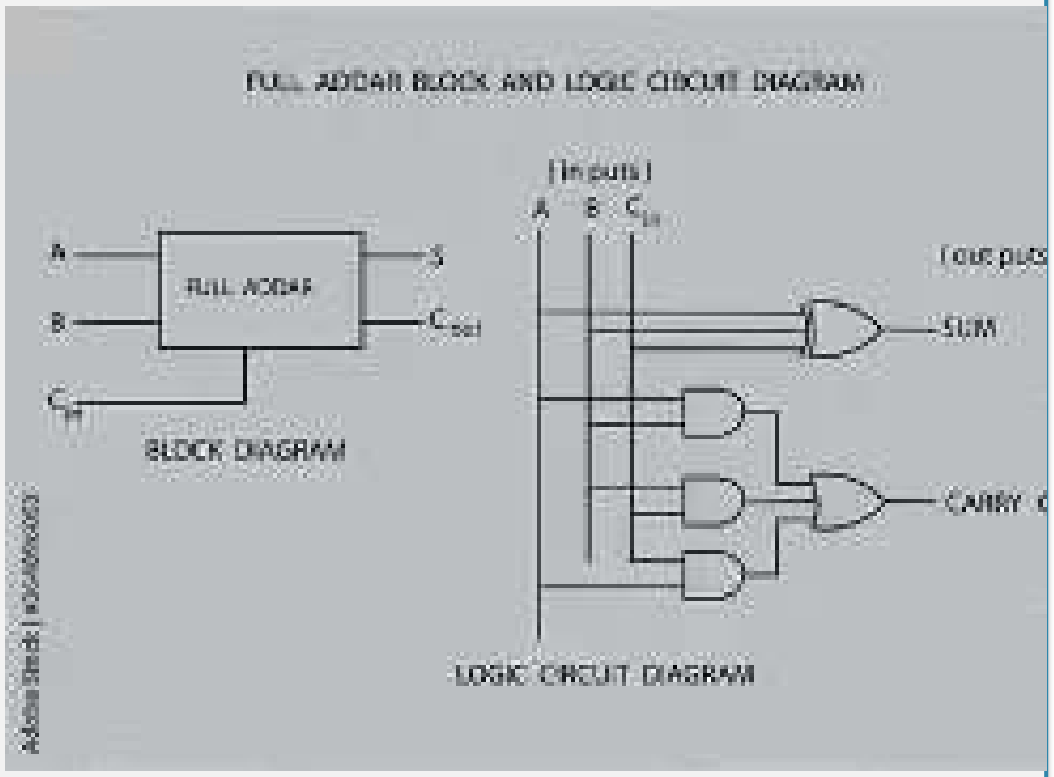
The Full Adder

Adds three 1-bit numbers (A, B, Cin)

Truth table (abbreviated):

A	B	Cin	SUM	Cout
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

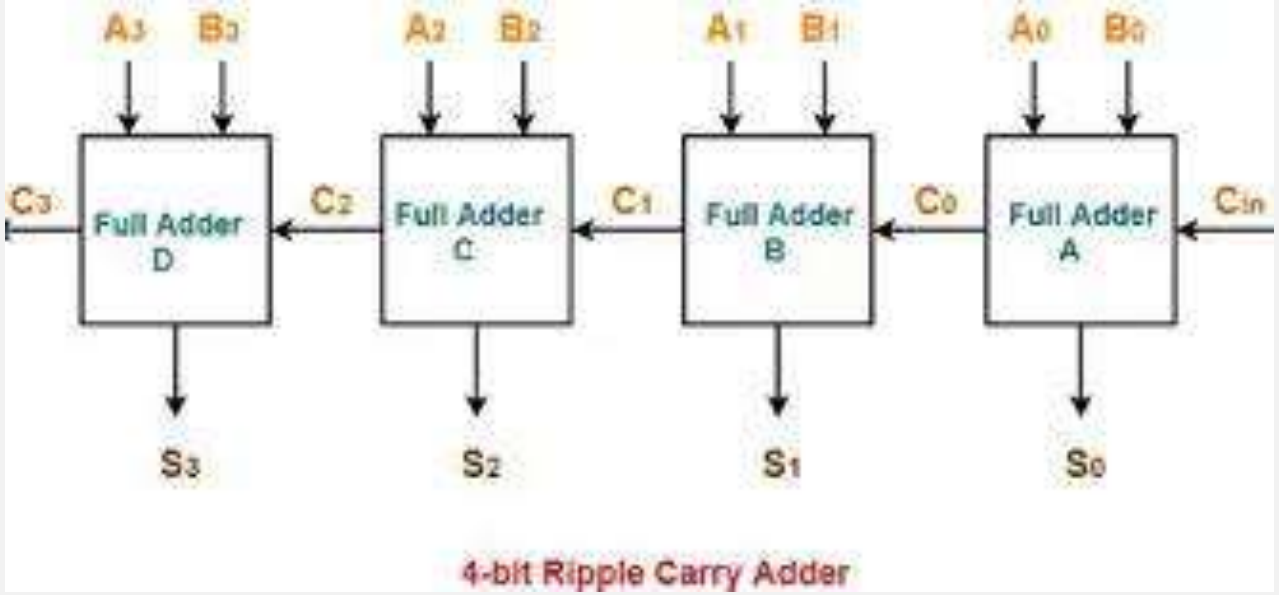
Equations: SUM = A XOR B XOR Cin
Cout = (A AND B) OR (B AND Cin) OR (A AND Cin)



Ripple-Carry Adder (n-bit)

Chain n full adders:

Delay: O(n) - worst case carry propagates through all stages
For 32-bit: ~32 gate delays - significant for high clock speeds
Solution: carry-lookahead, carry-select, or carry-skip adders



4-bit Ripple Carry Adder

Overflow Detection Circuit

```
For signed two's complement addition:
  Overflow = Cn XOR Cn-1
  where Cn is carry-out of MSB, Cn-1 is carry into MSB

Why this works:
  - Same-sign operands + opposite-sign result = overflow
  - Equivalent to: MSB carry-in differs from MSB carry-out

Example (4-bit):
  0111 (+7) + 0001 (+1):
  The MSB addition is 0 + 0 + Cin(=1) = 1 with Cout=0
  Cin=1, Cout=0 -> overflow!

For unsigned: overflow = Cn (carry-out of MSB)
RISC-V doesn't have a dedicated overflow flag by default
```

Subtraction = Addition with Negation

```
Key insight: A - B = A + (-B)

In two's complement:
  -B = invert(B) + 1

Circuit for A - B:
  - Invert all bits of B
  - Set carry-in of LSB to 1 (this adds the +1)
  - Use the same adder for A + (~B) + 1

  A ---------> +----+
               | FA |---> Result
  B --> XOR -->+----+
        ^
       SUB (control signal)

If SUB=0: B passes through unchanged -> addition
If SUB=1: B is inverted, carry-in=1 -> subtraction
```

Multiplication - The Basics

```
Decimal multiplication: partial products + addition
      1234
    x   56
    ------
      7404    (1234 x 6)
     61700    (1234 x 5, shifted left)
    ------
     69104

Binary multiplication: same idea, simpler digits
  - Partial product = multiplicand AND each multiplier bit
  - Each partial product is 0 or a copy of the multiplicand
  - Sum shifted partial products

n-bit x n-bit -> up to 2n-bit product
```

Unsigned Multiplication - Example

```
Multiply 1101 (13) x 1011 (11)

          1101   (13)
         x 1011   (11)
         ------
          1101   <- 13 x 1 (bit 0)
         1101    <- 13 x 1 (bit 1, shifted left 1)
        0000     <- 13 x 0 (bit 2, shifted left 2)
       1101      <- 13 x 1 (bit 3, shifted left 3)
      --------
     10001111    -> 143

Verify: 13 x 11 = 143 OK

Each multiplier bit: either 0 (skip) or a shifted copy of multiplicand
```

Unsigned Multiplication Hardware

```
Simple shift-and-add multiplier:

  Registers:
    Multiplicand (64-bit for n=32)
    Multiplier (32-bit)
    Product (64-bit, initialized to 0)

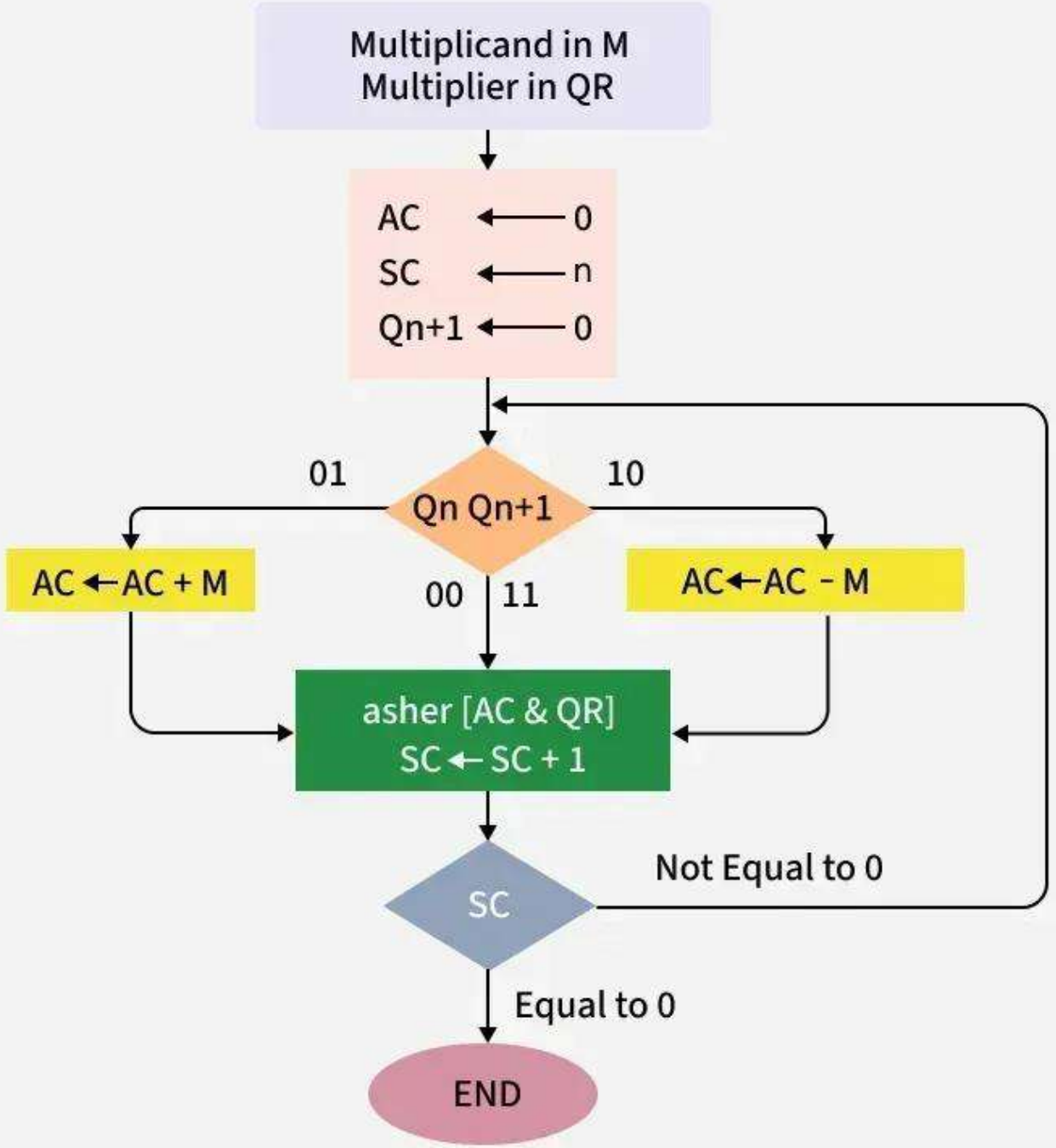
  Each iteration (32 iterations for 32-bit):
    if Multiplier[0] == 1:
        Product += Multiplicand
    Multiplicand <<= 1
    Multiplier >>= 1

Cost: ~32 cycles for 32-bit x 32-bit
Alternative: combinational array multiplier (single cycle, more area)
RISC-V: 'mul' instructions implemented by hardware multiplier
```

Signed Multiplication - Booth's Algorithm

- Problem: naive shift-and-add doesn't handle negatives directly
- Booth's algorithm: handles signed multiplication elegantly
- Key idea: instead of adding for every 1-bit, add or subtract for transitions
  - From 0 to 1: subtract multiplicand
  - From 1 to 0: add multiplicand
  - No transition: no operation
- Advantages:
  - Handles signed numbers uniformly
  - Fewer operations for many patterns (fewer partial products)
  - Extended in modern CPUs to radix-4 Booth (2 bits per step)

Booth's Algorithm



Booth's Algorithm

Input : 0110, 0010

Output :

qn	q[n+1]		AC	QR	sc(step count)
		initial	0000	0010	4
0	0	rightShift	0000	0001	3
1	0	A = A - BR	1010		
		rightShift	1101	0000	2
0	1	A = A + BR	0011		
		rightShift	0001	1000	1
0	0	rightShift	0000	1100	0

Result=1100

RISC-V M Extension - Multiply/Divide Instructions

Instruction	Operation	Result
mul rd, rs1, rs2	Signed x Signed	Lower 32 bits of product
mulh rd, rs1, rs2	Signed x Signed	Upper 32 bits of product
mulhu rd, rs1, rs2	Unsigned x Unsigned	Upper 32 bits
mulhsu rd, rs1, rs2	Signed x Unsigned	Upper 32 bits
div rd, rs1, rs2	Signed divide	Quotient
divu rd, rs1, rs2	Unsigned divide	Quotient
rem rd, rs1, rs2	Signed remainder	Remainder
remu rd, rs1, rs2	Unsigned remainder	Remainder

Division - Overview

- Binary division mirrors decimal long division
- Algorithm:
  - Start with divisor shifted left to align with dividend's MSB
  - Compare: if divisor <= dividend portion, subtract and set quotient bit to 1
  - Otherwise, set quotient bit to 0
  - Shift divisor right, repeat
- Restoring division: keeps original if subtraction fails
- Non-restoring division: faster variant, handles failures differently
- RISC-V provides div/rem instructions in the M extension
- Division is slow (typically 20-40 cycles); some embedded cores omit it

Section 2 Summary - Arithmetic Operations

- Half adder: 2 inputs; Full adder: 3 inputs (A, B, Cin)
- Ripple-carry adder: n full adders chained, O(n) delay
- Subtraction = addition with inverted B and Cin=1
- Overflow: MSB carry-in XOR MSB carry-out (for signed)
- Multiplication: shift-and-add of partial products
- Booth's algorithm handles signed multiplication
- Division: restoring or non-restoring long division
- RISC-V M extension: mul, mulh, div, rem and variants

Section 3

Single-Precision Floating-Point Numbers

Why Floating-Point?

- Integers can't represent:
  - Very large numbers (astronomical, scientific)
  - Very small numbers (atomic scales)
  - Fractional values (0.5, 3.14159, 1/3)
- Fixed-point: limited range and precision
- Floating-point: decimal point 'floats' to accommodate varying magnitudes
- Analogous to scientific notation:
  - 1.23 x 10^5, 6.02 x 10^23, 1.6 x 10^-19
- Standardized as IEEE 754 in 1985; universal today
- RISC-V F extension provides 32-bit single-precision floating-point

IEEE 754 Single Precision Format

```
32-bit format:

  31  30          23  22                      0
  +---+--------------+-----------------------+
  | S |   Exponent   |       Mantissa        |
  +---+--------------+-----------------------+
   1      8 bits             23 bits

Value = (-1)^S x 1.M x 2^(E-127)

Components:
  S = sign bit (0 = positive, 1 = negative)
  E = biased exponent (actual exponent = E - 127)
  M = mantissa (fractional part, implicit leading 1)

Double precision (64-bit): 1 + 11 + 52 = 64 bits
```

Encoding Example: 0.15625

```
Step 1: Convert to binary
  0.15625 = 0.00101 (binary)

Step 2: Normalize (one digit before point)
  0.00101 = 1.01 x 2^(-3)

Step 3: Extract fields
  Sign S = 0 (positive)
  Actual exponent = -3
  Biased exponent = -3 + 127 = 124 = 0111 1100
  Mantissa = 010 0000 0000 0000 0000 000  (23 bits)

Step 4: Combine
  0 01111100 01000000000000000000000
  Group into nibbles: 0011 1110 0010 0000 0000 0000 0000 0000
  Hex: 0x3E200000
```

Decoding Example: 0x40490FDB

```
Given 32-bit value 0x40490FDB, find its decimal value

Step 1: Convert to binary and split fields
  0x40490FDB = 0100 0000 0100 1001 0000 1111 1101 1011
  Sign S = 0
  Exponent E = 1000 0000 = 128
  Mantissa M = 100 1001 0000 1111 1101 1011

Step 2: Compute actual exponent
  Actual E = 128 - 127 = 1

Step 3: Compute mantissa as decimal fraction
  1.M = 1.10010010000111111011011 (binary)
      ~ 1.5707963268

Step 4: Combine
  Value = 1.5707963268 x 2^1 ~ 3.14159265 (pi!)
```

Normalized vs. Denormalized Numbers

- Normalized (E != 0, E != 255):
  - Value = (-1)^S x 1.M x 2^(E-127)
  - Implicit leading 1
  - Most common representation
- Denormalized (E = 0, M != 0):
  - Value = (-1)^S x 0.M x 2^(-126)
  - Implicit leading 0 (not 1)
  - Allow gradual underflow to zero
- Zero (E = 0, M = 0):
  - +/-0 (positive and negative zero exist!)
- Purpose: denormals fill the gap between smallest normal and zero

IEEE 754 Special Values

Exponent	Mantissa	Meaning	Example
0	0	+/- Zero	+0.0, -0.0
0	!= 0	Denormalized	Very small numbers
1-254	Any	Normalized	Most values
255	0	+/- Infinity	1.0 / 0.0
255	!= 0	NaN (Not a Number)	0.0 / 0.0, sqrt(-1)

Floating-Point Arithmetic

- Addition/Subtraction algorithm:
  - Align exponents (shift smaller-exponent number right)
  - Add/subtract mantissas
  - Normalize result (shift to restore 1.M form)
  - Round to fit 23-bit mantissa
  - Check overflow/underflow
- Multiplication:
  - Multiply mantissas
  - Add exponents (then subtract bias)
  - Determine sign (XOR of signs)
  - Normalize and round
- Division is similar but with subtraction of exponents

Floating-Point Rounding Modes

- IEEE 754 defines multiple rounding modes:
  - Round to nearest, ties to even (default)
  - Round toward zero (truncate)
  - Round toward +inf (ceiling)
  - Round toward -inf (floor)
- Round to nearest, ties to even:
  - Choose the closest representable value
  - If exactly halfway, choose the value with even LSB
  - Reduces statistical bias in long computations
- RISC-V F extension supports all modes via fcsr register
- Rounding introduces the 'unit in last place' (ULP) error

Precision Issue: 0.1 + 0.2 != 0.3

```
Why 0.1 has no exact binary representation:
  0.1 = 1/10 = 1/(2 x 5)
  Denominator includes 5, not a power of 2
  Binary can only exactly represent fractions with power-of-2 denominators

Python demonstration:
  >>> 0.1
  0.1000000000000000055511151231257827...
  >>> 0.1 + 0.2
  0.3000000000000000444089209850062616...
  >>> 0.1 + 0.2 == 0.3
  False

Analogy: 1/3 in decimal = 0.3333... never exactly terminates
```

Catastrophic Cancellation

```
Subtracting nearly-equal numbers destroys precision

Example:
  a = 1.0000001   (8 significant digits)
  b = 1.0000000
  a - b = 0.0000001   (only 1 significant digit left!)

If a and b both had 7 significant digits and we subtract:
  All digits cancel except the last
  The result has essentially no precision

Real example: quadratic formula
  x = (-b +/- sqrt(b^2 - 4ac)) / (2a)
  When b^2 >> 4ac, the two roots have severe cancellation
  Fix: use alternative form x = 2c / (-b -/+ sqrt(b^2 - 4ac))
```

Other Floating-Point Pitfalls

- Associativity lost:
  - (a + b) + c may differ from a + (b + c)
  - Order of summation matters in parallel reduction!
- Distributivity lost:
  - a x (b + c) may differ from a x b + a x c
- NaN comparisons:
  - NaN != NaN (even with itself)
  - Use isnan() to test
- -0.0 == 0.0 but 1/(-0.0) = -inf, 1/(0.0) = +inf
- Infinity arithmetic:
  - inf + inf = inf, inf - inf = NaN, inf x 0 = NaN
- Denormals are slow on some CPUs (or disabled)

Section 3 Summary - Floating-Point

- IEEE 754 single: 1 sign + 8 exponent + 23 mantissa
- Value = (-1)^S x 1.M x 2^(E-127)
- Normalized, denormalized, zero, infinity, NaN
- Rounding modes: nearest-even (default), truncate, ceil, floor
- Arithmetic: align exponents, operate on mantissas, normalize, round
- Precision issues: 0.1+0.2 != 0.3, cancellation, associativity lost
- RISC-V F extension provides single-precision FP instructions

Section 4

High-Level vs. Low-Level Programming Paradigm

Programming Language Levels

Level	Examples	Abstraction	Portability
High-level	Python, Java, C++, C	Very high	Cross-platform
Assembly	RISC-V asm, x86 asm	Low	Architecture-specific
Machine code	Binary opcodes	None	Architecture-specific
Microcode	CPU-internal	Below ISA	Specific to microarchitecture

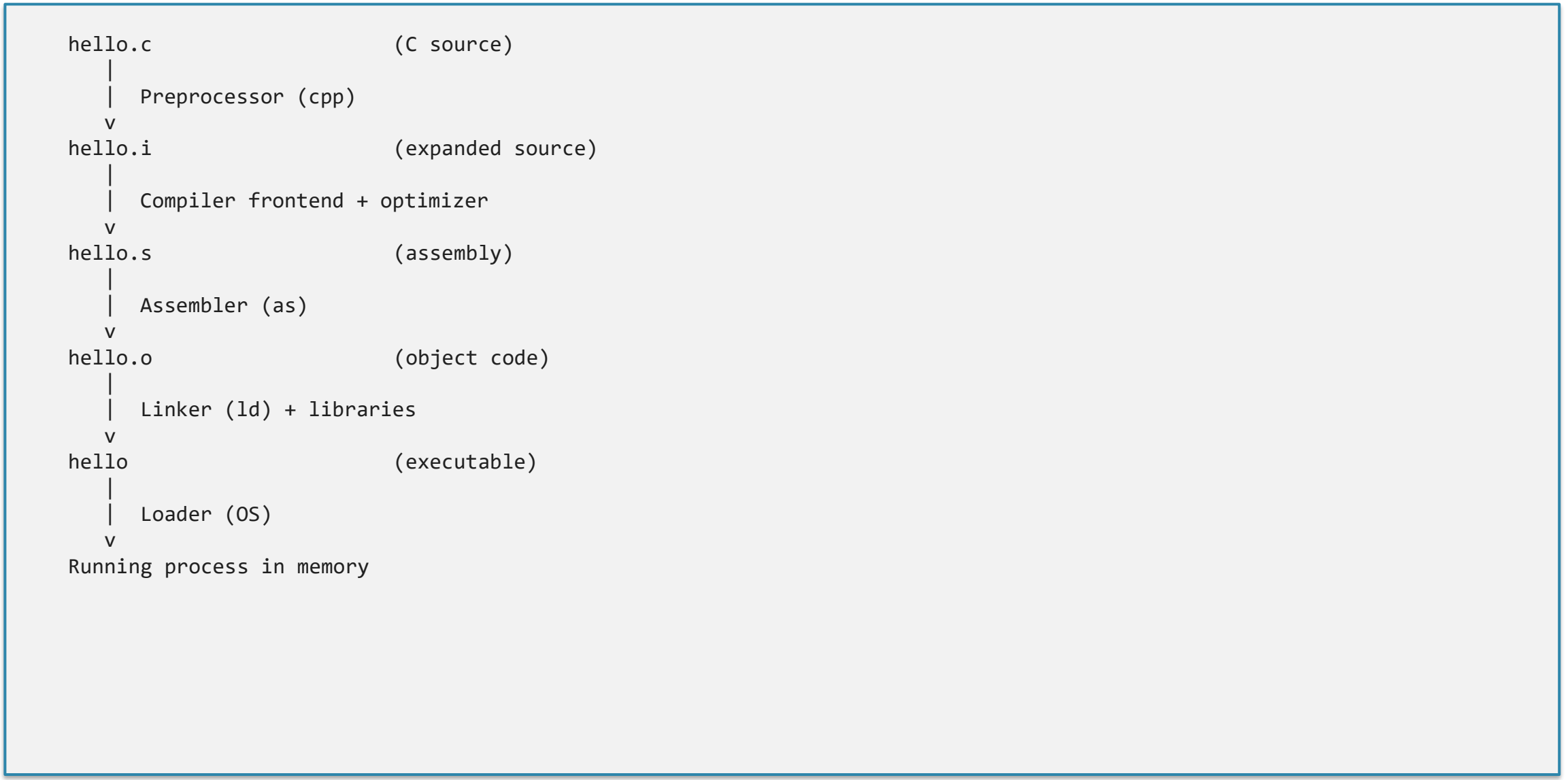
High-Level Language Characteristics

- Advantages:
  - Human-readable syntax (close to English/math)
  - Portable across architectures
  - Rich abstractions (objects, closures, generics)
  - Powerful libraries and frameworks
  - Faster development, easier maintenance
- Disadvantages:
  - Performance depends on compiler
  - Less control over hardware details
  - Hidden costs (allocations, bounds checks, GC)
- Examples: C, C++, Java, Python, Rust, Go, JavaScript

Low-Level Language Characteristics

- Advantages:
  - Direct hardware control
  - Predictable performance and timing
  - Minimal overhead
  - Access to special instructions (CSRs, atomics)
  - Small code size
- Disadvantages:
  - Architecture-specific (not portable)
  - Verbose, harder to read
  - Slower development
  - More bug-prone
- Used in: OS kernels, device drivers, bootloaders, embedded firmware, performance hotspots, cryptography

The Compilation Pipeline



What the Compiler Does

- Parse: build an abstract syntax tree
- Semantic analysis: type checking, name resolution
- IR generation: translate to intermediate representation (IR)
- Optimization: many passes on the IR
  - Constant folding, dead code elimination
  - Loop unrolling, inlining, vectorization
  - Register allocation
- Code generation: IR -> target assembly
- Target-specific optimization: instruction scheduling
- Result: readable (or optimized) assembly code

The Assembler and Linker

- Assembler (as):
  - Translates mnemonics to binary opcodes
  - Resolves local labels to addresses
  - Expands pseudo-instructions (li, mv)
  - Produces relocatable object file (.o)
- Object file contains:
  - Machine code sections (.text)
  - Data sections (.data, .bss)
  - Symbol table (function/variable names)
  - Relocation info (unresolved addresses)
- Linker (ld):
  - Combines multiple .o files
  - Resolves external symbols
  - Assigns final addresses
  - Produces executable

RISC-V Assembly Syntax

```
General form:
  LABEL:  MNEMONIC  OPERANDS   # comment

Example:
  main:                 # label
    li    a0, 42        # load immediate
    mv    a1, a0        # move (pseudo)
    add   a2, a0, a1    # arithmetic
    lw    t0, 0(sp)     # load word
    sw    t0, 4(sp)     # store word
    beq   a0, a1, done # branch if equal
    j     main          # jump (pseudo)
  done:
    ret                 # return (pseudo)

Registers: x0-x31 (hardware names) or ABI names
```

Directives and Sections

```
Assembly directives start with '.' (dot)

Sections:
  .data       - initialized data
  .bss        - uninitialized data
  .text       - executable code
  .rodata     - read-only data (constants)

Common directives:
  .globl sym  - make symbol visible to linker
  .word v     - emit 32-bit value
  .half v     - emit 16-bit value
  .byte v     - emit 8-bit value
  .string s   - emit null-terminated string
  .align n    - align next item to 2^n-byte boundary
  .space n    - reserve n bytes of zero
```

Example: Hello World in RISC-V

```
# hello.s - RISC-V Linux syscall demo

.data
msg:  .string  "Hello, RISC-V!\n"

.text
.globl  main
main:
  # Write syscall (a7 = 64)
  li    a0, 1          # a0 = file descriptor (stdout)
  la    a1, msg        # a1 = buffer address
  li    a2, 14         # a2 = length
  li    a7, 64         # a7 = write syscall number
  ecall                # invoke syscall

  # Exit syscall (a7 = 93)
  li    a7, 93
  ecall
```

Running RISC-V Programs

- Options for running RISC-V code:
  - RARS - RISC-V Assembler and Runtime Simulator (Java, popular in education)
  - Venus - web-based RISC-V simulator (no install needed)
  - SPIKE - official RISC-V ISA simulator (from Berkeley)
  - QEMU - full-system emulation (can run Linux)
  - Real hardware (SiFive HiFive, VisionFive, etc.)
- Typical workflow:
  - Write assembly in a text editor
  - Load into simulator
  - Assemble (check for errors)
  - Run, single-step, or set breakpoints
  - Inspect registers and memory

Why Learn Assembly?

- Even if you'll mostly write high-level code, learning assembly helps you:
  - Understand what your compiler produces
  - Optimize performance-critical code
  - Debug at the hardware level
  - Understand OS and embedded concepts
  - Write faster and smaller code when needed
  - Understand security vulnerabilities
  - Prepare for hardware design work
- Assembly is not obsolete — it's foundational

Section 4 Summary - Programming Paradigms

- High-level languages: portable, readable, abstract
- Low-level languages: direct control, architecture-specific
- Compilation pipeline: source -> assembly -> object -> executable
- Assembler: mnemonic -> binary; Linker: combines modules
- RISC-V assembly: simple, clean, regular syntax
- Simulators (RARS, Venus, SPIKE) for learning
- Assembly is a foundational skill for any computer engineer

Section 5

Topic 2 Wrap-Up and Key Takeaways

Topic 2 - Key Takeaways

- Number Representation:
  - Binary, hex, two's complement, sign extension, overflow
- Arithmetic Operations:
  - Full adders, ripple-carry, subtraction as addition, multiplication
- Floating-Point:
  - IEEE 754 single: S+E+M, precision issues, rounding
- Programming Paradigms:
  - Compilation flow, assembler, linker, RISC-V assembly syntax
- Everything here is assumed knowledge for the rest of CPE2323

Practice Problems

- Try these before the next lecture:
- 1. Convert 200 (decimal) to binary, octal, and hexadecimal
- 2. Find two's complement representation of -100 in 16-bit
- 3. Compute 1000 0110 + 0111 1010; identify if overflow occurs (signed)
- 4. Encode -3.75 in IEEE 754 single precision (hex)
- 5. Decode 0x41A00000 as a float
- 6. Write RISC-V assembly for 'if (a > b) c = a - b; else c = 0;'
- 7. Explain what 'sign extension' means and why it's needed
- 8. Why does 0.1 + 0.2 != 0.3 in most languages?

Questions?

Thank You!

Overview of RISC-V Microarchitecture

CPE2323: Computer Architecture and Organization

Programming Model • Registers • Memory & Stack

Lecture Outline

- Programming Model of RISC-V Microprocessors
- RISC-V Registers, Data Registers & Status Register
- Memory Address Space and Stack
- Summary and Review

What is a Programming Model?

- Abstract view of the computer used by programmers
- Defines:
  - Visible registers
  - Available instructions
  - Memory addressing and access
  - Data representation
  - Exception and interrupt handling
- Hides microarchitectural details (pipeline, cache)

Programming Model vs. Microarchitecture

- Programming Model (ISA): visible, stable, defines what
- Microarchitecture: hidden, implementation-specific, defines how
- Same ISA → many microarchitectures (e.g., SiFive E31, Rocket, BOOM)

RISC-V Design Philosophy

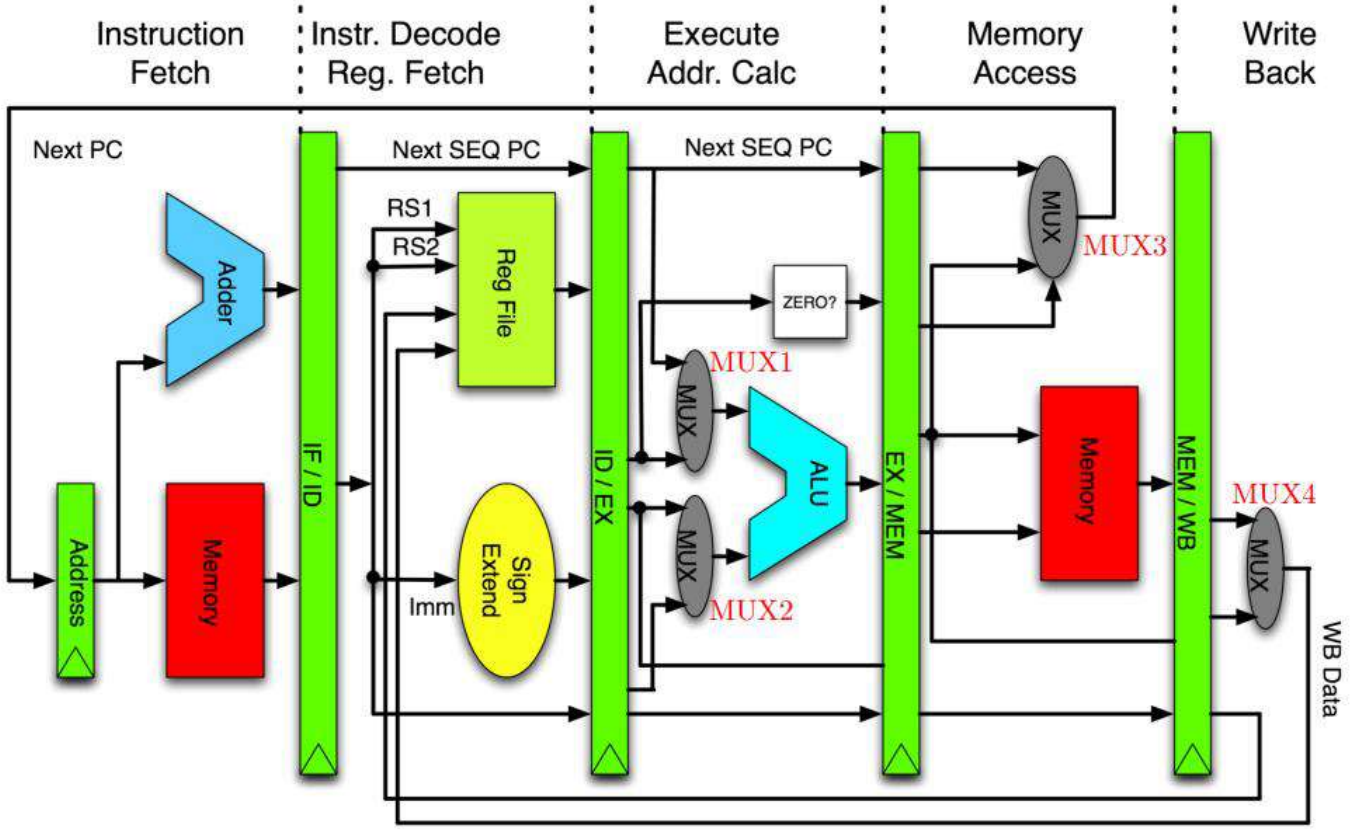
- Simplicity favors regularity
  - Fixed 32-bit instructions in base ISA
- Smaller is faster
  - 32 registers, compact instruction set
- Make the common case fast
  - Load/store architecture
- Good design demands good compromises
  - Base ISA + modular extensions (M, A, F, D, C)

RISC-V Base ISA Overview

- RV32I — 32-bit integer base
- RV64I — 64-bit integer base
- RV128I — 128-bit integer base
- Standard extensions: M, A, F, D, C
- G = I + M + A + F + D
- This course focuses on RV32I

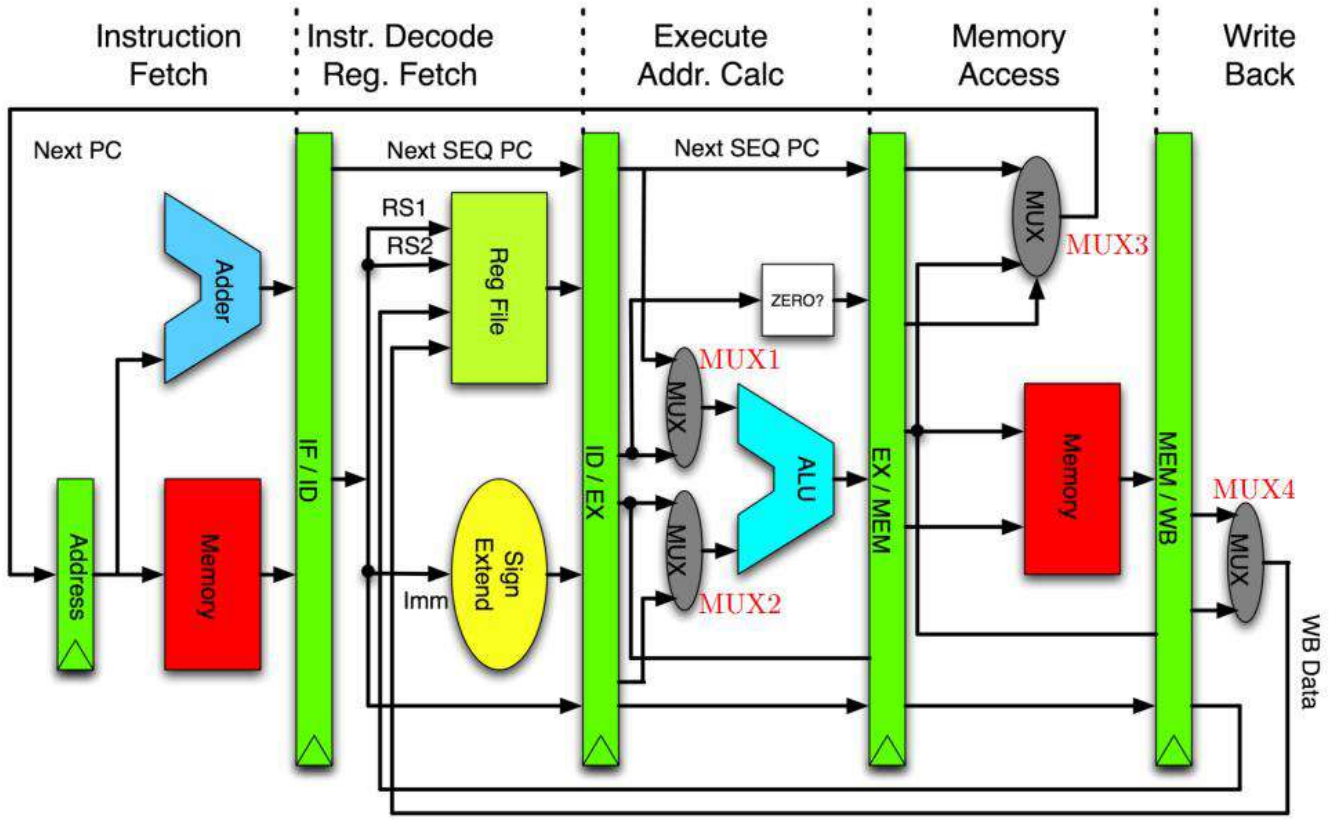
The RISC-V Programming Model — Big Picture

- Program Counter (PC) — address of next instruction
- Register file — x0 to x31
- Memory — byte-addressable, flat address space
  - Instructions (text)
  - Static data
  - Heap
  - Stack



Instruction Execution Model

- Fetch — read instruction at PC
- Decode — determine operation
- Execute — ALU / memory / branch
- Write-back — store result
- Update PC — PC+4 or branch target
- Pipelined CPUs overlap these stages



Program Counter (PC)

- Holds address of the next instruction
- Sequential: PC ← PC + 4
- Branch/Jump taken: PC ← target
- Compressed (C ext.): PC ← PC + 2
- Not a general-purpose register

Instruction Flow and Control

- Sequential: 0x1000 → 0x1004 → 0x1008
- Branch taken: 0x1000 → 0x2000 → 0x2004
- Jump: 0x1000 → 0x3000 → 0x3004
- PC-modifying instructions:
  - beq, bne, blt, bge
  - jal, jalr
  - ecall, ebreak

RISC-V Instruction Formats (Preview)

- All base instructions are 32 bits
- R-type — register-register ALU
- I-type — immediate ALU, loads
- S-type — stores
- B-type — branches
- U-type — upper immediate
- J-type — jumps

Load/Store Architecture

- Only load and store access memory
- All other instructions operate on registers
- Benefits:
  - Simpler hardware
  - Enables pipelining
  - High clock frequencies

Memory Access Instructions

- Loads: lb, lbu, lh, lhu, lw
- Stores: sb, sh, sw
- Effective address = rs1 + offset

RISC-V ALU Instructions

- Arithmetic: add, sub, addi
- Logical: and, or, xor
- Shift: sll, srl, sra

RISC-V Control Flow Instructions

- Branches: beq, bne, blt, bge, bltu, bgeu
- Jumps: jal, jalr
- jal x0, label is an unconditional jump

What Programmers See vs. Hidden State

- Visible: 32 GPRs, PC, memory, CSRs
- Hidden: pipeline registers, cache, branch predictor, physical memory

RISC-V Privilege Levels

- Level 0 — User
- Level 1 — Supervisor
- Level 3 — Machine
- Traps switch privilege levels
- This course focuses on Machine/User behavior

Programming Model — Key Takeaways

- Contract between hardware and software
- Simple, regular: 32 registers, load/store, fixed length
- PC controls instruction flow
- Flat byte-addressable memory
- CSRs provide control and status
- Stable across microarchitectures

RISC-V Registers — Introduction

- 32 general-purpose registers (RV32I)
- Each 32 bits wide: x0 – x31
- x0 hardwired to 0; writes ignored
- 5-bit encoding (2^5 = 32)

Register File Organization

- Two read ports (rs1, rs2)
- One write port (rd)
- Reads: combinational (asynchronous)
- Writes: clock-edge triggered
- x0 hardwired to ground

Register File — Timing and Behavior

- Read: address → data after propagation delay
- Write: on rising clock edge if RegWrite = 1
- x0 reads as 0; writes ignored
- Example: RegWrite=1, rd=x5, wdata=0x12345678 → x5 updated at clock edge

Register Names and ABI Conventions

- x0 zero — hardwired zero
- x1 ra — return address
- x2 sp — stack pointer
- x3 gp — global pointer
- x4 tp — thread pointer
- x5–x7 t0–t2 — temporaries
- x8 s0/fp — saved / frame pointer
- x9 s1 — saved
- x10–x17 a0–a7 — arguments / return values
- x18–x27 s2–s11 — saved
- x28–x31 t3–t6 — temporaries

Register	Description
x0	Hard-wired to 0
x1	Return address
x2	Stack pointer
x3	Global Pointer
x4	Thread Pointer
x5 to x7	Temporary Registers
x8	Frame Pointer/Saved Register
x9	Saved Register
x10 to x11	Function Arguments/Return Value
x12 to x17	Function arguments
x18 to x27	Saved Registers
x28 to x31	Temporary registers

Caller-Saved vs. Callee-Saved

- Caller-saved (t0–t6, a0–a7): caller must save before call
- Callee-saved (s0–s11, sp, ra): callee must preserve
- Reduces memory traffic — save only what is needed

Register x0 — The Zero Register

- Always reads 0
- Writes ignored
- Uses:
  - Constant 0: add x5, x0, x6 (move)
  - Discard result: add x0, x1, x2
  - Compare with zero: beq x1, x0, label
  - NOP: addi x0, x0, 0

Common Pseudo-Instructions

- nop → addi x0, x0, 0
- li rd, imm → addi rd, x0, imm
- mv rd, rs → addi rd, rs, 0
- not rd, rs → xori rd, rs, -1
- neg rd, rs → sub rd, x0, rs
- j label → jal x0, label
- ret → jalr x0, 0(ra)

Register Usage Examples

- add: add x7, x5, x6
- load-add-store: lw / lw / add / sw
- function call: a0, a1 set; jal ra, my_func; result in a0

Register File — Hardware Implementation

- 32-to-1 MUX for each read port
- 5-to-32 decoder for write port
- x0 hardwired to 0
- Two read ports, one write port
- RegWrite control signal

Why 32 Registers?

- More registers → fewer memory accesses, larger encoding
- Fewer registers → smaller encoding, more memory traffic
- RISC-V: 32 (5-bit encoding, compiler-friendly)
- Historical: MIPS 32, ARM 16/31, x86 8/16

Special Registers — Overview

- Program Counter (PC)
- Control and Status Registers (CSRs)
- Floating-point registers (F/D extension): f0–f31
- This lecture focuses on integer + CSRs

Control and Status Registers (CSRs)

- Accessed via csrrw, csrrs, csrrc, csrrwi, csrrsi, csrrci
- Example: csrrw x5, mstatus, x6
- Privileged — user mode cannot access most CSRs

Important CSRs in RISC-V

- mstatus (0x300) — machine status
- misa (0x301) — ISA and extensions
- mie (0x304) — interrupt enable
- mtvec (0x305) — trap vector base
- mepc (0x341) — exception PC
- mcause (0x342) — trap cause
- mtval (0x343) — trap value
- User-level: cycle, time, instret

mstatus — Machine Status Register

- MIE — machine interrupt enable
- MPIE — previous MIE
- MPP — previous privilege mode
- FS/XS — FPU and extension state
- Other control bits: MPRV, SUM, MXR, TVM, TW, TSR

mepc — Machine Exception Program Counter

- Holds PC of faulting instruction
- On trap: mepc ← PC
- On mret: PC ← mepc
- Enables precise exception handling and restart

mcause — Machine Cause Register

- Bit 31: 0 = exception, 1 = interrupt
- Exception codes: 0 misaligned, 2 illegal, 3 breakpoint, 8/9/11 ecall
- Interrupt codes: 3 software, 7 timer, 11 external

mtvec — Machine Trap Vector Base

- Direct mode (0): all traps → mtvec
- Vectored mode (1): trap → mtvec + 4 × cause
- Typical handler: save context, read mcause, handle, mret

CSR Access Instructions — Examples

- Read: csrrs x5, mstatus, x0
- Write: csrrw x0, mstatus, x6
- Set bits: csrrs x5, mie, x6
- Clear bits: csrrc x5, mie, x6
- Immediate: csrrwi x5, mstatus, 8

Register Summary — Key Points

- 32 GPRs, x0 hardwired to 0
- ABI names improve readability
- Caller/callee conventions reduce traffic
- CSRs provide control and status
- Register file: 2 read ports, 1 write port

Register Usage — Best Practices

- Use ABI names
- Follow calling conventions
- Use x0 for constants and discards
- Minimize memory accesses
- Use CSRs sparingly in hot code

Memory Address Space — Introduction

- Flat, byte-addressable memory model
- RV32I: 0 to 2^32 − 1 (4 GB)
- RV64I: 0 to 2^64 − 1
- Little-endian

Byte-Addressable Memory

- Each byte has its own address
- Word 0x12345678 at 0x1000:
  - 0x1000: 0x78 (LSB)
  - 0x1001: 0x56
  - 0x1002: 0x34
  - 0x1003: 0x12 (MSB)
- Little-endian = LSB at lowest address

Memory Alignment

- Byte: any address
- Halfword: multiple of 2
- Word: multiple of 4
- Doubleword: multiple of 8
- Misaligned access may trap or be emulated

Memory Address Space Layout

- 0xFFFFFFFF → Reserved / I/O
- 0x80000000 → Kernel / OS
- 0x40000000 → User program
- 0x10000000 → Stack (grows down) / Heap (grows up)
- Low addresses → .data, .bss, .text
- Exact addresses depend on platform

Program Sections

- .text — machine code (R + X)
- .rodata — constants (R)
- .data — initialized globals (R + W)
- .bss — uninitialized globals (R + W)
- .heap — dynamic allocation
- .stack — locals, frames

The Stack — Introduction

- LIFO region of memory
- Used for: return addresses, locals, arguments, saved registers
- Grows downward in RISC-V
- Managed via stack pointer sp (x2)

Stack Operations — Push and Pop

- Push: addi sp, sp, -4 ; sw x5, 0(sp)
- Pop: lw x5, 0(sp) ; addi sp, sp, 4
- Push x5, x6, x7: addi sp, sp, -12 ; sw x5, 0(sp) ; sw x6, 4(sp) ; sw x7, 8(sp)
- Pop in reverse order

Stack Pointer (sp = x2)

- Always points to top of stack
- 16-byte aligned at function boundaries (ABI)
- Callee-saved: must be restored before return
- Initialized by OS before program starts

Stack Frame — Structure

- Caller's frame (higher addresses)
- Arguments (if > 8)
- Return address (ra)
- Saved frame pointer
- Saved registers (s0–s11)
- Local variables
- Temporary space (lower addresses, at sp)

Stack Example — Function Call

- main: allocate frame, save ra
- Set a0 = 5, a1 = 10
- jal ra, add
- add: add a0, a0, a1 ; ret
- main: store result, restore ra, return

Stack Example — Nested Calls

- sum_of_squares(a, b) = square(a) + square(b)
- Prologue: save ra, s0, s1
- Save a and b in s0, s1
- Call square twice
- Add results, restore, return

Stack and Recursion

- Each recursive call creates a new frame
- fact(3) → frames for 3, 2, 1
- sp moves down with each call
- Base case stops recursion
- Unwinding restores frames and sp

Memory Address Space — Key Points

- Byte-addressable, little-endian
- Alignment required for multi-byte
- Flat address space (instructions + data)
- Sections: .text, .data, .bss, .heap, .stack
- Stack grows downward; sp 16-byte aligned

Stack Operations — Common Patterns

- Prologue: allocate frame, save ra and s-registers
- Body: use s-registers for locals
- Epilogue: restore, deallocate, return
- Local variable access via 0(sp)

Memory-Mapped I/O (Brief)

- I/O devices accessed via memory addresses
- Load/store instructions access device registers
- Example: UART data register at 0x10000000
- Simple but requires address decoding

Memory Access Examples

- Load word: la x10, data ; lw x5, 0(x10)
- Store word: la x10, result ; sw x5, 0(x10)
- Array access: slli, add, lw, addi, sw

Memory Organization — Summary Table

- Address space: flat, 2^XLEN bytes
- Addressability: byte
- Endianness: little-endian
- Word size: 32 bits (RV32I)
- Alignment: natural
- Sections: .text, .rodata, .data, .bss, .heap, .stack
- Stack growth: downward
- sp = x2; fp = x8
- Stack alignment: 16 bytes

Stack — Key Takeaways

- LIFO memory for calls and locals
- sp (x2) points to top
- Grows downward
- Frames: ra, saved regs, locals, args
- 16-byte aligned at function boundaries
- Push = decrement + store; Pop = load + increment
- Recursion naturally uses multiple frames

Summary — Programming Model

- 32 GPRs (x0–x31), PC, flat memory, CSRs
- Load/store architecture
- Fixed 32-bit instructions (base ISA)
- Simple, regular, extensible

Summary — Registers

- x0 zero, x1 ra, x2 sp, x3 gp, x4 tp
- t0–t2 and t3–t6: caller-saved
- s0–s11: callee-saved
- a0–a7: arguments / returns
- CSRs: mstatus, mepc, mcause, mtvec

Summary — Memory and Stack

- Byte-addressable, little-endian, flat
- Sections: .text, .data, .bss, .heap, .stack
- Stack: LIFO, grows down, sp = x2
- 16-byte alignment at function boundaries
- Essential for assembly and processor design

Review Questions

- Programming model vs. microarchitecture?
- How many GPRs? What is special about x0?
- Caller-saved vs. callee-saved?
- What is a CSR? Name three important ones.
- Why byte-addressable? What is little-endian?
- Describe the RISC-V stack and frame layout.
- Write assembly to push/pop x5 and x6.
- Purpose of sp? Required alignment?
- How does recursion use the stack?
- Key sections of program memory?

References and Further Reading

- Patterson & Hennessy, Computer Organization and Design (RISC-V Edition), 2020 — Chapter 2
- Hamacher, Zaky, Manjikian, Computer Organization and Embedded Systems, 6th ed.
- RISC-V International: https://riscv.org/technical/specifications/
- RISC-V Instruction Set Manual, Volumes I and II
- RISC-V Assembly Programmer's Manual

RISC-V Instruction Set Architecture

CPE2323: Computer Architecture and Organization

Data Transfer • Arithmetic/Logical/Shift • Control Flow • Machine Codes

Lecture Outline

- Data Transfer Instructions and Addressing Modes
- Arithmetic, Logical and Shift Instructions
- Control Flow and Jump Instructions
- RISC-V Machine Codes (instruction encoding)
- Worked examples and review

Section 1 — Data Transfer Instructions and Addressing Modes

- Overview of load and store instructions
- Addressing modes: register, immediate, base+offset, PC-relative
- Sign extension and zero extension
- Alignment and endianness in practice
- Worked examples

Why Load/Store Architecture Matters Here

- All memory access goes through load/store instructions
- ALU instructions only read/write registers
- Consequences:
  - Explicit data movement between memory and registers
  - Simpler pipeline — only one stage touches memory
  - Load/store instructions are the most performance-critical

Load Instructions — Overview

- lb rd, offset(rs1) — load byte (sign-extended)
- lbu rd, offset(rs1) — load byte (zero-extended)
- lh rd, offset(rs1) — load halfword (sign-extended)
- lhu rd, offset(rs1) — load halfword (zero-extended)
- lw rd, offset(rs1) — load word (32 bits)
- Effective address = rs1 + sign-extended offset

Store Instructions — Overview

- sb rs2, offset(rs1) — store byte (8 bits)
- sh rs2, offset(rs1) — store halfword (16 bits)
- sw rs2, offset(rs1) — store word (32 bits)
- Effective address = rs1 + sign-extended offset
- No sign extension for stores — only the low bits are written

Load vs. Store — Key Differences

- Loads write to rd (destination register)
- Stores read from rs2 (source register)
- Loads sign/zero-extend to 32 bits; stores truncate
- Both use the same effective address: rs1 + offset
- Loads are the critical path in many pipelines

Addressing Modes — Overview

- Register addressing: operand in a register
- Immediate addressing: constant embedded in instruction
- Base + offset: rs1 + immediate (loads/stores)
- PC-relative: PC + immediate (branches, jal)
- RISC-V deliberately keeps addressing modes simple

Mode 1 — Register Addressing

- Operand is the contents of a register
- Example: add x5, x6, x7
  - x6 and x7 are read; x5 is written
- Used by: all R-type ALU instructions
- Fastest mode — no memory access

Mode 2 — Immediate Addressing

- Operand is a constant embedded in the instruction
- Example: addi x5, x6, 10
  - 10 is the immediate; sign-extended to 32 bits
- Used by: addi, andi, ori, xori, slli, srli, srai, loads, stores, branches
- Immediate size varies by instruction format (12 or 20 bits)

Mode 3 — Base + Offset Addressing

- Effective address = rs1 + sign-extended immediate
- Used by: all loads and stores
- Example: lw x5, 8(x10)
  - EA = x10 + 8; load word into x5
- Offset range: −2048 to +2047 (12-bit signed)
- The base register can change; the offset is fixed in the instruction

Base + Offset — Practical Example

- C struct: struct Point { int x; int y; };
- Layout: x at offset 0, y at offset 4
- Assume pointer to struct in x10
- Load x: lw x5, 0(x10)
- Load y: lw x6, 4(x10)
- Store x: sw x7, 0(x10)
- Store y: sw x8, 4(x10)

Mode 4 — PC-Relative Addressing

- Effective address = PC + immediate
- Used by: branches (beq, bne, etc.) and jal
- Immediate is a multiple of 2 (13-bit signed, ×2)
- Range: ±4 KB for branches, ±1 MB for jal
- Position-independent — no absolute addresses needed

Addressing Modes — Summary Table

- Register: operand in register — add x5, x6, x7
- Immediate: operand is constant — addi x5, x6, 10
- Base + offset: rs1 + imm — lw x5, 8(x10)
- PC-relative: PC + imm — beq x5, x6, label
- RISC-V keeps these four simple modes only

Sign Extension vs. Zero Extension

- Loads: lb/lh sign-extend; lbu/lhu zero-extend
- Example: memory byte = 0xF0
  - lb → 0xFFFFFFF0 (sign-extended)
  - lbu → 0x000000F0 (zero-extended)
- Immediates: always sign-extended in I-type
- Stores: no extension — low bits are written

Alignment — Practical Implications

- Word access must be at address divisible by 4
- Halfword access must be at address divisible by 2
- Byte access — any address
- Misaligned load/store may:
  - Cause an exception (trap handler emulates)
  - Be handled by hardware (slow)
  - Return wrong data (in some implementations)
- Always align your data in assembly

Endianness in Practice

- RISC-V is little-endian
- A word at address A occupies bytes A, A+1, A+2, A+3
- Least significant byte at A, most significant at A+3
- Example: 0x12345678 at 0x1000:
  - 0x1000 = 0x78
  - 0x1001 = 0x56
  - 0x1002 = 0x34
  - 0x1003 = 0x12
- This affects byte-level manipulation and data exchange

Worked Example — Array Sum

- C: sum = 0; for (i=0; i<n; i++) sum += a[i];
- Assume: a in x10, n in x11, sum in x5
- Assembly (loop body):
  - lw x6, 0(x10) — load a[i]
  - add x5, x5, x6 — sum += a[i]
  - addi x10, x10, 4 — advance pointer
  - addi x11, x11, -1 — decrement count
  - bnez x11, loop — repeat if n ≠ 0

Section 2 — Arithmetic, Logical and Shift Instructions

- Arithmetic: add, sub, addi
- Logical: and, or, xor, andi, ori, xori
- Shift: sll, srl, sra, slli, srli, srai
- Set comparisons: slt, sltu, slti, sltiu
- Upper immediate: lui, auipc

Arithmetic Instructions

- add rd, rs1, rs2 — rd = rs1 + rs2
- sub rd, rs1, rs2 — rd = rs1 − rs2
- addi rd, rs1, imm — rd = rs1 + imm (12-bit signed)
- No subi — use addi with negative immediate
- Overflow wraps around (no trap in base ISA)

Logical Instructions

- and rd, rs1, rs2 — bitwise AND
- or rd, rs1, rs2 — bitwise OR
- xor rd, rs1, rs2 — bitwise XOR
- andi, ori, xori — immediate variants
- Common uses:
  - and with mask — extract bits
  - or with mask — set bits
  - xor with mask — toggle bits
  - xor with -1 — bitwise NOT

Shift Instructions

- sll rd, rs1, rs2 — shift left logical (fill with 0)
- srl rd, rs1, rs2 — shift right logical (fill with 0)
- sra rd, rs1, rs2 — shift right arithmetic (fill with sign bit)
- Immediate forms: slli, srli, srai (5-bit shift amount)
- Left shift = multiply by 2^n
- Right shift = divide by 2^n (sra for signed)

Shift Amount — Register vs. Immediate

- Register form: sll rd, rs1, rs2 (rs2 = shift amount)
- Immediate form: slli rd, rs1, shamt (shamt = 0–31)
- Only the low 5 bits of rs2 are used (since 32-bit words)
- Immediate form is faster — no extra register read
- Use immediate form when shift amount is a constant

Set-Less-Than Instructions

- slt rd, rs1, rs2 — rd = (rs1 < rs2) ? 1 : 0 (signed)
- sltu rd, rs1, rs2 — same, but unsigned comparison
- slti rd, rs1, imm — signed compare with immediate
- sltiu rd, rs1, imm — unsigned compare with immediate
- Used by compiler to implement <, >, <=, >=

Upper Immediate Instructions

- lui rd, imm — load upper immediate (imm << 12 into rd)
- auipc rd, imm — add upper immediate to PC
- Used to construct 32-bit constants
- Pattern: lui + addi
- Example: load 0x12345678
  - lui x5, 0x12345 — x5 = 0x12345000
  - addi x5, x5, 0x678 — x5 = 0x12345678

Constructing 32-bit Constants — The Full Picture

- Problem: immediate fields are too small (12 bits)
- Solution: lui + addi (2 instructions)
- Example: li x5, 0x12345678 expands to:
  - lui x5, 0x12346 — note the +1 adjustment
  - addi x5, x5, 0x678
- Why +1? Because addi sign-extends 0x678 → 0xFFFFF678
- Assembler does this automatically for the li pseudo-instruction

ALU Instructions — Summary Table

- Arithmetic: add, sub, addi
- Logical: and, or, xor + andi, ori, xori
- Shift: sll, srl, sra + slli, srli, srai
- Set: slt, sltu, slti, sltiu
- Upper immediate: lui, auipc
- All operate on registers only — no memory access

Worked Example — Compute (a+b)*(c-d)

- Assume a in x10, b in x11, c in x12, d in x13
- add x5, x10, x11 — x5 = a + b
- sub x6, x12, x13 — x6 = c - d
- mul x7, x5, x6 — x7 = (a+b)*(c-d)
- Note: mul requires the M extension
- Alternative without M: shift-and-add loop

Section 3 — Control Flow and Jump Instructions

- Conditional branches: beq, bne, blt, bge, bltu, bgeu
- Unconditional jumps: jal, jalr
- PC-relative addressing for branches
- Branch offset encoding (13-bit signed, ×2)
- Function calls and returns

Conditional Branch Instructions

- beq rs1, rs2, label — branch if rs1 == rs2
- bne rs1, rs2, label — branch if rs1 != rs2
- blt rs1, rs2, label — branch if rs1 < rs2 (signed)
- bge rs1, rs2, label — branch if rs1 >= rs2 (signed)
- bltu rs1, rs2, label — branch if rs1 < rs2 (unsigned)
- bgeu rs1, rs2, label — branch if rs1 >= rs2 (unsigned)

Branch Offset Encoding

- Branches use PC-relative addressing: target = PC + offset
- Offset is 13-bit signed, encoded as multiple of 2
- Range: −4096 to +4094 bytes (±4 KB)
- Offset is stored in the instruction, not the target address
- Why ×2? Because instructions are 2-byte aligned (C extension)

Computing Branch Targets — Example

- Suppose beq x5, x6, label is at address 0x1000
- Instruction encodes offset = +16 (bytes)
- Branch taken → PC = 0x1000 + 16 = 0x1010
- Branch not taken → PC = 0x1004 (fall-through)
- Note: the offset is relative to the branch instruction's address
- Assembler computes the offset automatically

Branch Range and Long Branches

- Branch range: ±4 KB (13-bit signed offset, ×2)
- What if the target is farther?
  - Use an unconditional jump (jal) with a larger range
  - Or use a branch to skip over a jal
- Pattern for long conditional branch:
  - bne rs1, rs2, skip — branch if condition false
  - jal x0, target — unconditional jump
  - skip: — continue here if condition false
- This is called a 'branch inversion'

Unconditional Jumps — jal

- jal rd, label — jump and link
  - rd = PC + 4 (return address)
  - PC = PC + offset (jump target)
- Offset is 21-bit signed, ×2 — range ±1 MB
- Used for function calls (rd = ra)
- Used for unconditional jumps (rd = x0)

Unconditional Jumps — jalr

- jalr rd, rs1, offset — jump and link register
  - rd = PC + 4 (return address)
  - PC = rs1 + offset (jump target)
- Used for function returns: jalr x0, 0(ra)
- Used for indirect calls (function pointers)
- Offset is 12-bit signed

Function Calls — Full Sequence

- Caller:
  - Set up arguments in a0–a7
  - jal ra, function
- Callee (prologue):
  - Allocate frame, save ra and s-registers
- Callee (body):
  - Compute result, place in a0
- Callee (epilogue):
  - Restore registers, deallocate frame
  - jalr x0, 0(ra) — return

If-Else — Assembly Pattern

- C: if (a < b) x = 1; else x = 2;
- Assume a in x10, b in x11, x in x12
- Pattern:
  - bge x10, x11, else — branch if a >= b
  - addi x12, x0, 1 — then: x = 1
  - jal x0, done — skip else
  - else: addi x12, x0, 2 — x = 2
  - done:
- Note the inverted condition: bge for 'if (a < b)'

Loops — Assembly Patterns

- While loop:
  - loop: bge x5, x6, done — exit if condition false
  - body
  - jal x0, loop
  - done:
- For loop (equivalent):
  - init; loop: exit-check; body; update; jump loop
- Do-while:
  - loop: body; condition; branch to loop if true
- Compiler transforms all loops to these patterns

Worked Example — Fibonacci

- C: fib(n) = fib(n-1) + fib(n-2), fib(0)=0, fib(1)=1
- Recursive version uses the stack (from Lecture 3)
- Iterative version uses only registers:
  - a = 0, b = 1
  - loop: if n == 0 return a
  - t = a + b; a = b; b = t; n--
  - jump loop
- Iterative version is much faster (no stack traffic)

Control Flow — Common Pitfalls

- Forgetting the unconditional jump after a then-block
- Using the wrong branch condition (e.g., blt when you need bge)
- Forgetting that branches are PC-relative, not absolute
- Not saving ra before a nested call
- Modifying sp without restoring it
- Branching to an unaligned address (should be 2-byte aligned)

Section 4 — RISC-V Machine Codes

- Instruction formats: R, I, S, B, U, J
- Field layout and encoding
- Decoding examples
- Why fixed-length encoding simplifies hardware

Why Fixed-Length Instructions?

- All base RISC-V instructions are exactly 32 bits
- Benefits:
  - Simple fetch — always read 4 bytes
  - Simple decode — fields at fixed positions
  - Easy pipelining — all instructions progress together
- Contrast with x86: 1–15 byte instructions, complex decode
- C extension adds 16-bit instructions for code density

Instruction Formats — Overview

- R-type: register-register ALU
- I-type: immediate ALU, loads, jalr
- S-type: stores
- B-type: branches
- U-type: upper immediate (lui, auipc)
- J-type: jumps (jal)
- Each format places fields at fixed positions

R-Type Format

- Layout: funct7 (7) | rs2 (5) | rs1 (5) | funct3 (3) | rd (5) | opcode (7)
- Used by: add, sub, and, or, xor, sll, srl, sra, slt, sltu
- funct3 and funct7 together select the operation
- Example: add x5, x6, x7
  - opcode = 0110011, funct3 = 000, funct7 = 0000000
  - rd = 5, rs1 = 6, rs2 = 7

I-Type Format

- Layout: imm[11:0] (12) | rs1 (5) | funct3 (3) | rd (5) | opcode (7)
- Used by: addi, andi, ori, xori, slti, sltiu, loads, jalr
- Immediate is 12-bit sign-extended
- Loads use funct3 to select size/sign: lb=000, lh=001, lw=010, lbu=100, lhu=101
- Example: addi x5, x6, 10
  - opcode = 0010011, funct3 = 000, imm = 10, rd = 5, rs1 = 6

S-Type Format

- Layout: imm[11:5] (7) | rs2 (5) | rs1 (5) | funct3 (3) | imm[4:0] (5) | opcode (7)
- Used by: sb, sh, sw
- Immediate is split across two fields (imm[11:5] and imm[4:0])
- Why split? To keep rs1 and rs2 in the same positions as R-type
- Example: sw x5, 8(x10)
  - opcode = 0100011, funct3 = 010, imm = 8, rs1 = 10, rs2 = 5

B-Type Format

- Layout: imm[12|10:5] (7) | rs2 (5) | rs1 (5) | funct3 (3) | imm[4:1|11] (5) | opcode (7)
- Used by: beq, bne, blt, bge, bltu, bgeu
- Immediate is 13-bit signed, ×2 (always even)
- Bits are scrambled — this is the most confusing format
- Example: beq x5, x6, +16
  - opcode = 1100011, funct3 = 000, imm = 16

U-Type Format

- Layout: imm[31:12] (20) | rd (5) | opcode (7)
- Used by: lui, auipc
- Immediate is 20 bits, placed in the upper 20 bits of the result
- Lower 12 bits are zero
- Example: lui x5, 0x12345
  - opcode = 0110111, imm = 0x12345, rd = 5
  - Result: x5 = 0x12345000

J-Type Format

- Layout: imm[20|10:1|11|19:12] (20) | rd (5) | opcode (7)
- Used by: jal
- Immediate is 21-bit signed, ×2
- Bits are scrambled like B-type, but with more bits
- Example: jal ra, +1024
  - opcode = 1101111, imm = 1024, rd = 1 (ra)

Decoding Example — add x5, x6, x7

- Machine code: 0x007302B3
- Binary: 0000000 00111 00110 000 00101 0110011
- Decode:
  - opcode = 0110011 → R-type
  - funct3 = 000, funct7 = 0000000 → add
  - rd = 00101 = 5, rs1 = 00110 = 6, rs2 = 00111 = 7
- Result: add x5, x6, x7

Decoding Example — lw x5, 8(x10)

- Machine code: 0x00852283
- Binary: 000000001000 01010 010 00101 0000011
- Decode:
  - opcode = 0000011 → load
  - funct3 = 010 → lw
  - rd = 00101 = 5, rs1 = 01010 = 10, imm = 8
- Result: lw x5, 8(x10)

Decoding Example — beq x5, x6, +16

- Machine code: 0x00628863
- Binary: 0000000 00110 00101 000 01000 1100011
- Decode:
  - opcode = 1100011 → branch
  - funct3 = 000 → beq
  - rs1 = 00101 = 5, rs2 = 00110 = 6
  - imm: bit 12=0, bits 10:5=000000, bits 4:1=1000, bit 11=0 → 0x10 = 16
- Result: beq x5, x6, +16

Encoding Summary — Field Positions

- opcode: always bits 0–6 (7 bits)
- rd: always bits 7–11 (5 bits) — except S-type and B-type
- funct3: always bits 12–14 (3 bits)
- rs1: always bits 15–19 (5 bits)
- rs2: always bits 20–24 (5 bits) — except I-type, U-type, J-type
- funct7: always bits 25–31 (7 bits) — R-type only
- Consistent positions = simple decoder

Decoding in Hardware — The Immediate Generator

- Separate immediate generator for each format
- R-type: no immediate
- I-type: sign-extend 12 bits
- S-type: concatenate two fields, sign-extend
- B-type: unscramble, sign-extend, shift left by 1
- U-type: place 20 bits in upper position
- J-type: unscramble, sign-extend, shift left by 1

Worked Example — Encode addi x5, x6, -10

- Format: I-type
- opcode = 0010011 (addi)
- rd = 5 = 00101
- funct3 = 000
- rs1 = 6 = 00110
- imm = -10 = 111111110110 (12-bit two's complement)
- Binary: 111111110110 00110 000 00101 0010011
- Hex: 0xFF628293

Summary — Data Transfer & Addressing

- Loads: lb, lbu, lh, lhu, lw
- Stores: sb, sh, sw
- Addressing modes: register, immediate, base+offset, PC-relative
- Effective address = rs1 + sign-extended offset
- Sign vs. zero extension matters for loads
- Alignment required for multi-byte access

Summary — ALU Instructions

- Arithmetic: add, sub, addi
- Logical: and, or, xor + immediate variants
- Shift: sll, srl, sra + immediate variants
- Set: slt, sltu, slti, sltiu
- Upper immediate: lui, auipc
- All operate on registers only

Summary — Control Flow

- Branches: beq, bne, blt, bge, bltu, bgeu
- Jumps: jal (PC-relative), jalr (register)
- PC-relative addressing for branches
- Branch offset: 13-bit signed, ×2 (±4 KB)
- jal offset: 21-bit signed, ×2 (±1 MB)
- Function calls: jal ra, func; return: jalr x0, 0(ra)

Summary — Machine Codes

- Six formats: R, I, S, B, U, J
- All 32 bits, fixed length
- Opcode always bits 0–6
- rd, rs1, rs2 at consistent positions
- Immediate fields differ by format
- Decoder can start before format is known

Review Questions

- What is the difference between lb and lbu?
- Write assembly for 'a = b + c - d'.
- What is the effective address for lw x5, 8(x10) if x10 = 0x1000?
- Difference between srl and sra?
- How do you compute x5 = x6 * 16 without mul?
- Write assembly for 'if (a < b) x = 1; else x = 2;'
- Difference between jal and jalr?
- What is the branch offset range?
- Decode 0x007302B3.
- Encode addi x5, x6, 100.

References and Further Reading

- Patterson & Hennessy, Computer Organization and Design (RISC-V Edition), 2020 — Chapter 2
- RISC-V Instruction Set Manual, Volume I: Unprivileged ISA
- RISC-V Assembly Programmer's Manual
- RISC-V Online Simulator: https://riscv.org/software-status/
- RARS Simulator for practice

Week 5: Arithmetic, Logical, and Shift Instructions

CPE2323: Computer Architecture and Organization
Capital University of Science and Technology

Week 8 Learning Objectives

- By the end of this week, you should be able to:
  - Use all RISC-V arithmetic instructions
  - Use logical and shift instructions
  - Apply set-less-than instructions
  - Use lui/auipc to load 32-bit constants
  - Use M extension multiply and divide
  - Pass the midterm exam

Arithmetic Instructions

Instruction	Format	Operation
add rd, rs1, rs2	R	rd = rs1 + rs2
sub rd, rs1, rs2	R	rd = rs1 − rs2
addi rd, rs1, imm	I	rd = rs1 + sign_ext(imm)
addu/subu	R	Same but no overflow trap
addiu	I	addi with no overflow trap

Logical and Shift Instructions

Instruction	Format	Operation
and rd, rs1, rs2	R	rd = rs1 & rs2
or rd, rs1, rs2	R	rd = rs1 | rs2
xor rd, rs1, rs2	R	rd = rs1 ^ rs2
andi/ori/xori	I	Same with immediate
sll rd, rs1, rs2	R	rd = rs1 << rs2[4:0]
srl rd, rs1, rs2	R	Logical right shift
sra rd, rs1, rs2	R	Arithmetic right shift
slli/srli/srai	I	Shift by immediate

Set Instructions

Instruction	Format	Operation
slt rd, rs1, rs2	R	rd = (rs1 < rs2) ? 1 : 0 (signed)
sltu rd, rs1, rs2	R	rd = (rs1 < rs2) ? 1 : 0 (unsigned)
slti rd, rs1, imm	I	signed, immediate
sltiu rd, rs1, imm	I	unsigned, immediate

Upper Immediate and Multiply

```
Upper immediate:
  lui   rd, imm20    # rd = imm20 << 12
  auipc rd, imm20    # rd = PC + (imm20 << 12)

M extension (integer multiply/divide):
  mul    rd, rs1, rs2    # lower 32 bits of product
  mulh   rd, rs1, rs2    # upper 32 (signed × signed)
  mulhu  rd, rs1, rs2    # upper 32 (unsigned × unsigned)
  mulhsu rd, rs1, rs2    # upper 32 (signed × unsigned)
  div    rd, rs1, rs2    # signed divide
  divu   rd, rs1, rs2    # unsigned divide
  rem    rd, rs1, rs2    # signed remainder
  remu   rd, rs1, rs2    # unsigned remainder
```

Midterm Preparation

- Midterm covers Weeks 1-7:
  - Week 1: Introduction, Von Neumann, ISA basics
  - Week 2: Number systems, two's complement, IEEE 754
  - Week 3: Floating-point, compilation flow
  - Week 4: RISC-V philosophy, families, extensions
  - Week 5: Registers, ABI, stack
  - Week 6: CSRs, traps, complete programs
  - Week 7: Instruction formats, load/store
- Format: 2 hours, closed book, mixed multiple choice and problems

Week 8 Summary

- Arithmetic: add, sub, addi
- Logical: and, or, xor
- Shift: sll, srl, sra
- Set: slt, sltu
- Upper: lui, auipc
- M extension: mul, div, rem
- Midterm exam at end of week

Questions?

Thank You!

Week 6: Control Flow and Machine Codes

CPE2323: Computer Architecture and Organization
Capital University of Science and Technology

Week 9 Learning Objectives

- By the end of this week, you should be able to:
  - Use all RISC-V branch instructions
  - Use jal and jalr
  - Translate if-else and loops to assembly
  - Encode any RISC-V instruction
  - Read hex dumps and disassemble

Branch Instructions

Instruction	Condition	Type
beq rs1, rs2, offset	rs1 == rs2	equality
bne rs1, rs2, offset	rs1 != rs2	inequality
blt rs1, rs2, offset	rs1 < rs2 (signed)	signed
bge rs1, rs2, offset	rs1 >= rs2 (signed)	signed
bltu rs1, rs2, offset	rs1 < rs2 (unsigned)	unsigned
bgeu rs1, rs2, offset	rs1 >= rs2 (unsigned)	unsigned

Jump Instructions

Instruction	Format	Operation
jal rd, offset	J	rd = PC+4; PC = PC + offset
jalr rd, rs1, imm	I	rd = PC+4; PC = rs1 + imm
jal offset (rd=ra)	J	Function call
jal x0, offset	J	Unconditional jump
j offset	J (pseudo)	Same as jal x0, offset
ret	I (pseudo)	jalr x0, ra, 0
jr rs	I (pseudo)	jalr x0, rs, 0

If-Else in Assembly

```
C:                          Assembly:
if (a == b)                 # a in x5, b in x6
  c = a + 1;                bne   x5, x6, else
else                        addi  x7, x5, 1
  c = a - 1;                j     end
                            else:
                            addi  x7, x5, -1
                            end:
```

Note: we branch on the OPPOSITE condition
to skip the 'then' block when false.

Loops in Assembly

```
C:                          Assembly:
sum = 0;                    li    x5, 0       # sum
for (i=0; i<n; i++)         li    x6, 0       # i
  sum += a[i];              li    x7, n       # n
loop:
                            bge   x6, x7, end
                            slli  x8, x6, 2   # i*4
                            add   x8, x8, a0  # &a[i]
                            lw    x9, 0(x8)
                            add   x5, x5, x9  # sum
                            addi  x6, x6, 1   # i++
                            j     loop
end:
```

Encoding add x5, x6, x7

```
Format: R-type
  funct7 (7) | rs2 (5) | rs1 (5) | funct3 (3) | rd (5) | opcode (7)

Values:
  funct7 = 0000000
  rs2    = 00111  (x7)
  rs1    = 00110  (x6)
  funct3 = 000    (add)
  rd     = 00101  (x5)
  opcode = 0110011 (OP)

Binary:  0000000 00111 00110 000 00101 0110011
Hex:     0x007302B3
```

Reading Hex Dumps

- Memory dump of a program (each word = one instruction):
-
- 0x00000000: 0x00A30293 # addi x5, x6, 10
- 0x00000004: 0x007302B3 # add x5, x6, x7
- 0x00000008: 0x00832283 # lw x5, 8(x6)
- 0x0000000C: 0x00732423 # sw x7, 8(x6)
- 0x00000010: 0x00628A63 # beq x5, x6, +16
- 0x00000014: 0x008000EF # jal ra, +8
-
- Little-endian storage: bytes reversed within word
- Simulator's 'disassemble' does this automatically

Week 9 Summary

- Branches: beq, bne, blt, bge, bltu, bgeu
- Jumps: jal, jalr
- If-else: branch on opposite + jump
- Loops: label + branch + jump
- Encoding: R, I, S, B, U, J formats
- Reading hex dumps requires understanding encoding

Questions?

Thank You!

Week 7: Single-Cycle Processor – Data Path

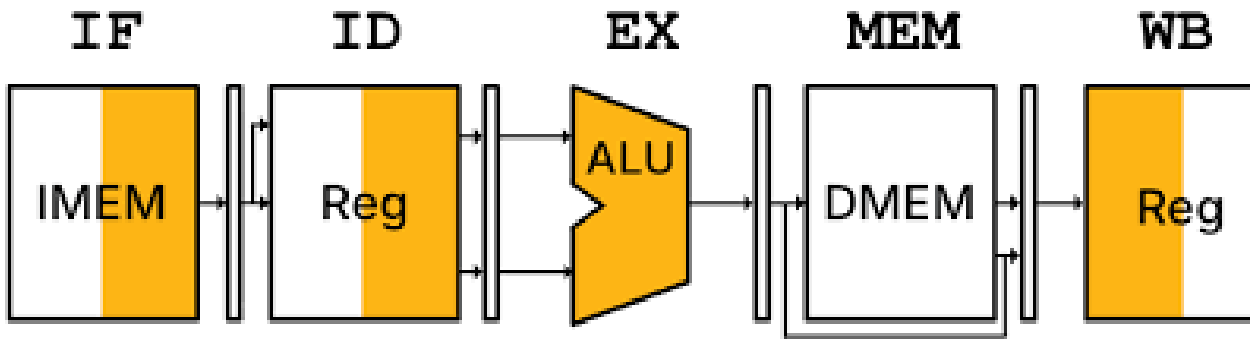
CPE2323: Computer Architecture and Organization
Capital University of Science and Technology

Week 7 Learning Objectives

- By the end of this week, you should be able to:
  - Explain single-cycle design philosophy
  - Describe each data path component
  - Trace R-type, load, store through the data path
  - Trace branch, jal, jalr through the data path
  - Identify multiplexer placement

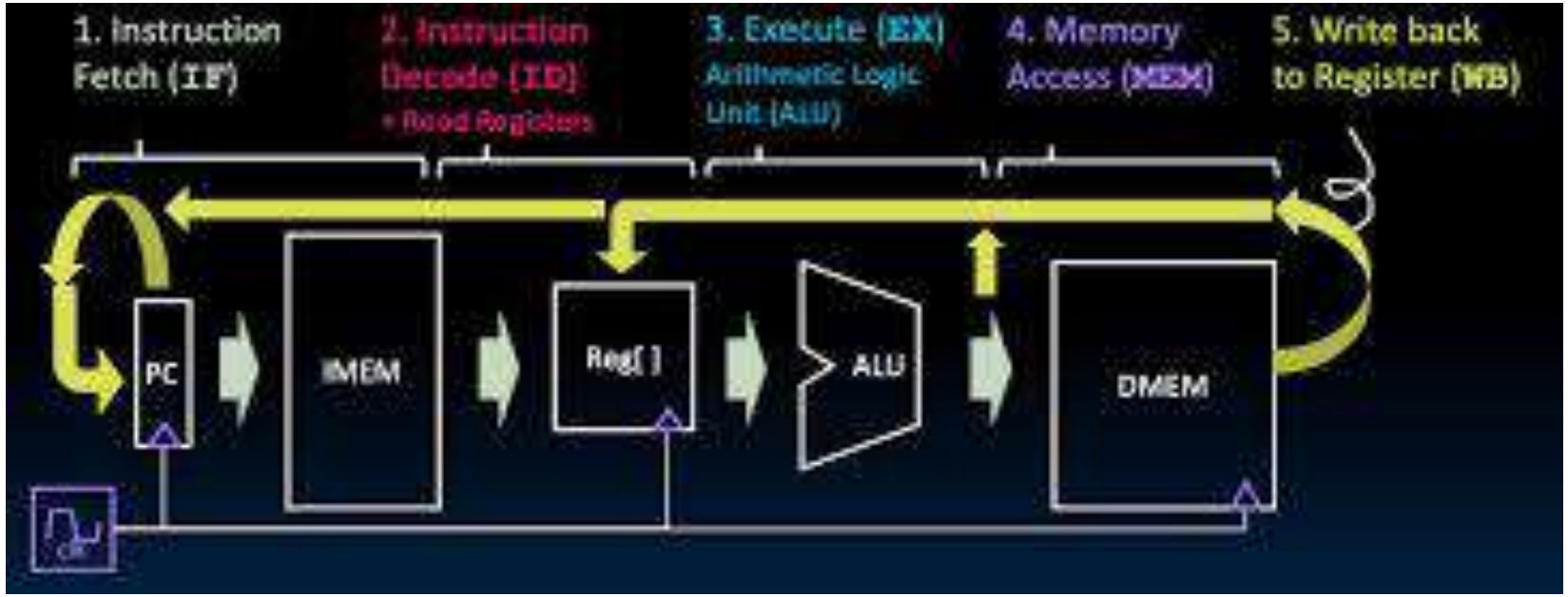
Single Cycle Design Philosophy

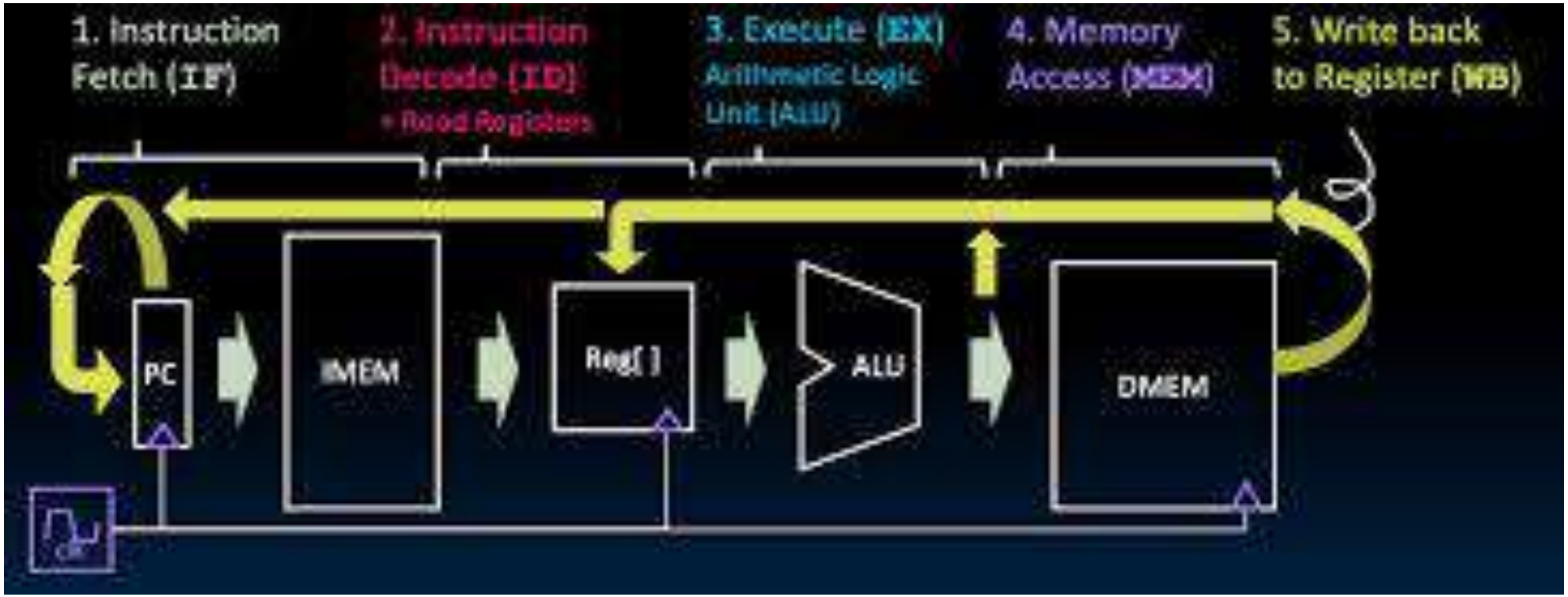
- One instruction per clock cycle
- Clock period = longest instruction delay
  - Load: PC → IMEM → RegFile → ALU → DMEM → RegFile
- Simple control: combinational logic
- No intermediate state (no pipeline registers)
- Advantages: simple, easy to verify
- Disadvantages: slow, hardware underutilized



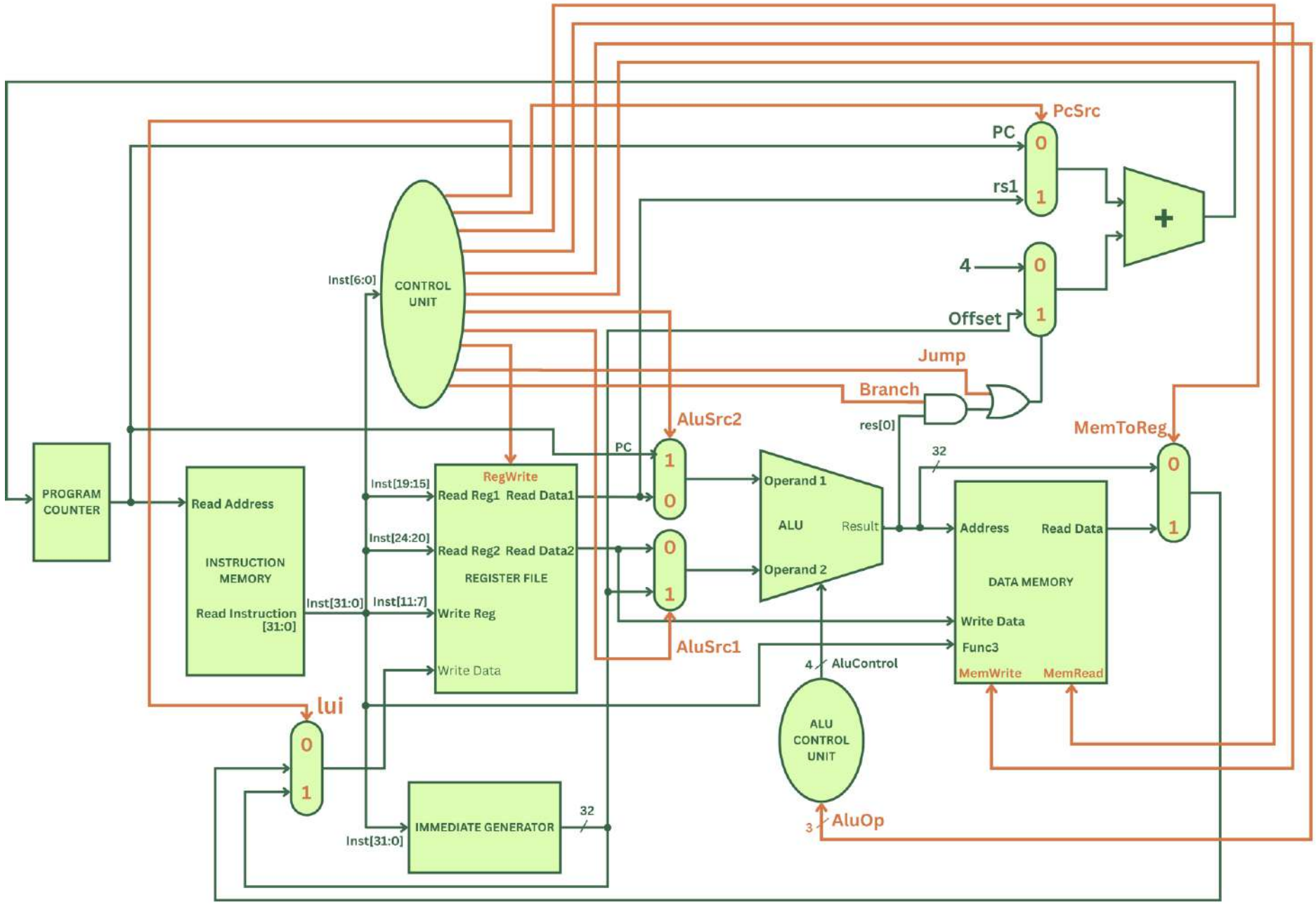
Single Cycle Design Philosophy

- One instruction per clock cycle
- Clock period = longest instruction delay
  - Load: PC → IMEM → RegFile → ALU → DMEM → RegFile
- Simple control: combinational logic
- No intermediate state (no pipeline registers)
- Advantages: simple, easy to verify
- Disadvantages: slow, hardware underutilized

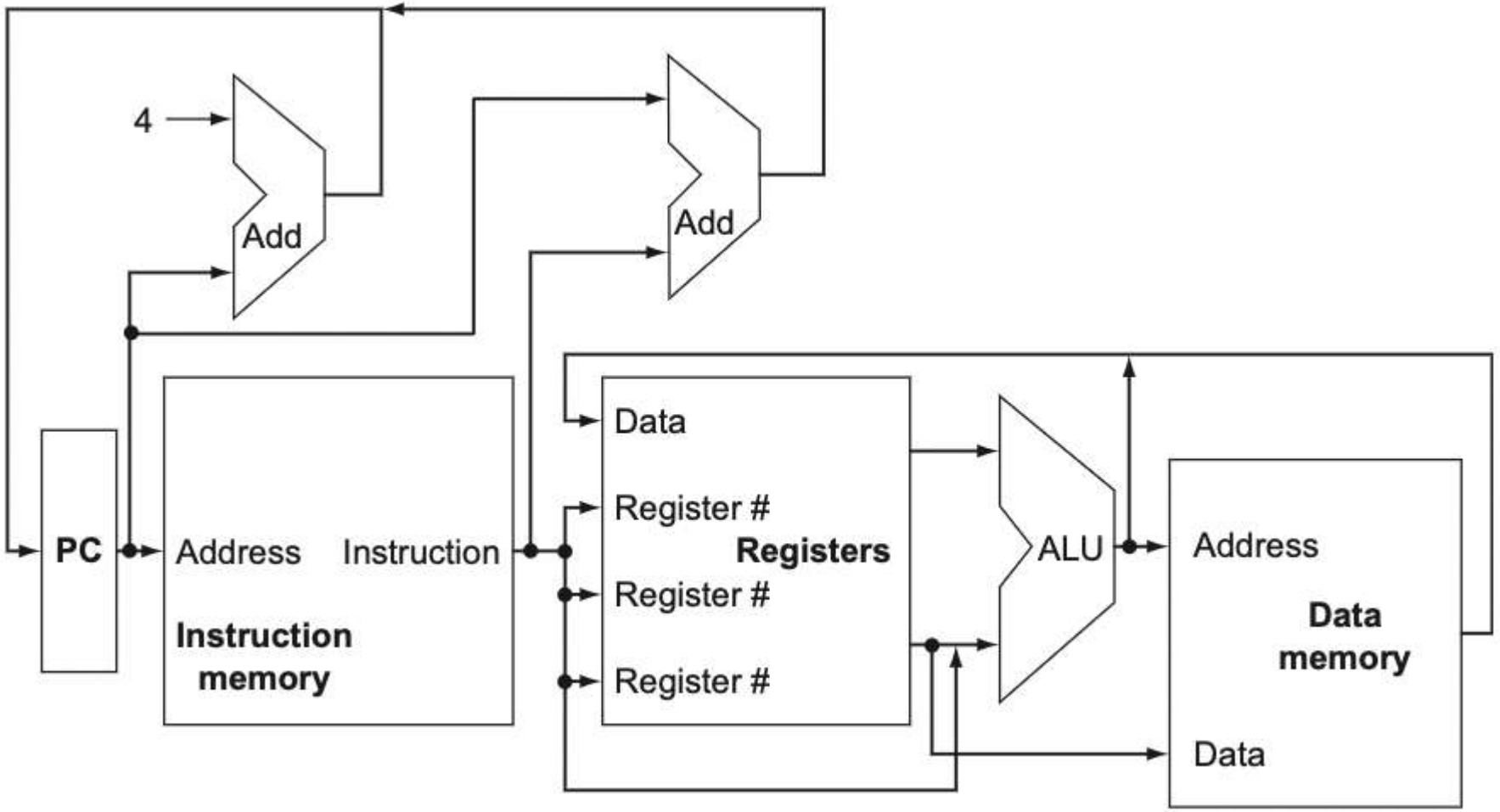



1. Instruction Fetch (IF)
2. Instruction Decode (ID)
3. Execute (EX)
Arithmetic Logic Unit (ALU)
4. Memory Access (MEM)
5. Write back to Register (WB)
PC
IMEM
Reg[ ]
ALU
DMEM

RV32I SINGLE-CYCLE PROCESSOR DATAPATH

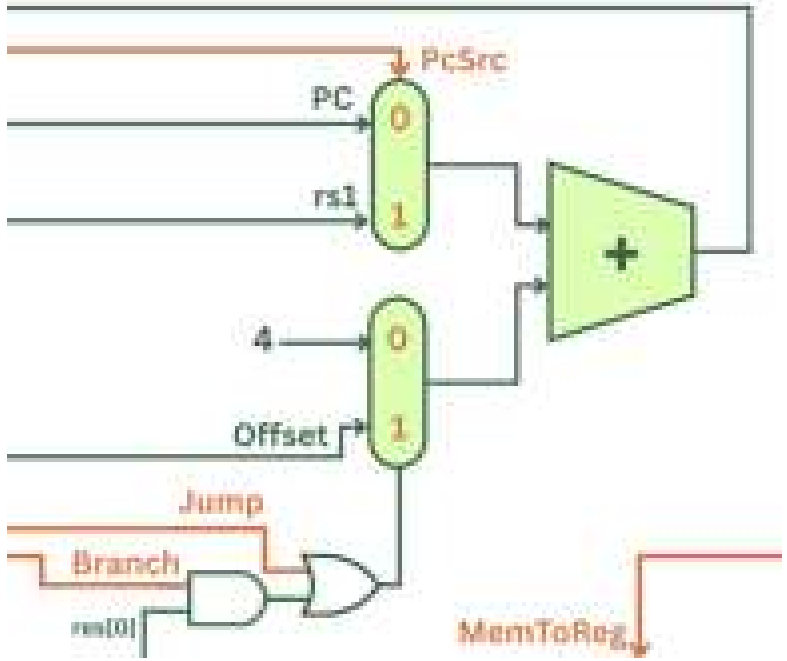


High-Level Data Path



The Program Counter (PC)

- A register holding the address of the current instruction
- Updates at each clock edge
- Two possible next values:
  - PC + 4 (sequential execution)
  - Branch/jump target
- Branch target = PC + (sign-extended offset)
- Multiplexer selects between them

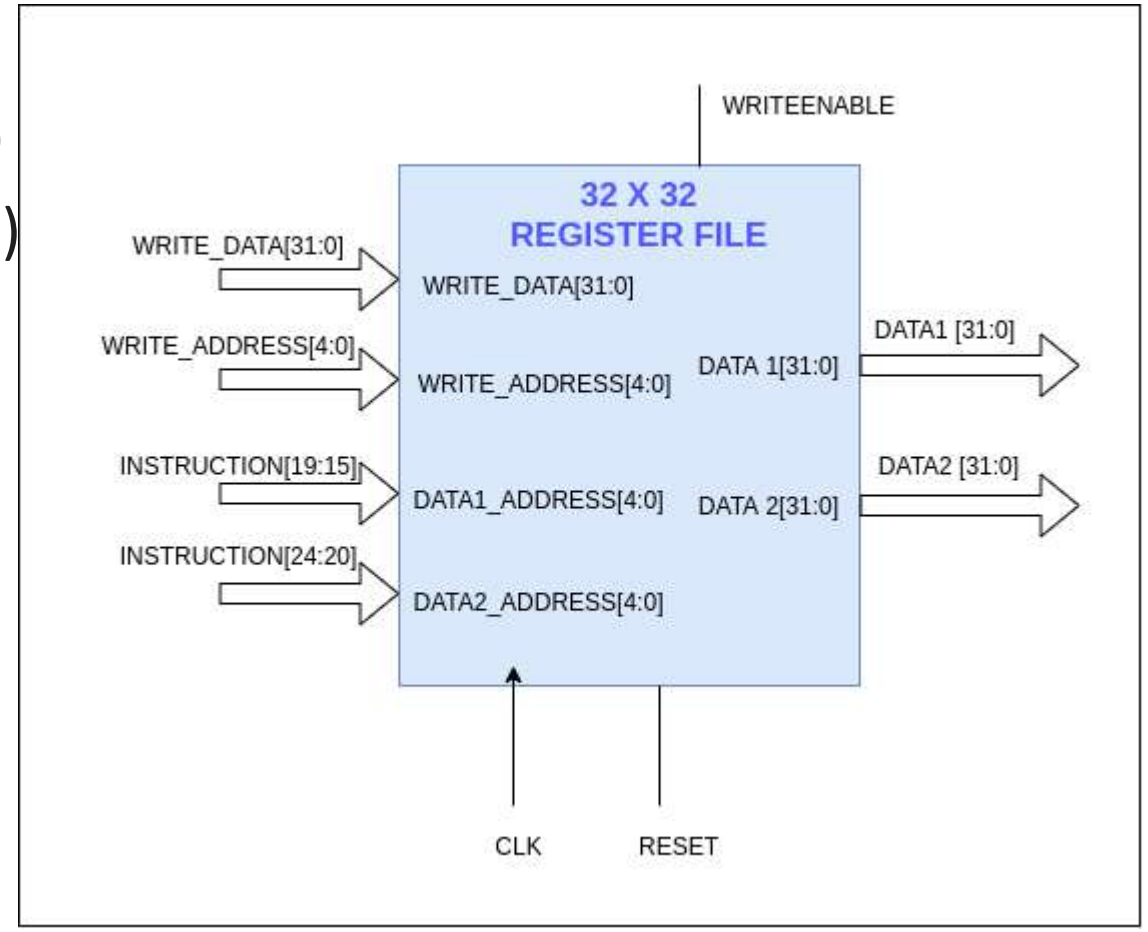


Instruction Memory

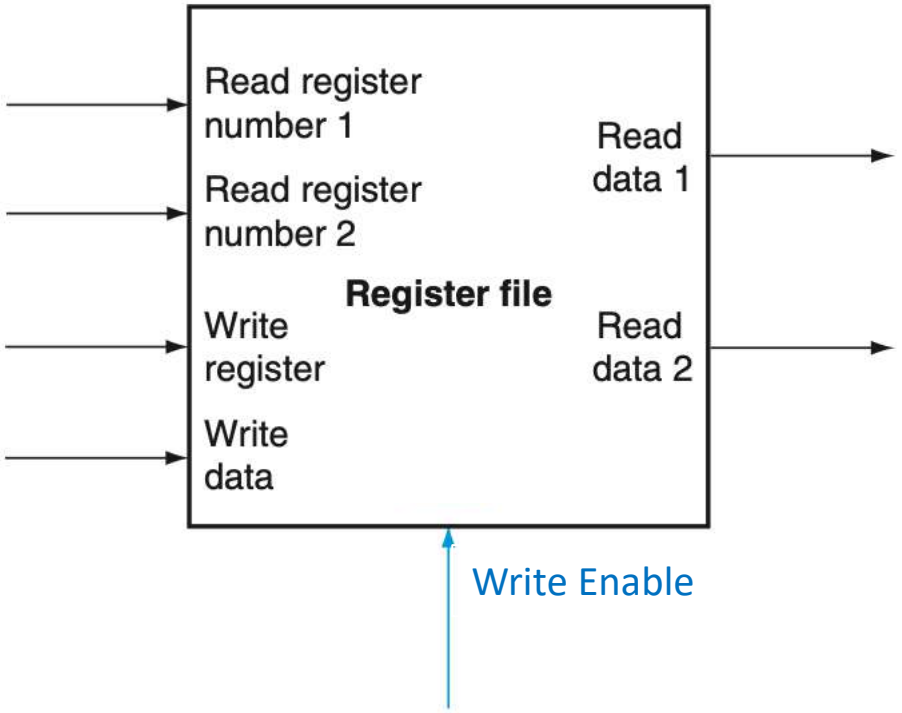
- Read-only memory holding the program
- Input: address (from PC)
- Output: 32-bit instruction
- Combinational read (no clock needed for read)
- In a real processor:
  - Usually an instruction cache
  - May be separate from data (Harvard) or shared (Von Neumann)
- For our design: separate instruction memory

Register File

- 32 registers, each 32 bits
- Two read ports, one write port
- Read: combinational (addresses in, data out)
- Write: clocked (on rising edge, if RegWrite=1)
- x0 is hardwired to 0:
  - Reading x0 returns 0
  - Writing x0 is ignored



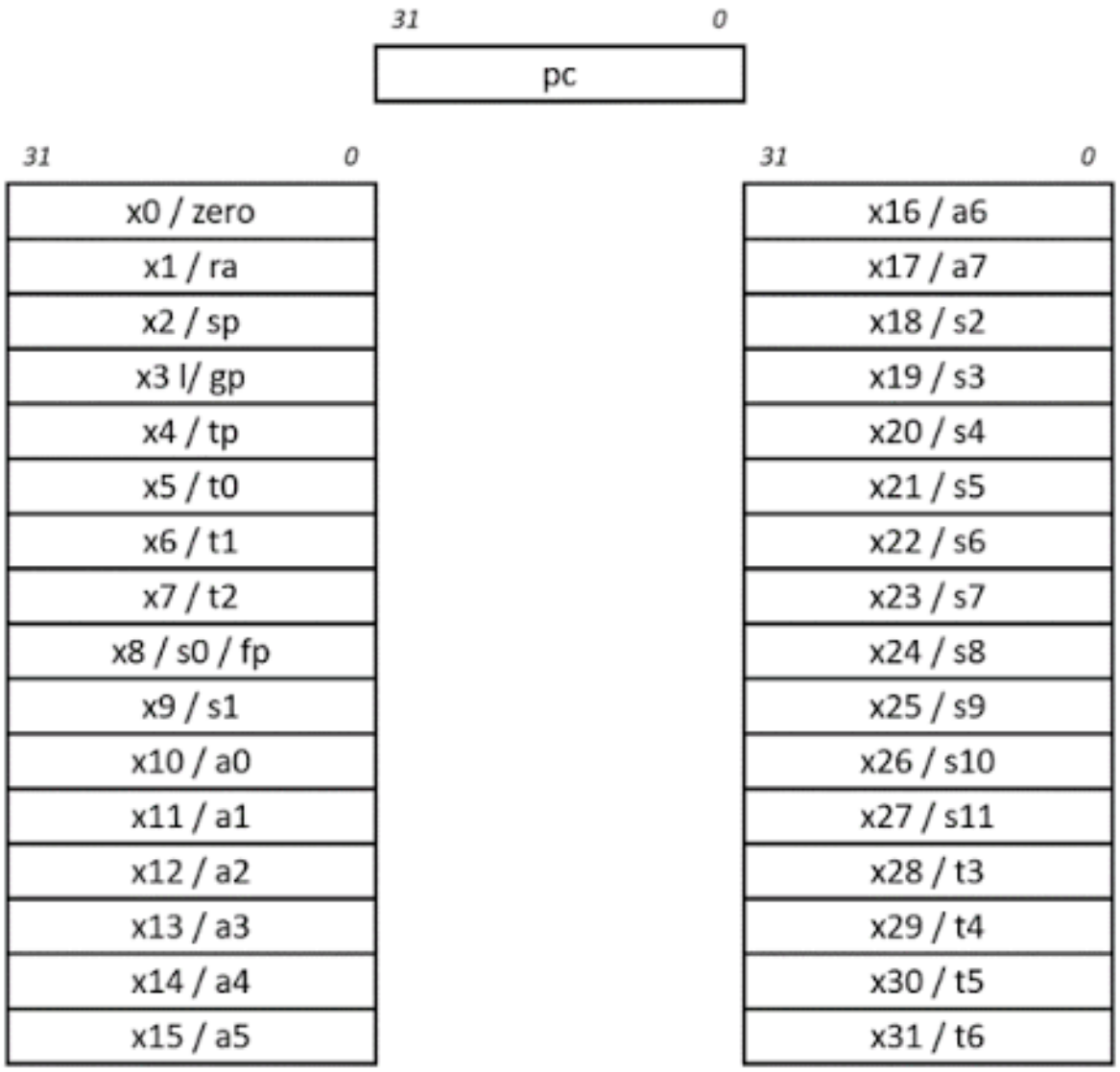
Register File



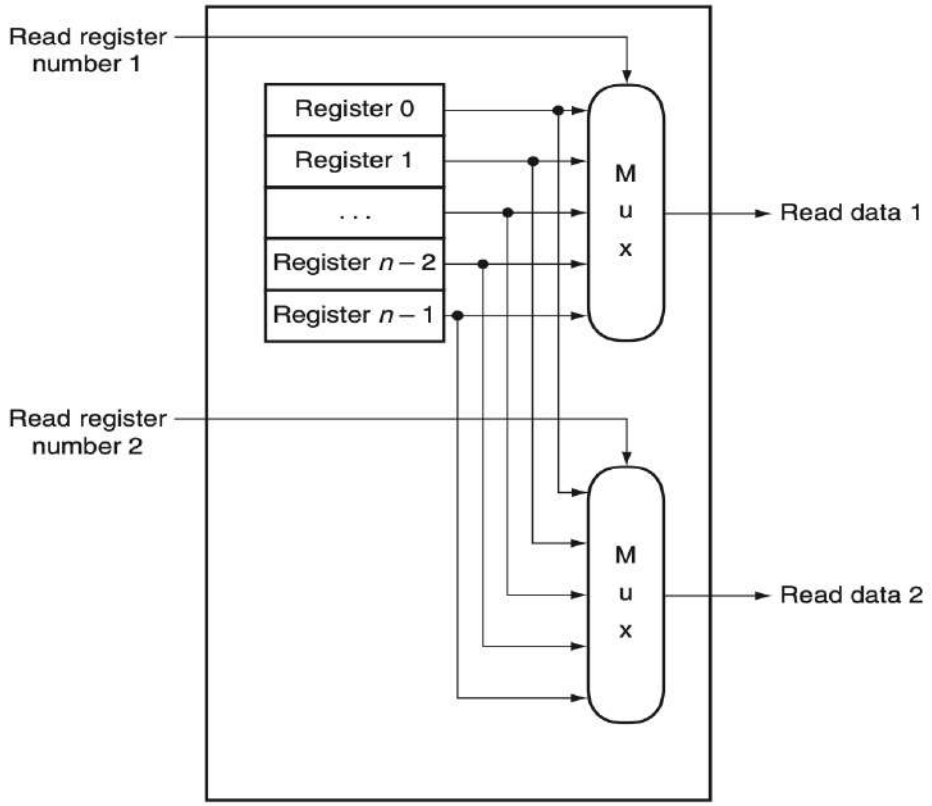
A register file with two read ports and one write port

Ref: Harris and Harris Book: *Digital Design and Computer Architecture: RISC-V Edition*

Register File



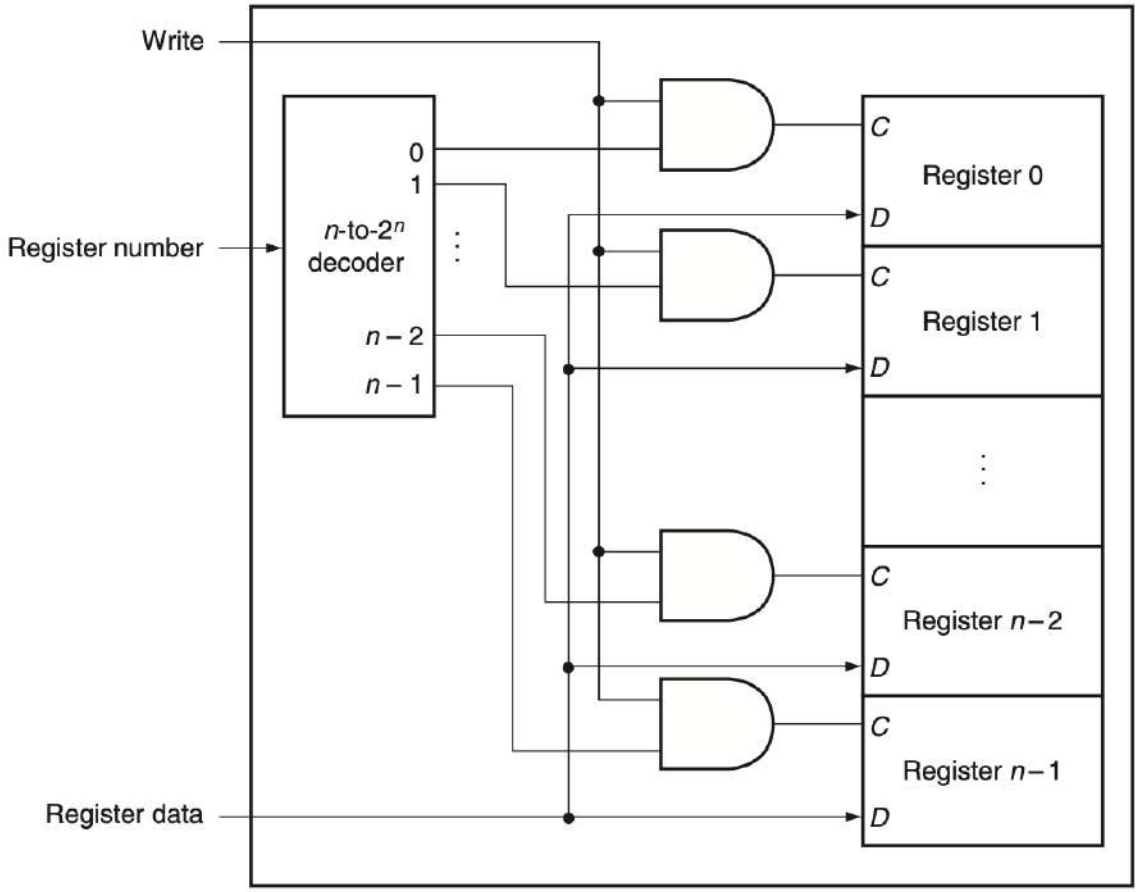
Register File



The implementation of two read ports for a register file with *n* registers can be done with a pair of n-to-1 multiplexors

Ref: Harris and Harris Book: *Digital Design and Computer Architecture: RISC-V Edition*

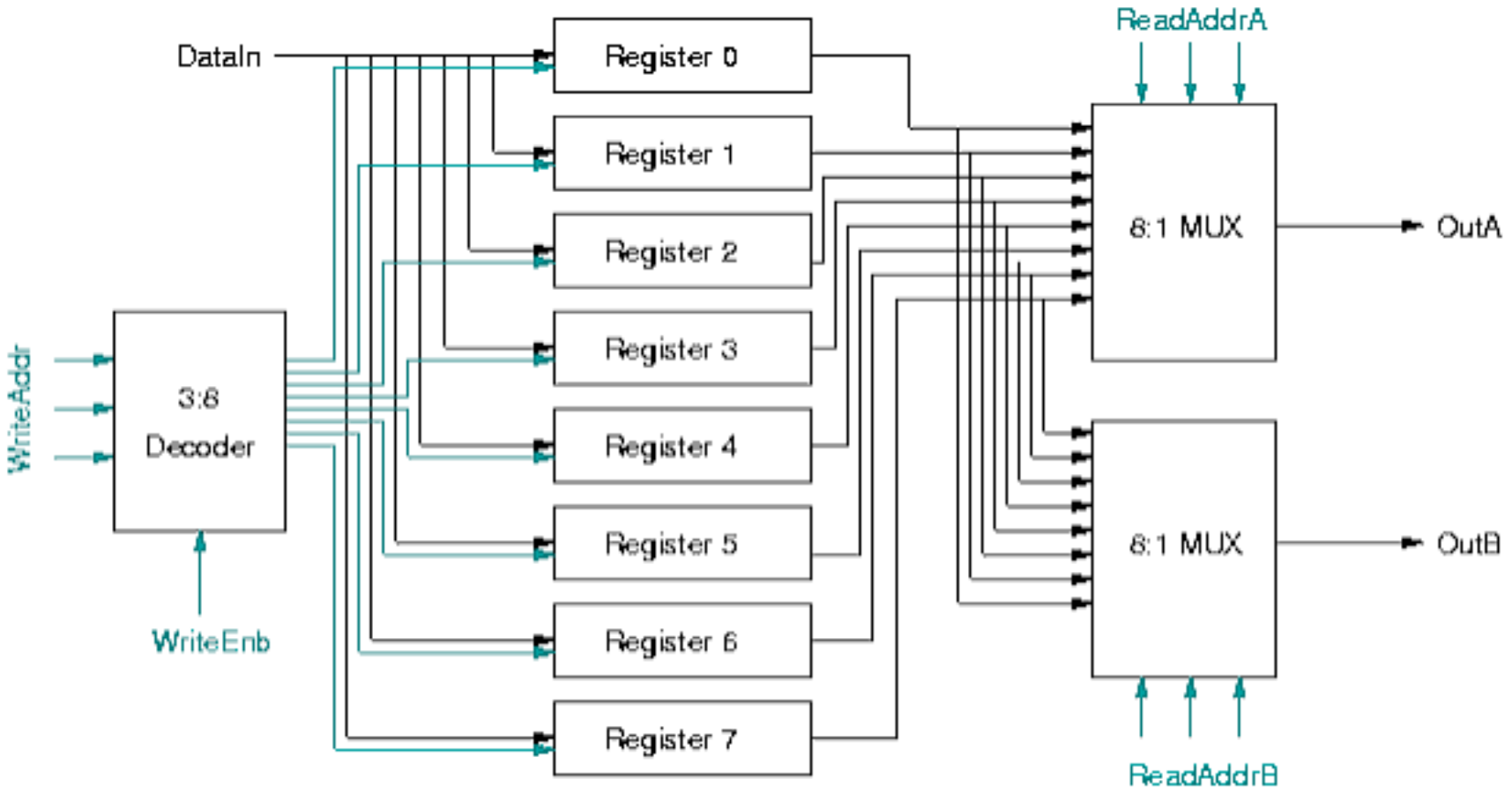
Register File



The write port for a register file is implemented with a decoder that is used with the write signal to generate the *C* input to the registers

Ref: Harris and Harris Book: *Digital Design and Computer Architecture: RISC-V Edition*

Register File Architecture (8-bit)



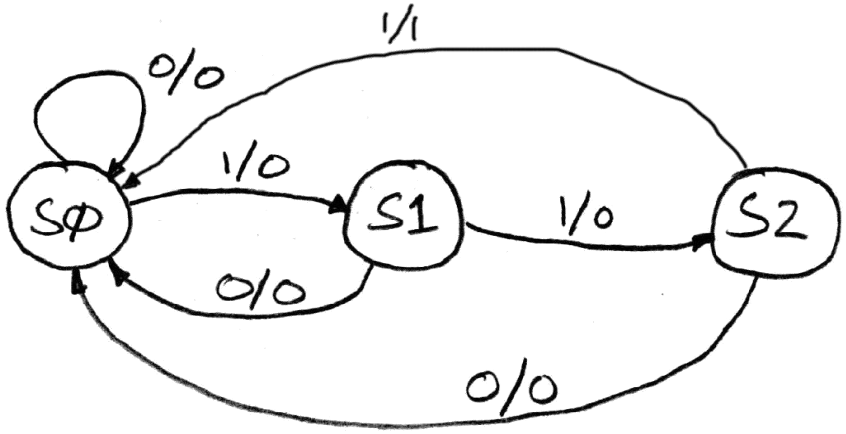
Finite State Machines (FSMs)

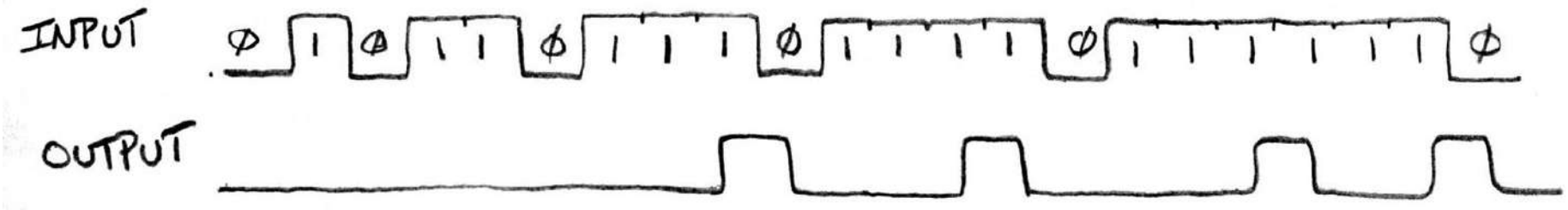
- How do we represent sequential logic?
  - Truth tables could account for history
  - We could do boolean logic with prior state as a variable
- We can also use a new representation: Finite State Machines

Finite State Machines (FSMs)

- A convenient way to conceptualize computation over time
- Function can be represented with a state transition diagram
- With combinational logic and registers, any FSM can be implemented in hardware!

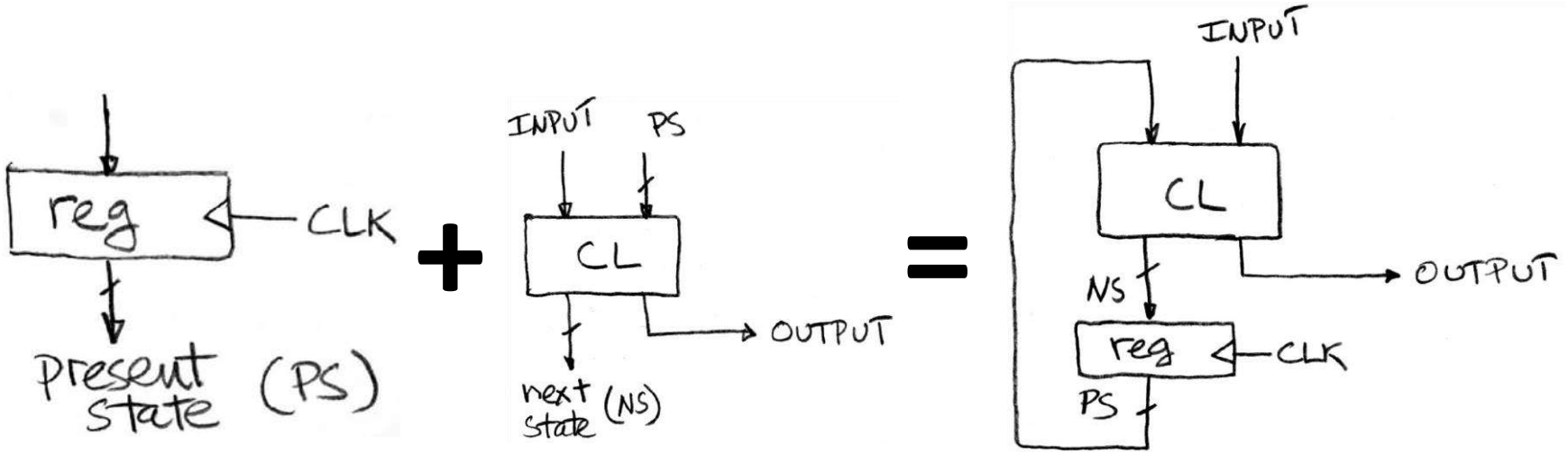
States: S0, S1, S2
Initial State: S0
Transitions of form:
input/output



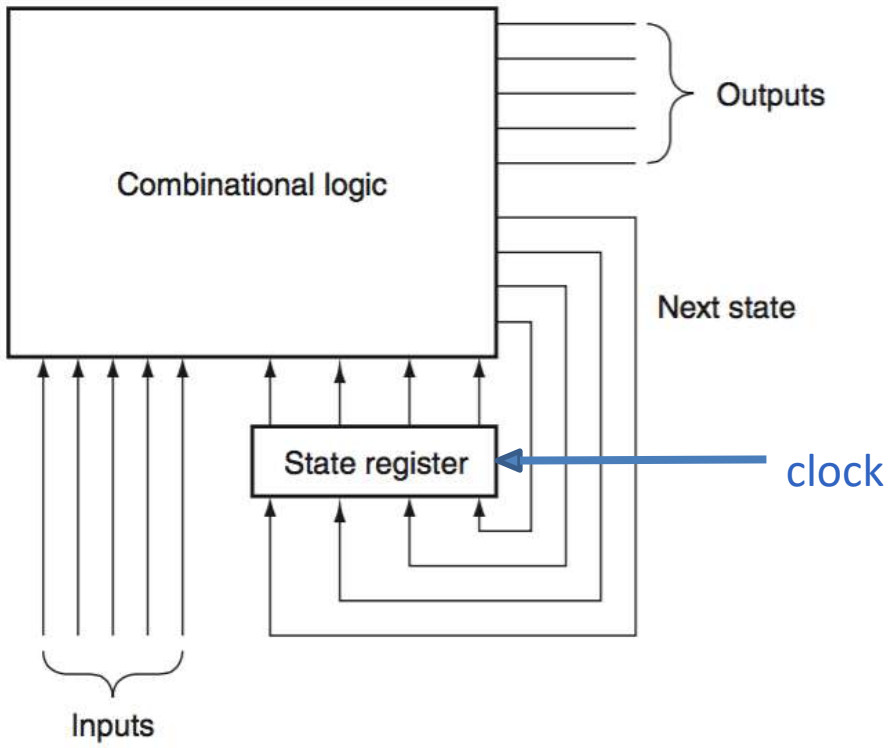


Hardware Implementation of FSM

- Register holds a representation of the FSM's state
  - Must assign a unique bit pattern for each state
  - Output is present/current state (PS/CS)
  - Input is next state (NS)
- Combinational Logic implements transition function (state transitions + output)



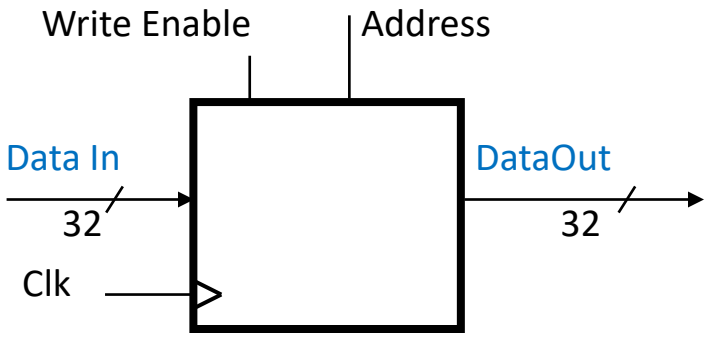
Hardware Implementation of FSM



Storage Elements

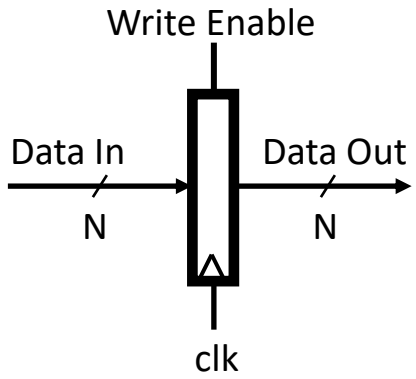
Storage Element: Idealized Memory

- Memory (idealized)
  - One input bus: Data In
  - One output bus: Data Out
- Memory word is found by:
  - Address selects the word to put on Data Out
  - Write Enable = 1: address selects the memory word to be written via the Data In bus
- Clock input (CLK)
  - CLK input is a factor ONLY during write operation
  - During read operation, behaves as a combinational logic block: Address valid ⇒ Data Out valid after "access time"



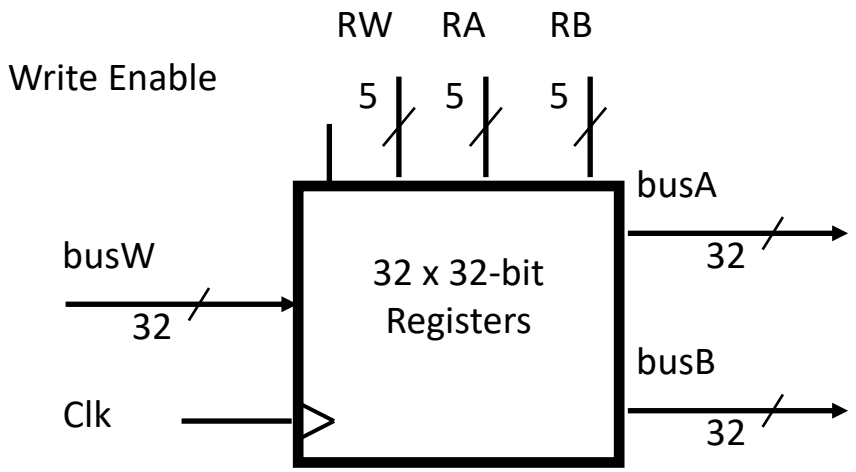
Storage Element: Register (Building Block)

- Similar to D Flip Flop except
  - N-bit input and output
  - Write Enable input
- Write Enable:
  - Negated (or deasserted) (0): Data Out will not change
  - Asserted (1): Data Out will become Data In on rising edge of clock



Storage Element: Register File

- Register File consists of 32 registers:
  - Two 32-bit **output busses**:
    busA and busB
  - One 32-bit **input bus**: busW
- Register is selected by:
  - RA (number) selects the register to put on busA (data)
  - RB (number) selects the register to put on busB (data)
  - RW (number) selects the register to be written
    via busW (data) when Write Enable is 1
- Clock input (clk)
  - Clk input is a factor ONLY during write operation
  - During read operation, behaves as a combinational logic block:
    - RA or RB valid ⇒ busA or busB valid after "access time."



State Required by RV32I ISA

Each instruction reads and updates this state during execution:

- Registers (**`x0..x31`**)
  - Register file (or *regfile*) **`Reg`** holds 32 registers x 32 bits/register: **`Reg[0].. Reg[31]`**
  - First register read specified by *rs1* field in instruction
  - Second register read specified by *rs2* field in instruction
  - Write register (destination) specified by *rd* field in instruction
  - **`x0`** is always 0 (writes to **`Reg[0]`** are ignored)
- Program Counter (**`PC`**)
  - Holds address of current instruction
- Memory (**`MEM`**)
  - Holds both instructions and data, in one 32-bit byte-addressed memory space
  - We'll use separate memories for instructions (**`IMEM`**) and data (**`DMEM`**)
    - *Later we'll replace these with instruction and data caches*
  - Instructions are read (*fetched*) from instruction memory (assume **`IMEM`** read-only)
  - Load/store instructions access data memory

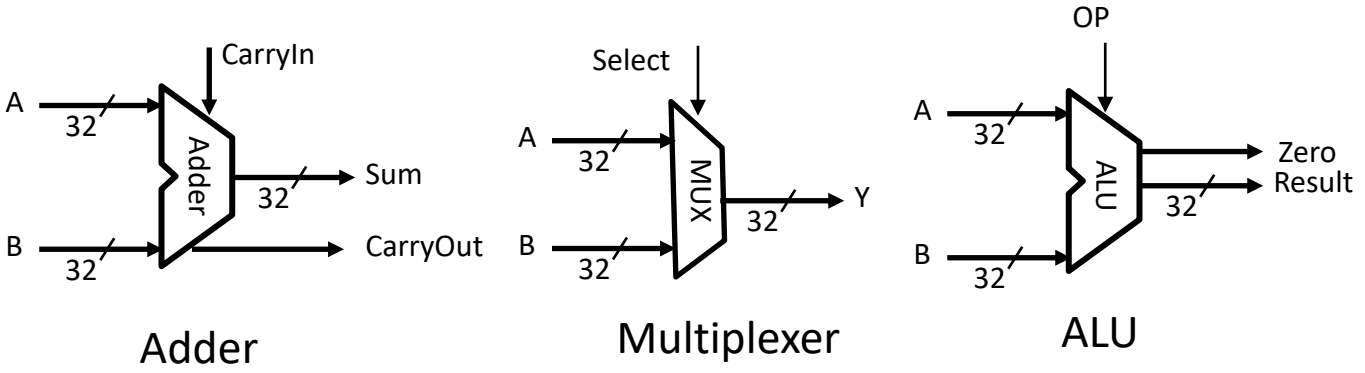
Requirements of Instruction Set

Requirements of the Instruction Set

- Memory (MEM)
  - Instructions Memory and data Memory
- Registers (x0 - x31)
- PC
- Sign Extender
- Add/Sub/OR unit for operation on register(s) or sign extended immediate
- Add 4 (or maybe sign extended immediate) to PC
- Compare if registers equal?

Components of the Datapath

- Combinational Elements
- State Elements + Clocking Methodology
- Building Blocks



One Instruction Per Cycle Datapath

Register Transfer Language (RTL)

- RTL gives the meaning of the instructions

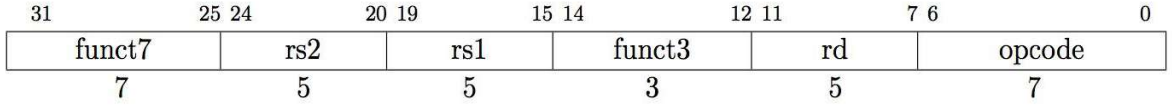
Inst	Register Transfers
ADD	R[rd] ← R[rs1] + R[rs2]; PC ← PC + 4
SUB	R[rd] ← R[rs1] – R[rs2]; PC ← PC + 4
ORI	R[rd] ← R[rs1] | sign_ext(Imm12); PC ← PC + 4
LW	R[rd] ← MEM[R[rs1] + sign_ext(Imm12)]; PC ← PC + 4
SW	MEM[R[rs1] + sign_ext(Imm12)] ← R[rs2]; PC ← PC + 4
BEQ	if (R[rs] == R[rt]) then PC ← PC + (sign_ext(Imm12) || 0) else PC ← PC + 4

Designing our Datapath: Where to start?

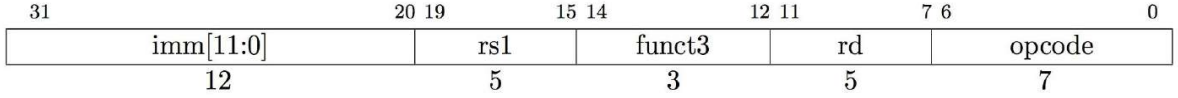
- 6 different formats: R, I , S, B, U, J
  - Arithmetic, Immediate, Store, Branch, Upper-immediate, JAL
- Instructions are classified into these formats based on their behaviors, meaning each type does something a little different

The RISC-V Lite Subset

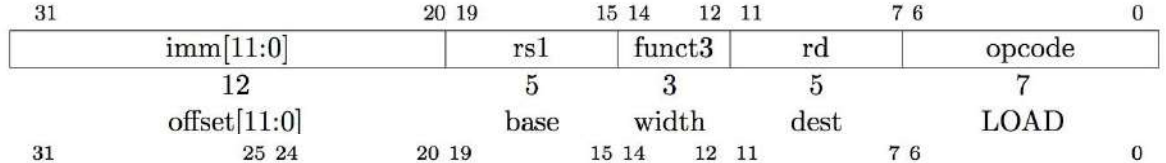
- ADD and SUB
  - `add rd,rs1,rs2`
  - `sub rd,rs1,rs2`



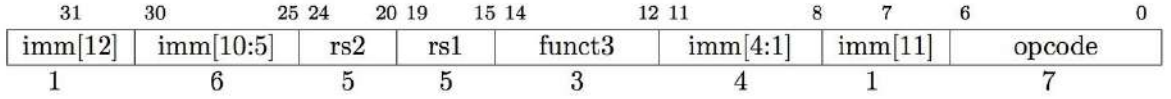
- OR Immediate:
  - `ori rd,rs1,imm12`



- LOAD and STORE Word
  - `lw rd,rs1,imm12`
  - `sw rs2,rs1,imm12`

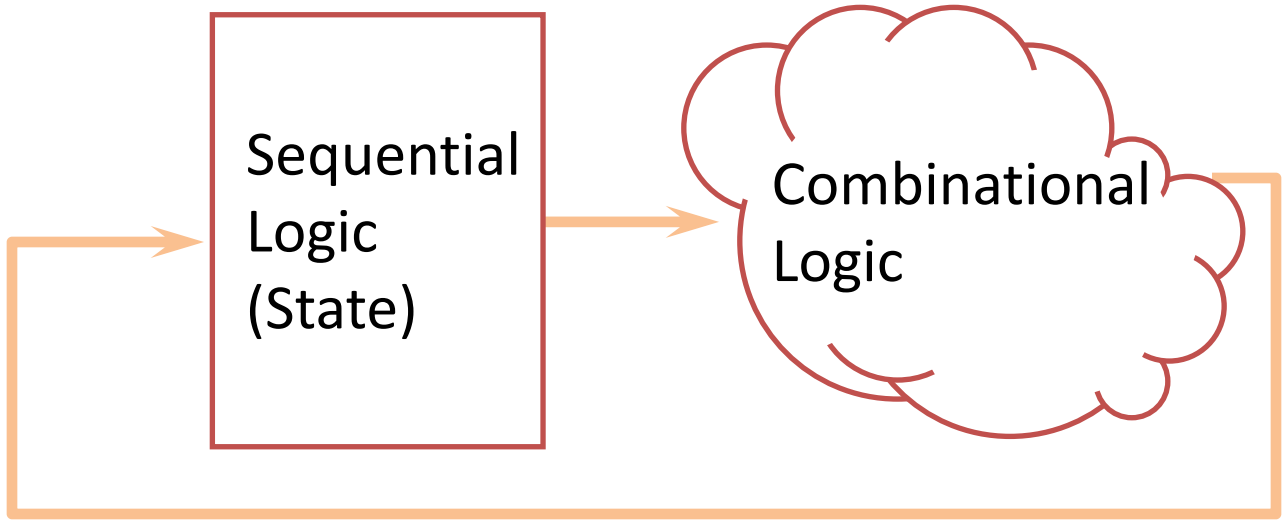


- BRANCH:
  - `beq rs1,rs2,imm12`

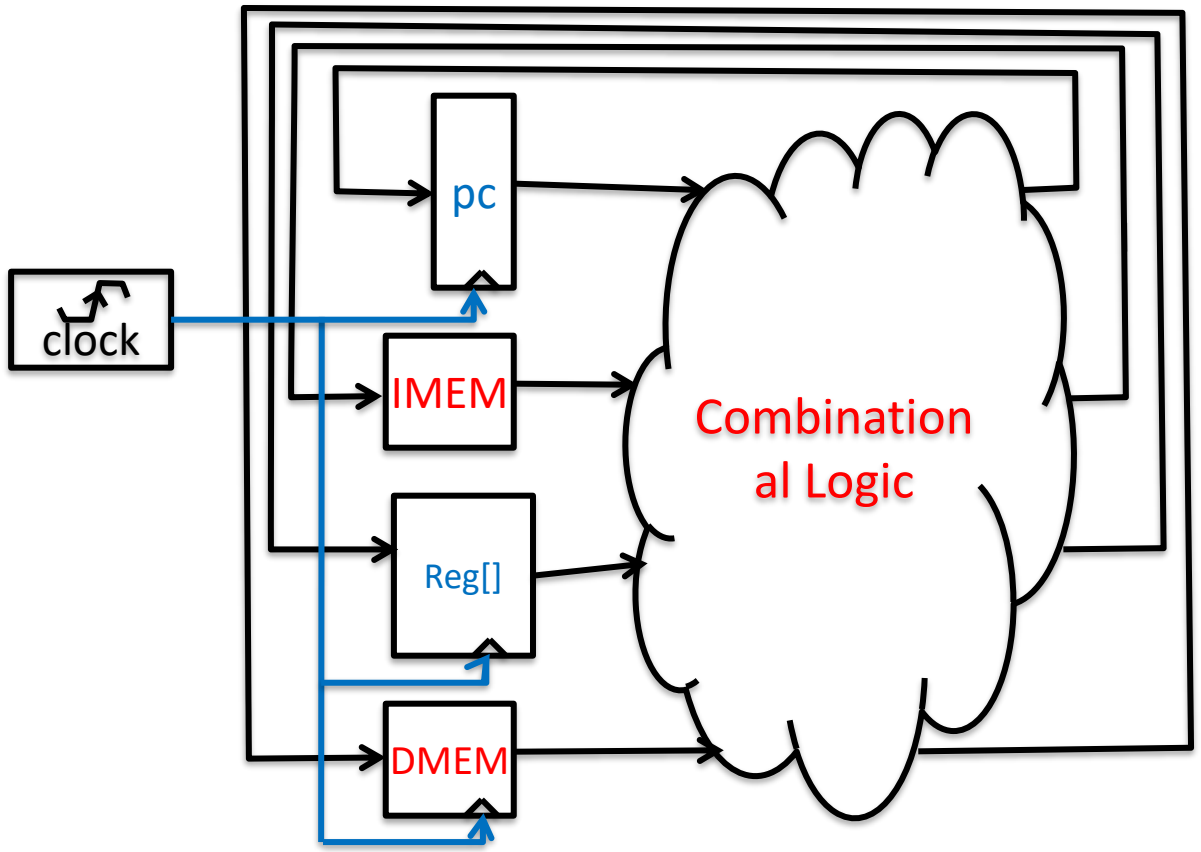


One instruction per cycle machine

- Single-cycle machine



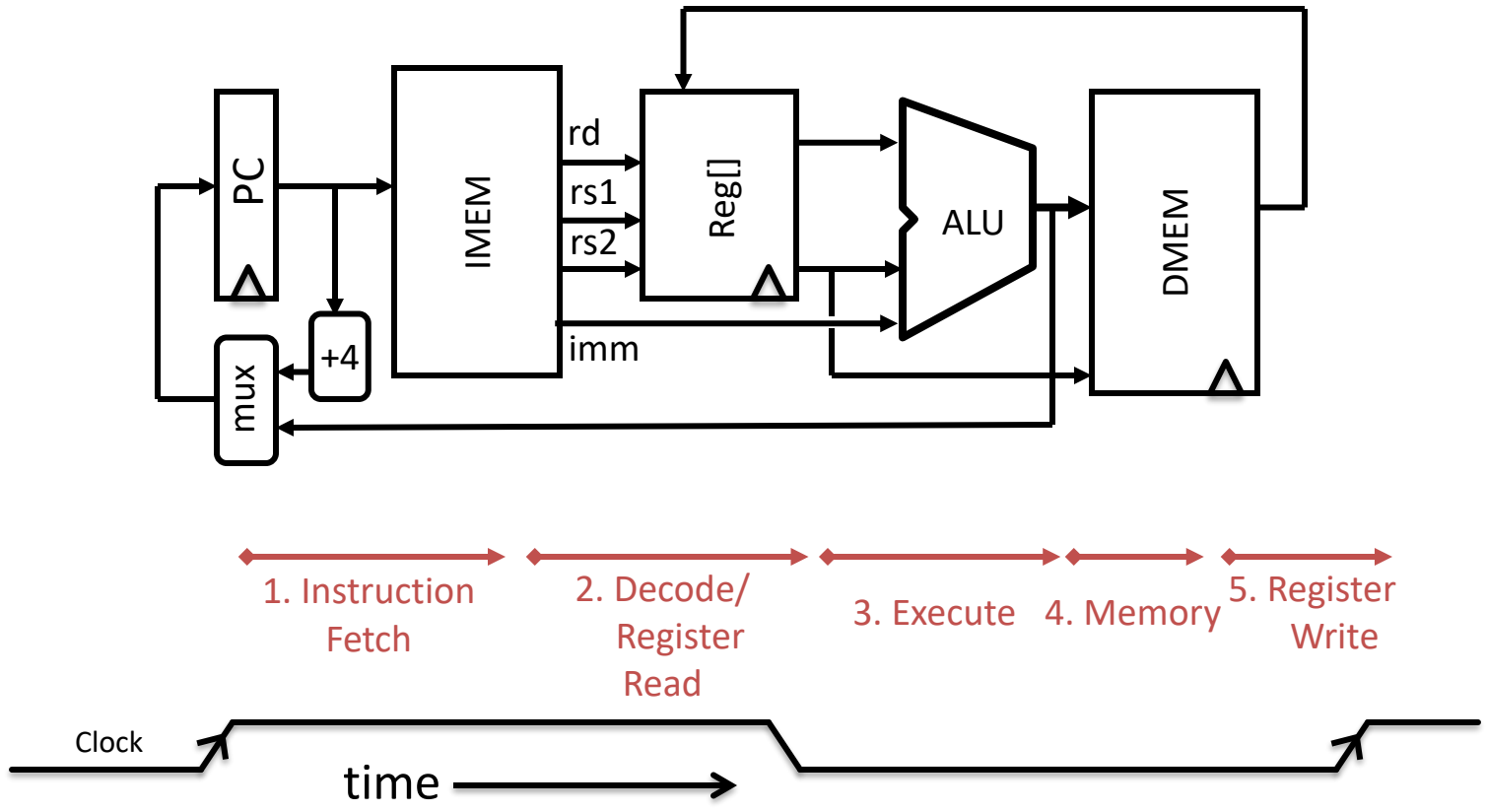
One-Instruction-Per-Cycle RISC-V Machine



- On every tick of the clock, the computer executes one instruction
- Current state outputs drive the inputs to the combinational logic, whose outputs settles at the values of the state before the next clock edge
- At the rising clock edge, all the state elements are updated with the combinational logic outputs, and execution moves to the next clock cycle

IMEM: Instruction Memory
DMEM: Data Memory

Basic Phases of Instruction Execution

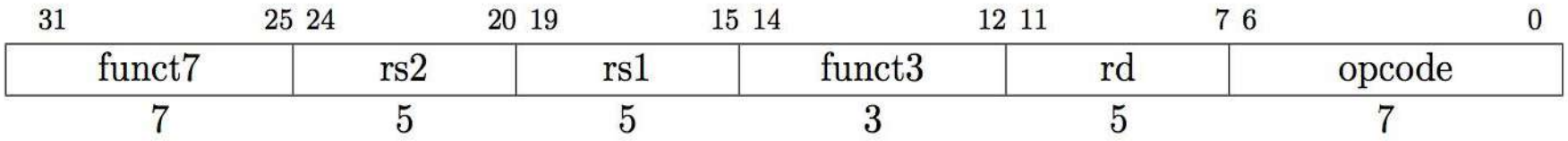


Why Five Stages?

- Could we have a different number of stages?
  - Yes. Many other architectures do
- So why does RISC-V have five stages if instructions tend to idle for at least one stage?
  - The five stages are the union of all the operations needed by all the instructions
  - There is one instruction, load (lw/lb) that uses all five stages

Assembling the Datapath for add/sub instruction

Implementing R type instruction



`add rd, rs1, rs2`

- Instruction makes two changes to machine's state:
  - **`Reg[rd] = Reg[rs1] + Reg[rs2]`**
  - **`PC = PC + 4`**

Implementing R type instruction

- What needs to be done to execute R type instruction
  (1) Get the instruction
  (2) Parse instruction fields (rd, rs1, rs2, operation…)
  (3) Read data based on parsed operands
  (4) Perform operation
  (5) Write result to our destination register

Implementing R type instruction

(1) Get the instruction

add x1 x2 x3

(2) Parse instruction fields (rd, rs1, rs2, **operation**…)

rd = x1 rs1 = x2 rs2 = x3

(3) Read data based on parsed operands

R[x2] R[x3]

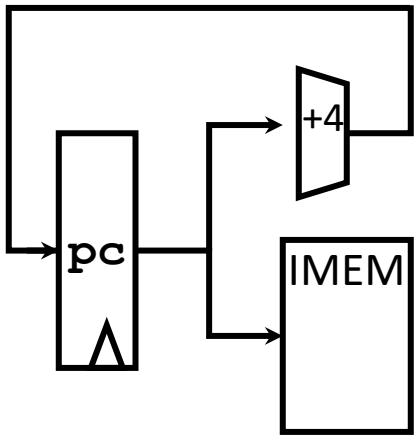
(4) Perform operation

R[x2] + R[x3]

(5) Write result to our destination register

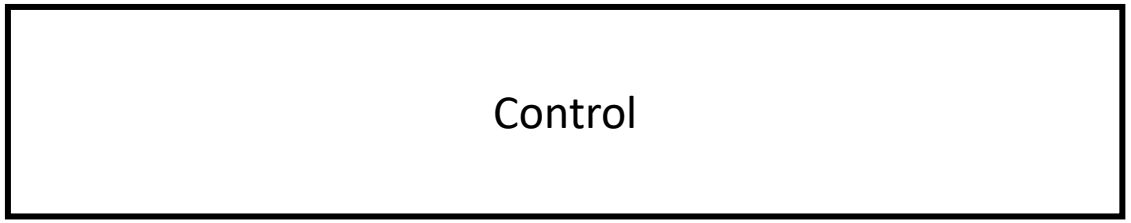
R[x1] = R[x2] + R[x3]

Implementing R-Types

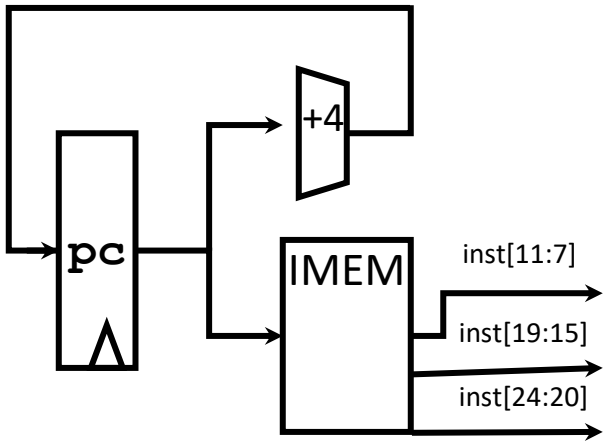


(1) Get the instruction

- PC holds the address of our current instruction
- Get instruction from IMEM

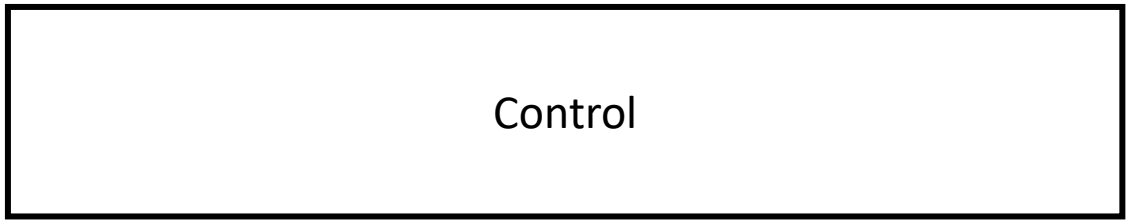


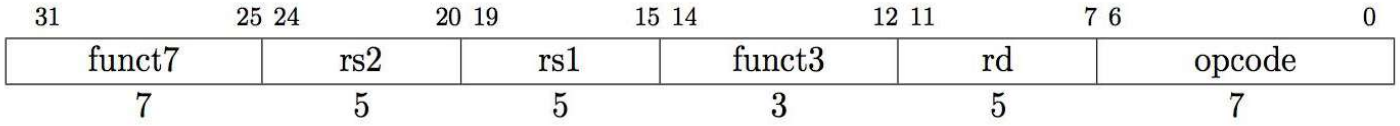
Implementing R-Types



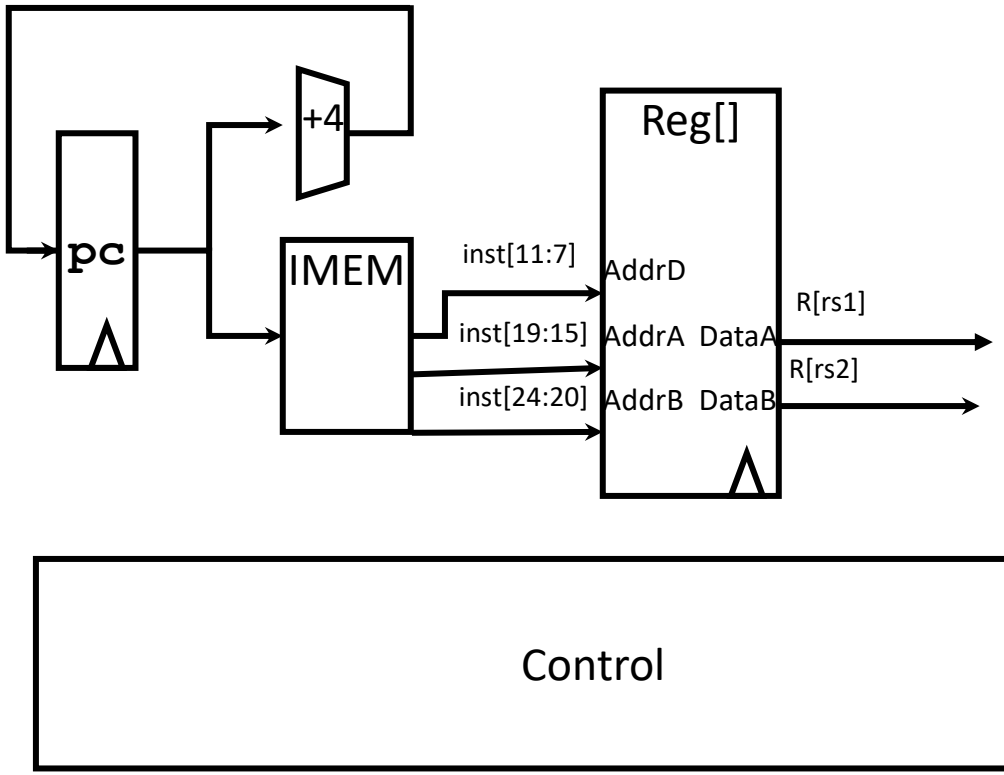
(2) Parse instruction fields (rd, rs1, rs2, **operation**…)

- What registers are we operating on?
- Where do they lie in our instruction format?
- How big is each field?





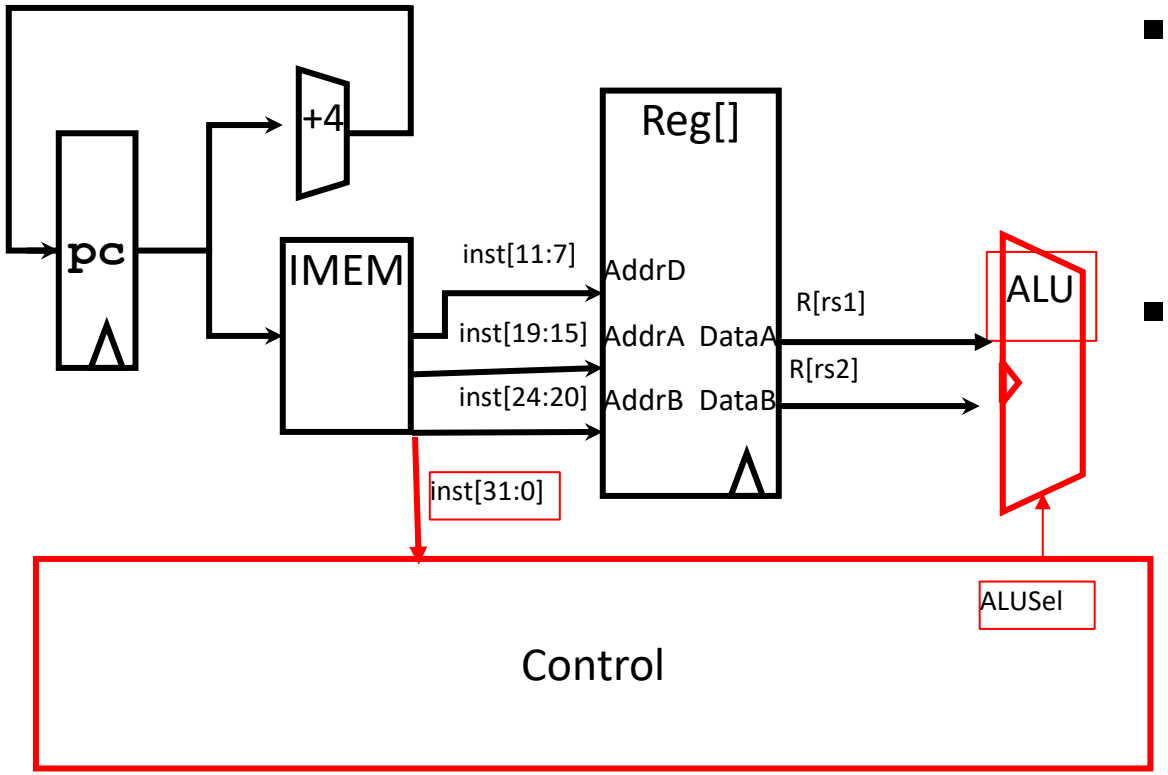
Implementing R-Types



(3) Read data based on parsed operands

- New hardware: Register File
- Simultaneous read R[rs1] and R[rs2]
- Note that reading can be done anytime

Implementing R-Types



(4) Perform operation

- New hardware: ALU (Arithmetic Logic Unit)
  - Abstraction for adders, multipliers, dividers, etc.
- How does ALU know what operation to execute?
  - Our first control bit! ALUSel(ect)

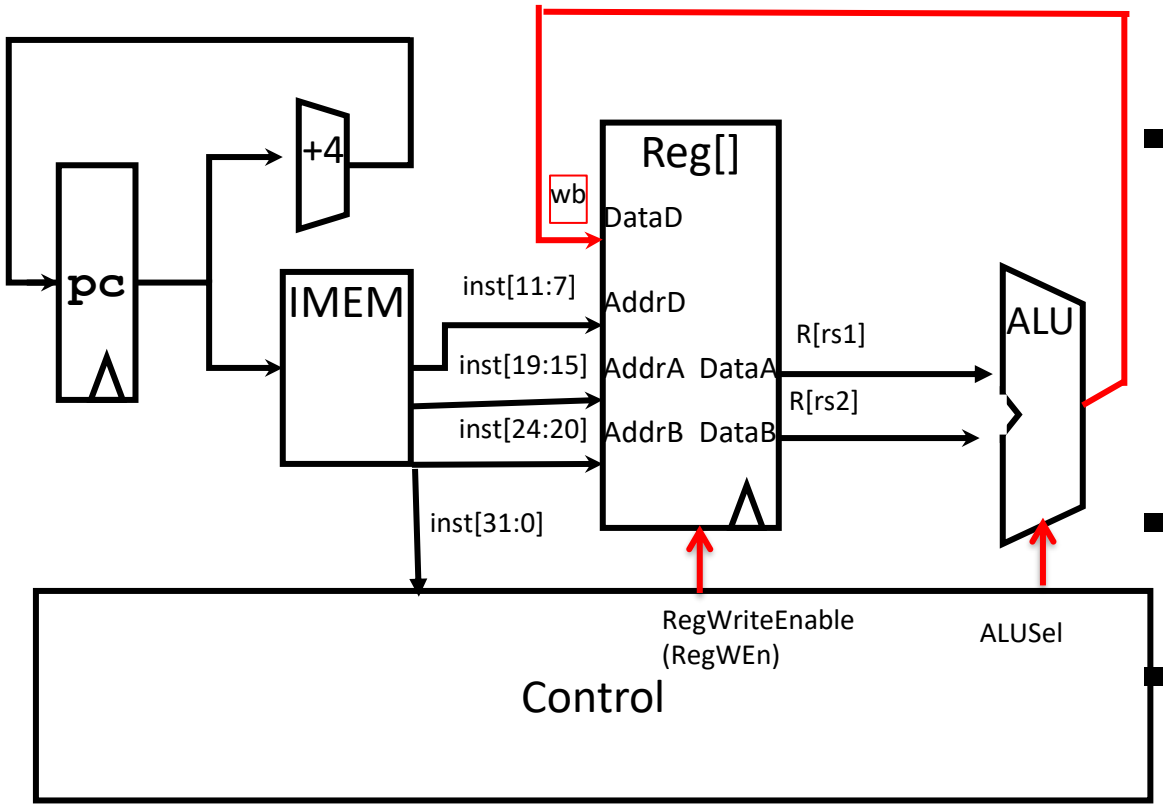
There are many R-Type operations!

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

ALUSel is a control bit which encodes the operation to be performed on operands

- The value of **ALUSel** is a mapping from func3 and func7 values to operations
- “if func3 == 000 and func7 == 0000000, perform an add”

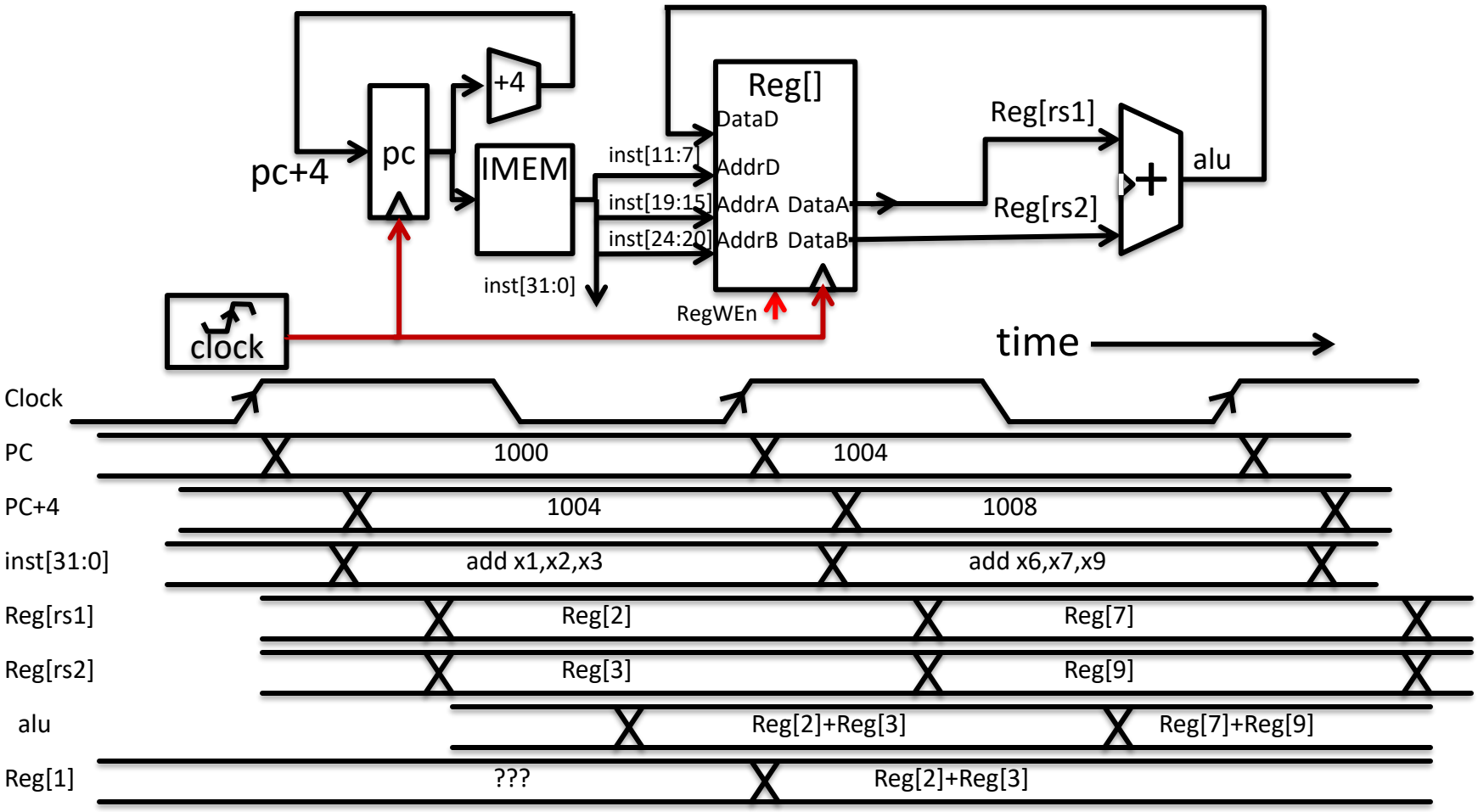
Implementing R-Types



(5) Write result to our destination register

- The data we want to write is the result of computing operation on operands, ie. the output from our ALU
- Send it back to the **regfile** for writing
- RegWEn control signal

Timing Diagram for add



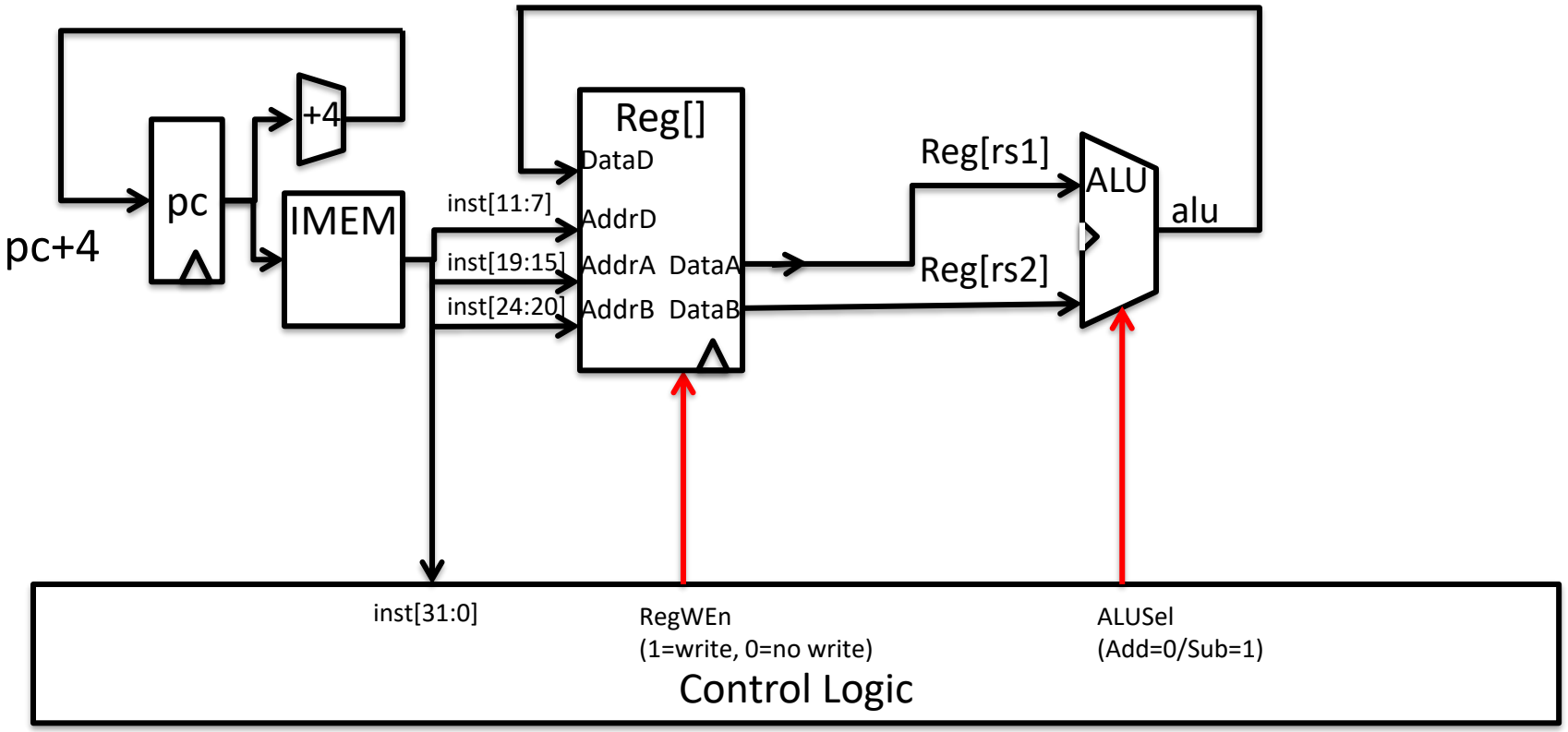
Implementing `sub` instruction

0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB

`sub rd, rs1, rs2`

- Almost the same as add, except now have to subtract operands instead of adding them
- Instruction bit 30 selects between add and subtract

Datapath for **add/sub**



Implementing other R-Format instructions

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

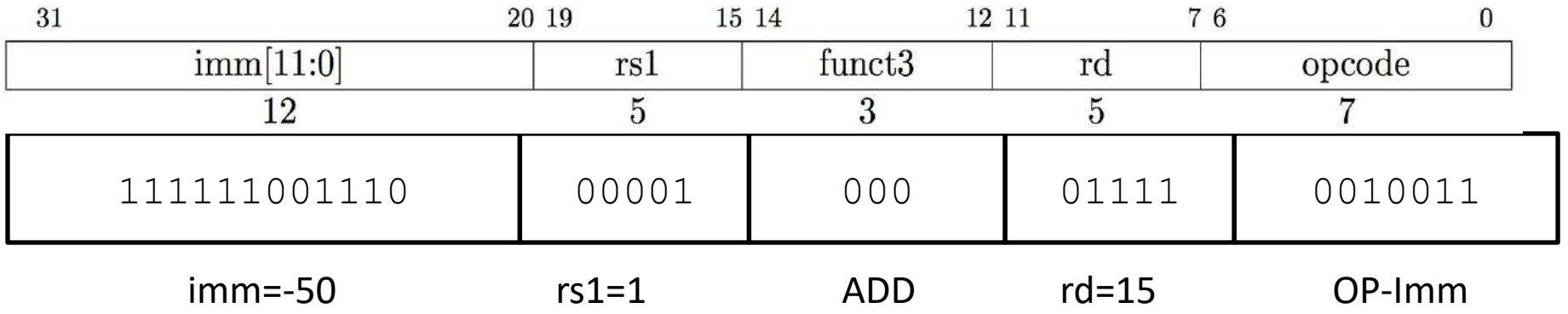
- All implemented by decoding funct3 and funct7 fields and selecting appropriate ALU function

Datapath for `addi`

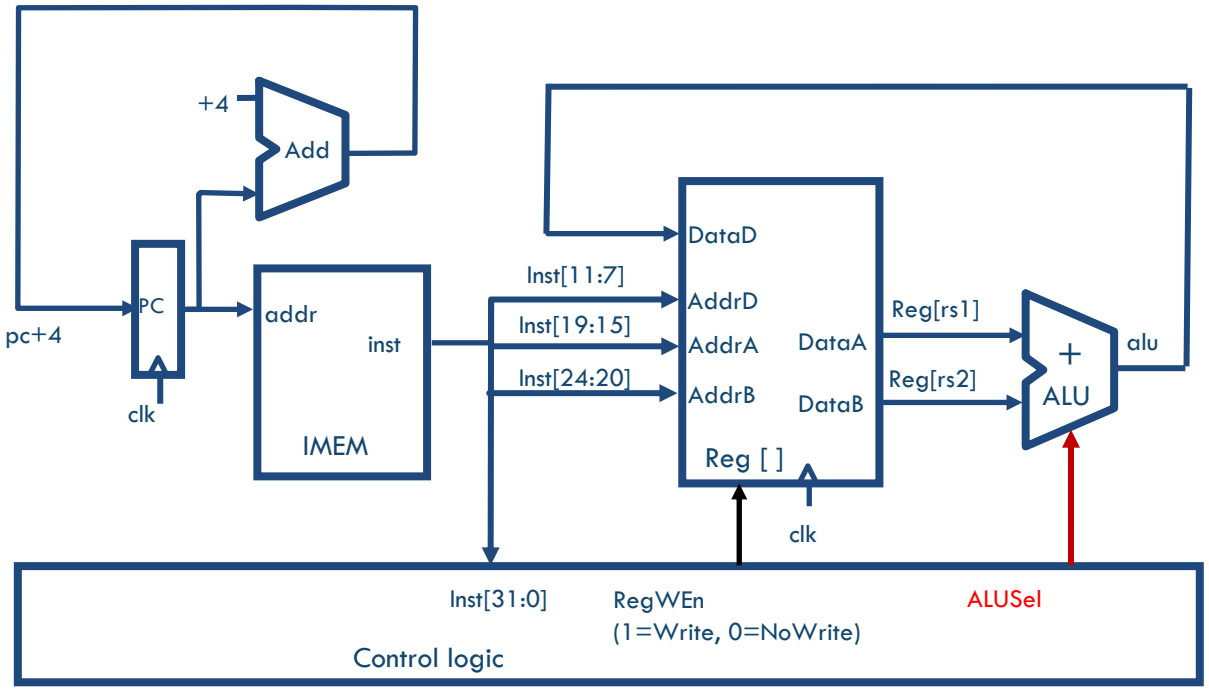
Implementing the `addi` instruction

- RISC-V Assembly Instruction:

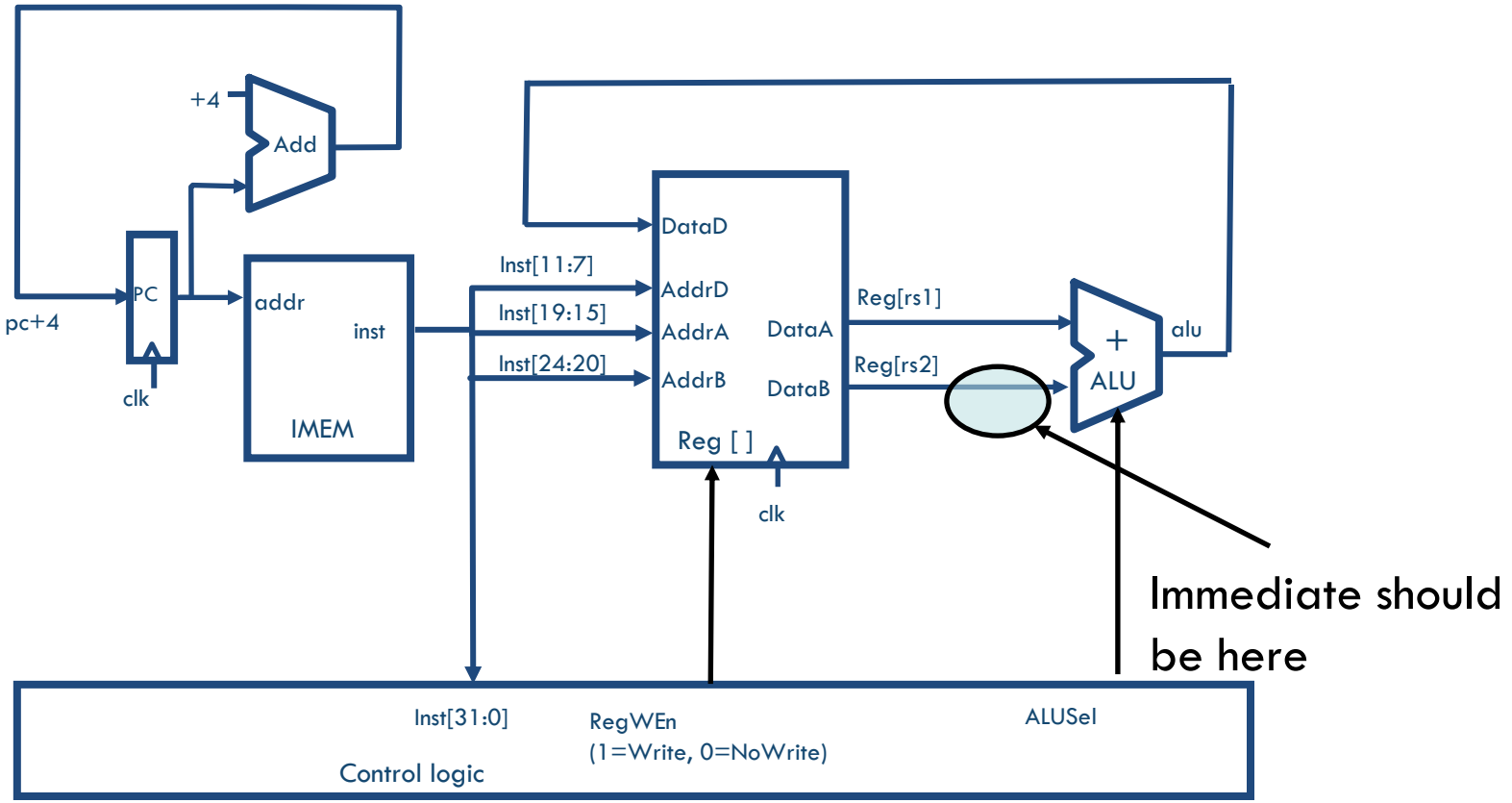
```
addi   x15,x1,-50
```



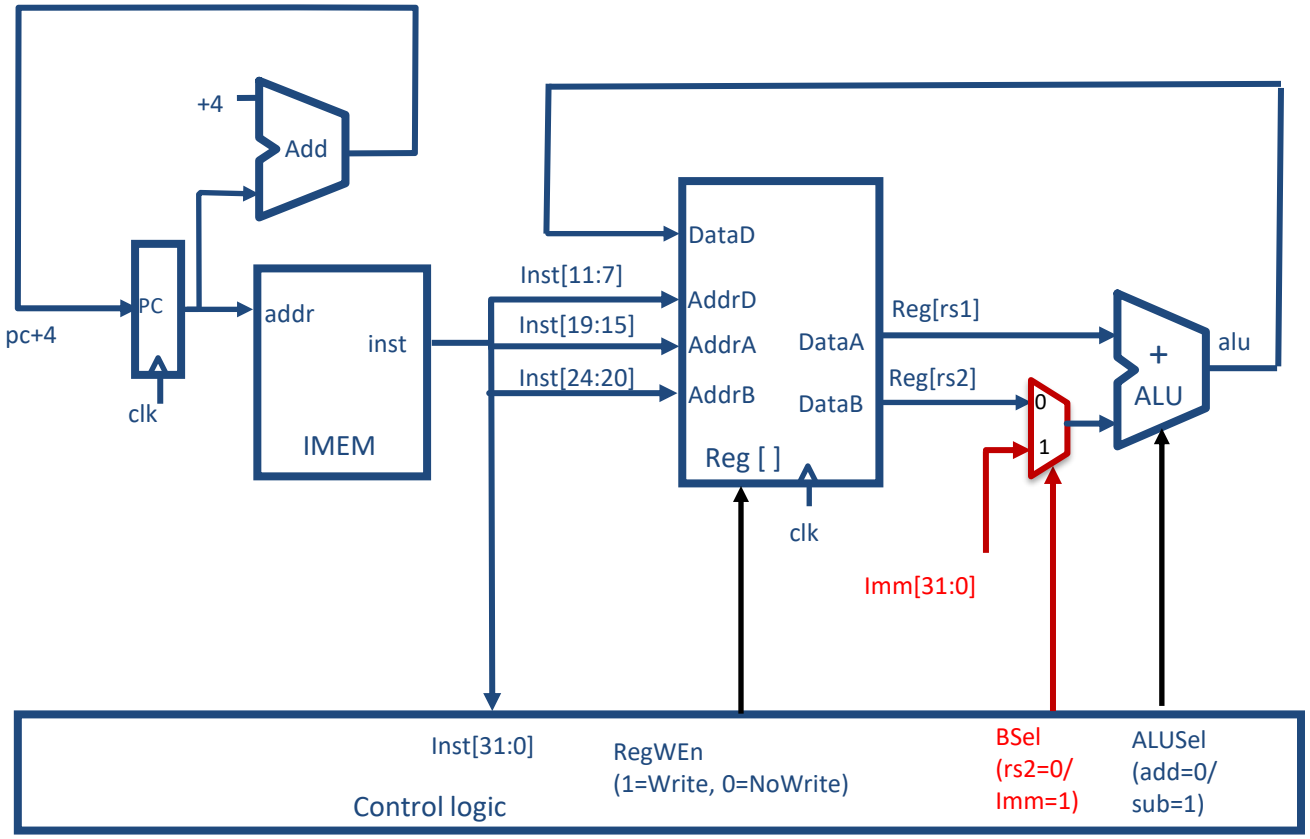
Datapath for `add/sub`



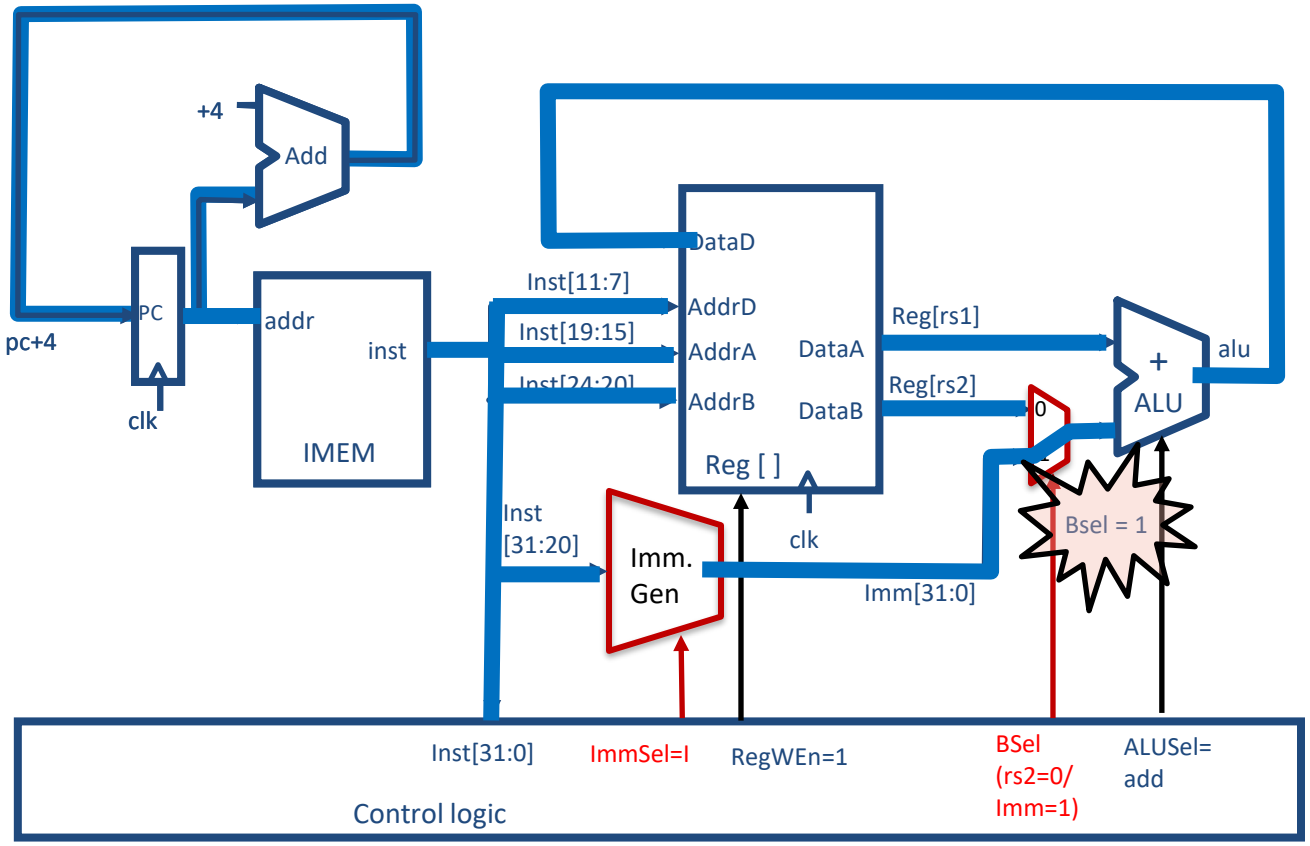
Datapath for `add/sub`



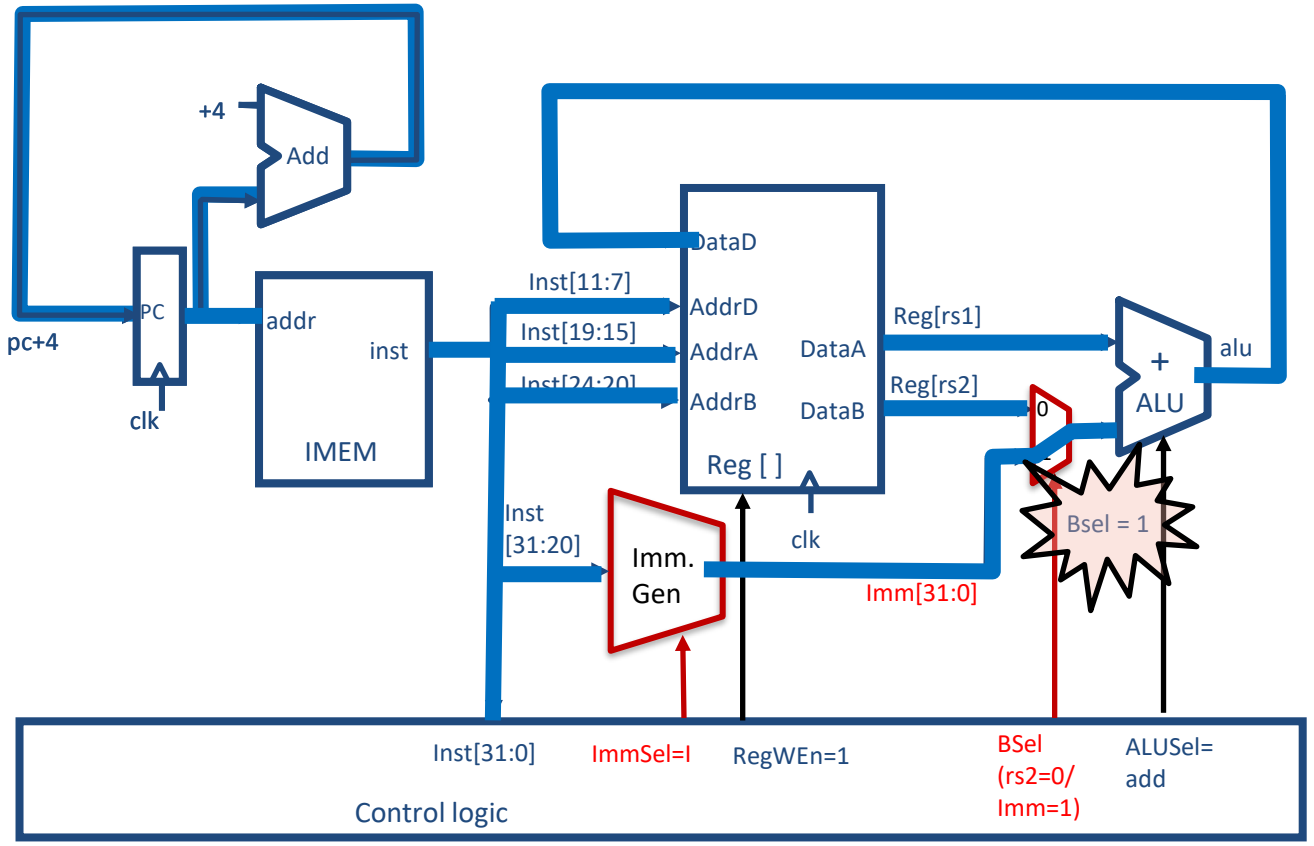
Adding `addi` to Datapath



Adding `addi` to Datapath



Adding `addi` to Datapath



(1) Get the instruction

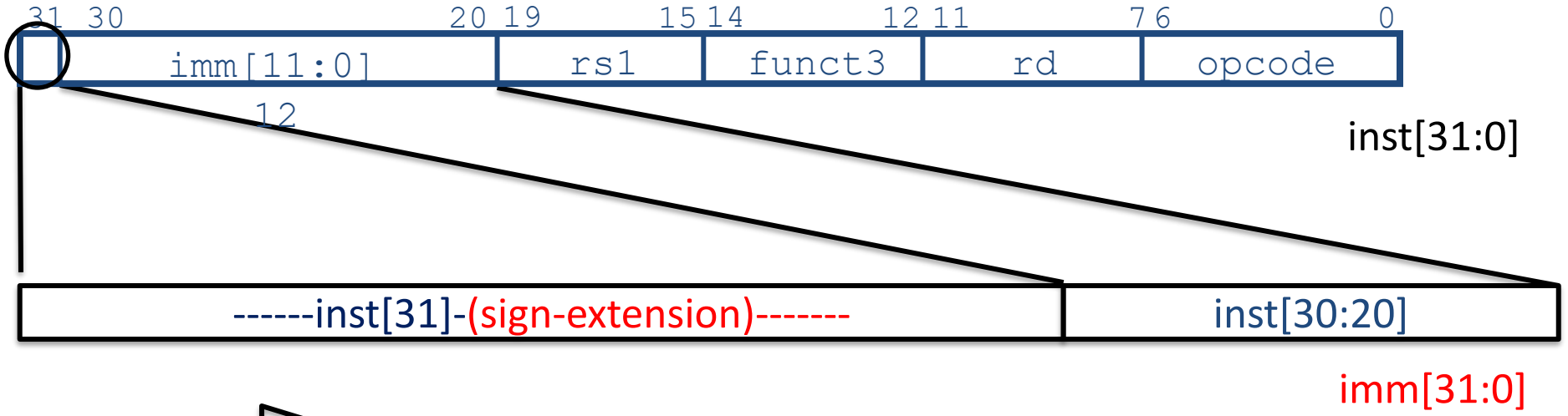
(2) Parse instruction fields (rd, rs1, rs2, operation, imm…)

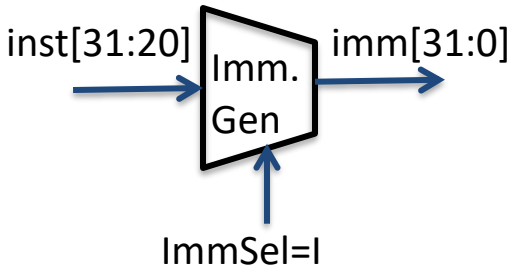
(3) Read data based on parsed operands

(4) Perform operation

(5) Write result to our destination register

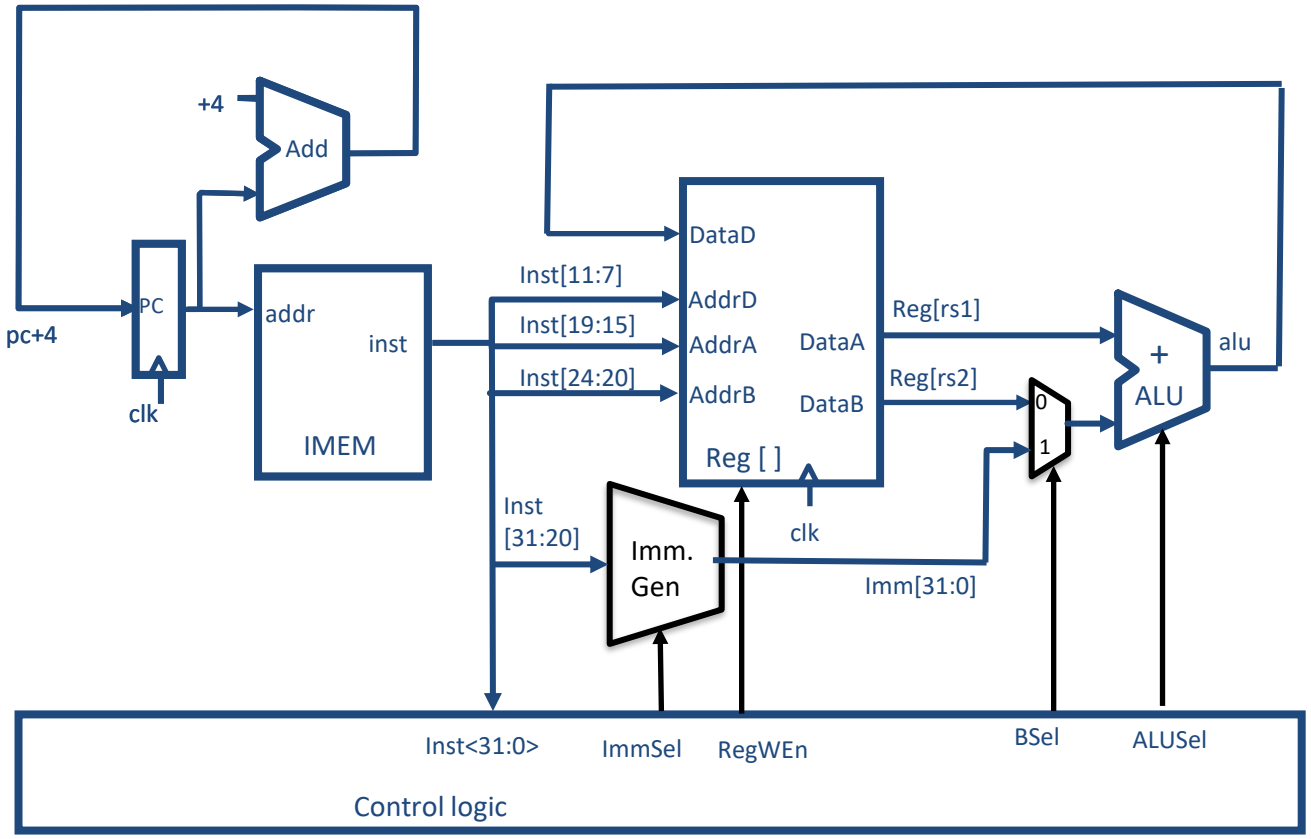
I-Format immediates





- High 12 bits of instruction (inst[31:20]) copied to low 12 bits of immediate (imm[11:0])
- Immediate is sign-extended by copying value of inst[31] to fill the upper 20 bits of the immediate value (imm[31:12])

R+I Datapath



Works for all other I-format arithmetic instructions (slti,sltiu,andi, ori,xori,slli,srli, srai) just by changing ALUSel

The ALU

- Performs arithmetic, logical, and shift operations
- Inputs: two 32-bit operands, ALU control signal
- Output: 32-bit result, Zero flag
- Operations needed for RV32I:
  - add, sub, and, or, xor
  - sll, srl, sra
  - slt, sltu
  - Pass-through for addresses
- ALU control: 4 bits (enough for 16 operations)

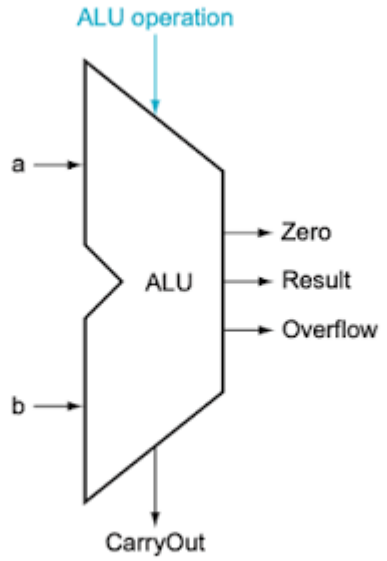
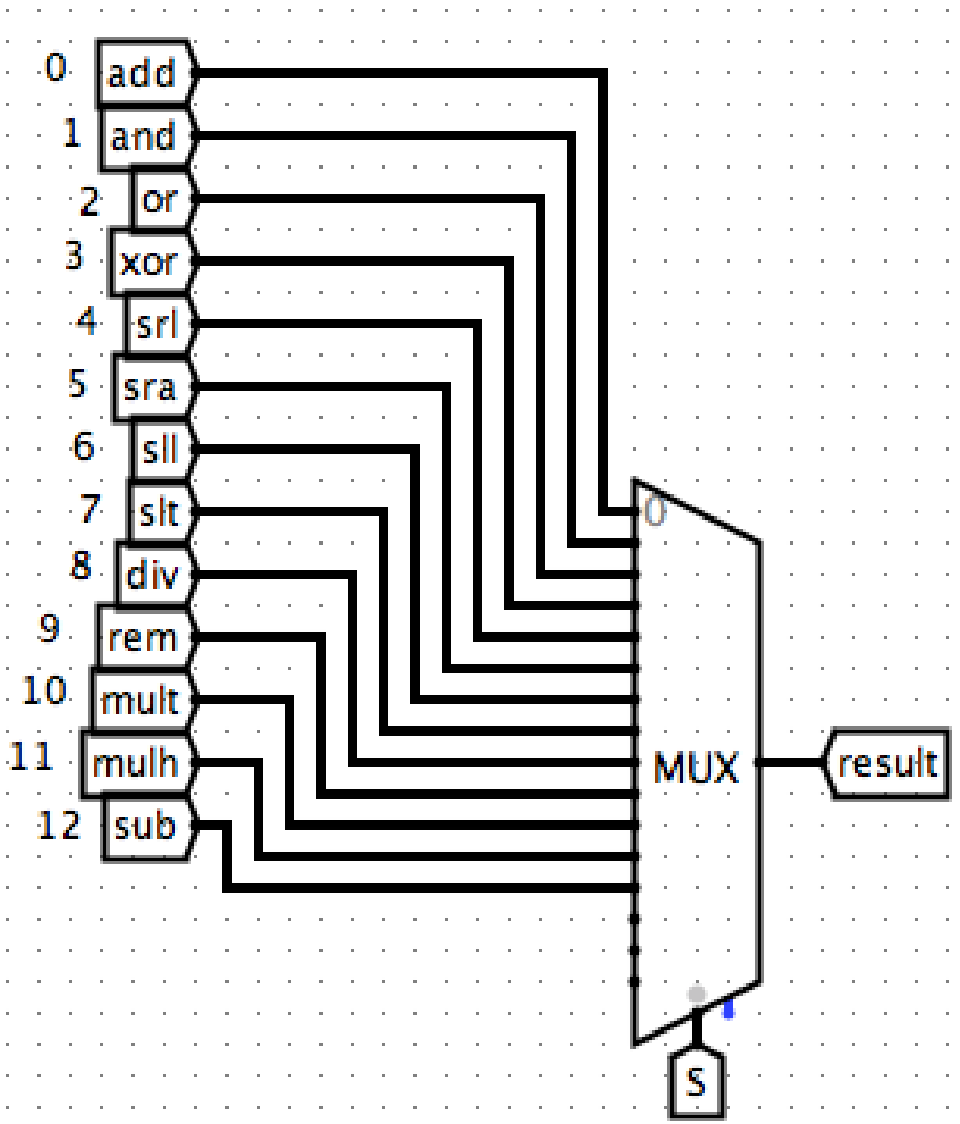


FIGURE A.5.14 The symbol commonly used to represent an ALU, as shown in Figure A.5.12. This symbol is also used to represent an adder, so it is normally labeled either with ALU or Adder.

ALU Operations Table

ALU Control	Operation	Used by
0000	AND	and, andi
0001	OR	or, ori
0010	ADD	add, addi, lw, sw, jalr
0110	SUB	sub, beq
0011	XOR	xor, xori
0100	SLL	sll, slli
0101	SRL	srl, srli
0111	SRA	sra, srai
1000	SLT	slt, slti
1001	SLTU	sltu, sltiu
1010	PASS B	lui

ALU Operations Table



Questions?

Thank You!