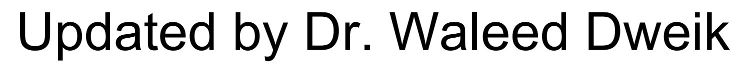
Chapter 1

Computer Abstractions and Technology


Updated by Dr. Waleed Dweik

The Computer Revolution

- Progress in computer technology
  - Underpinned by Moore's Law
- Makes novel applications feasible
  - Computers in automobiles
  - Cell phones
  - Human genome project
  - World Wide Web
  - Search Engines
- Computers are pervasive

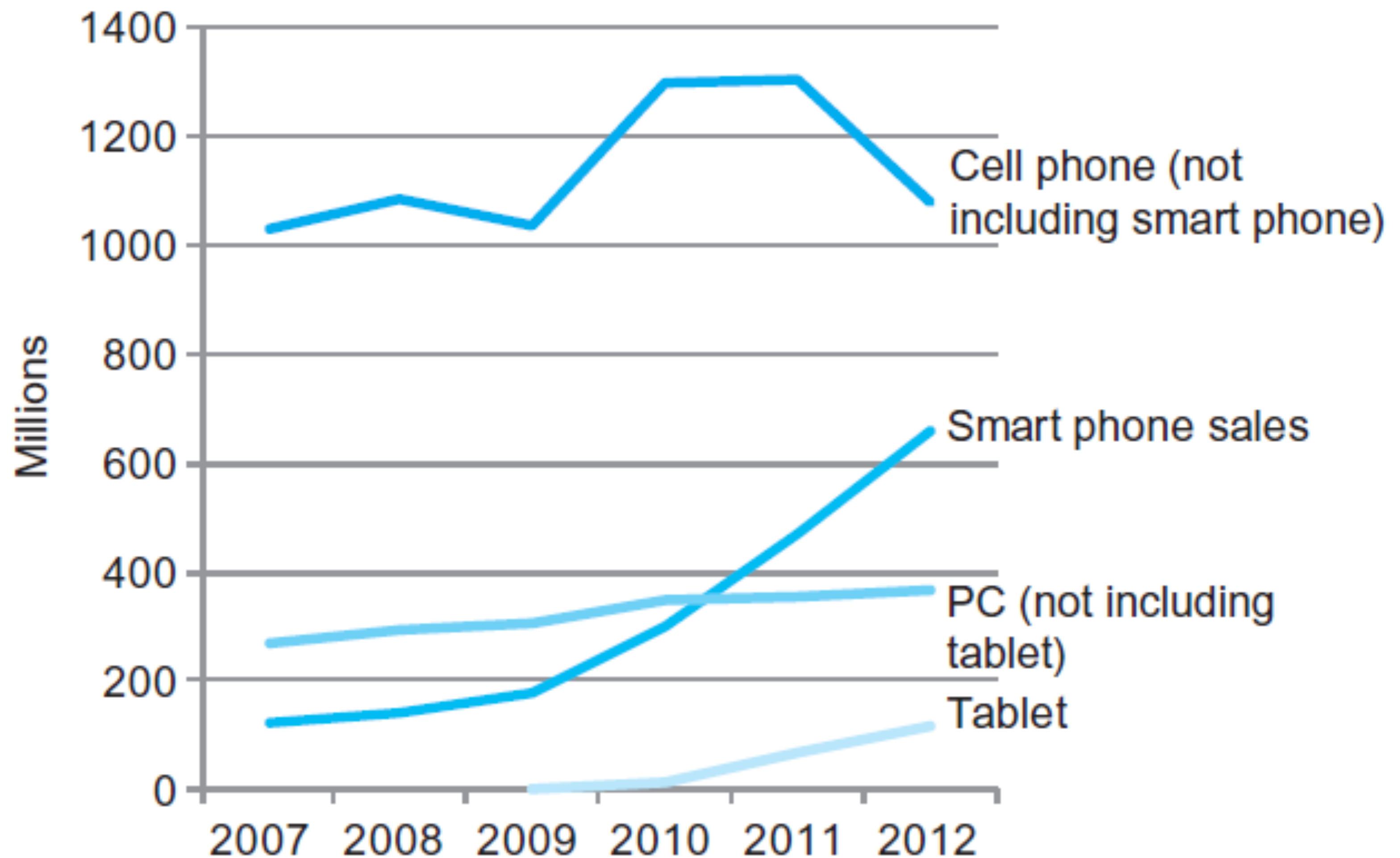
Classes of Computers

- Personal computers
  - General purpose, variety of software
  - Subject to cost/performance tradeoff
- Server computers
  - Network based
  - High capacity, performance, reliability
  - Range from small servers to building sized

Classes of Computers

- Supercomputers
  - High-end scientific and engineering calculations
  - Highest capability but represent a small fraction of the overall computer market
- Embedded computers
  - Hidden as components of systems
  - Stringent power/performance/cost constraints
  - Run one set of related applications that are normally integrated with the hardware

The PostPC Era



The PostPC Era

- Personal Mobile Device (PMD)
  - Battery operated
  - Connects to the Internet
  - Hundreds of dollars
  - Smart phones, tablets, electronic glasses
- Cloud computing
  - Warehouse Scale Computers (WSC)
  - Software as a Service (SaaS)
  - Portion of software run on a PMD and a portion run in the Cloud
  - Amazon and Google

Understanding Performance

- Algorithm
  - Determines number of operations executed
  - Not included in the book
- Programming language, compiler, architecture
  - Determine number of machine instructions executed per operation
  - Chapters 2 and 3
- Processor and memory system
  - Determine how fast instructions are executed
  - Chapters 4, 5, and 6
- I/O system (including OS)
  - Determines how fast I/O operations are executed
  - Chapters 4, 5, and 6

Eight Great Ideas (1)

- Design for ***Moore's Law***

  - Integrated circuits resources double every 18-24 months
  - Computer design takes years, the resources can double or quadruple between the start and finish of the project

- Use ***abstraction*** to simplify design

  - Represent the design at multiple levels such that low-level details are hidden to offer simpler model at high-level
  - Increase productivity for computer architects and programmers

- Make the ***common case fast***

  - Enhance the performance is better than optimizing the rare case
  - Example?

Eight Great Ideas (2)

- Performance *via* ***parallelism***

- Performance *via* ***pipelining***
  - Special pattern of parallelism
  - Example: Human bridge to fight fires

- Performance *via* ***prediction***
  - Better ask for forgiveness than ask for permission

- ***Hierarchy*** of memories
  - Speed, size, and cost
  - Fastest, smallest, and most expensive at the top of the hierarchy (Cache)
  - Slowest, largest, and least expensive at the bottom of the hierarchy (Hard Drives)

- ***Dependability*** *via* redundancy

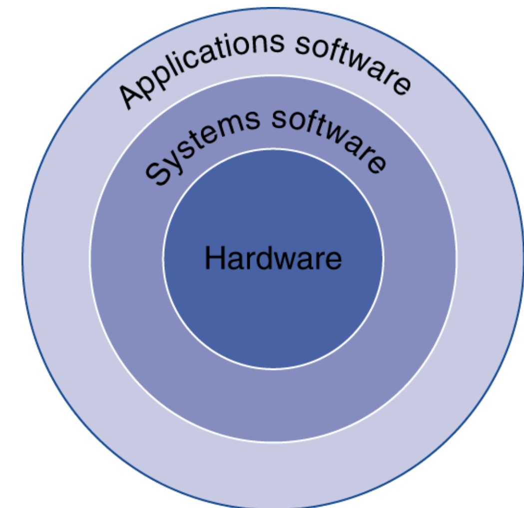
Eight Great Ideas (3)

- Why these ideas are considered great?
  - Lasted long after the 1st computer that used them
  - They have been around for the last 60 years

MK MORGAN KAUFMANN

Below Your Program

- Application software
  - Written in high-level language (HLL)
    - Allow the programmer to think in a more natural language (look much more like a text than tables of cryptic symbols)
    - Improved productivity (conciseness)
    - Programs become independent from hardware
- System software
  - Compiler: translates HLL code to machine code
  - Operating System: service code
    - Handling input/output
    - Managing memory and storage
    - Scheduling tasks & sharing resources
    - E.g. Windows, Linux, iOS
- Hardware
  - Processor, memory, I/O controllers



Levels of Program Code

- **High-level language**
  - Level of abstraction closer to problem domain
  - Provides for productivity and portability
  - E.g. A + B
- **Assembly language**
  - Textual representation of instructions
  - E.g. Add A,B
- **Hardware representation**
  - Binary digits (bits)
  - Encoded instructions and data
  - E.g. 11100011010

High-level language program (in C)

```
swap(int v[], int k)
{int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

Compiler

Assembly language program (for RISC-V)

```
swap:
      slli x6, x11, 3
      add  x6, x10, x6
      ld   x5, 0(x6)
      ld   x7, 8(x6)
      sd   x7, 0(x6)
      sd   x5, 8(x6)
      jalr x0, 0(x1)
```

Assembler

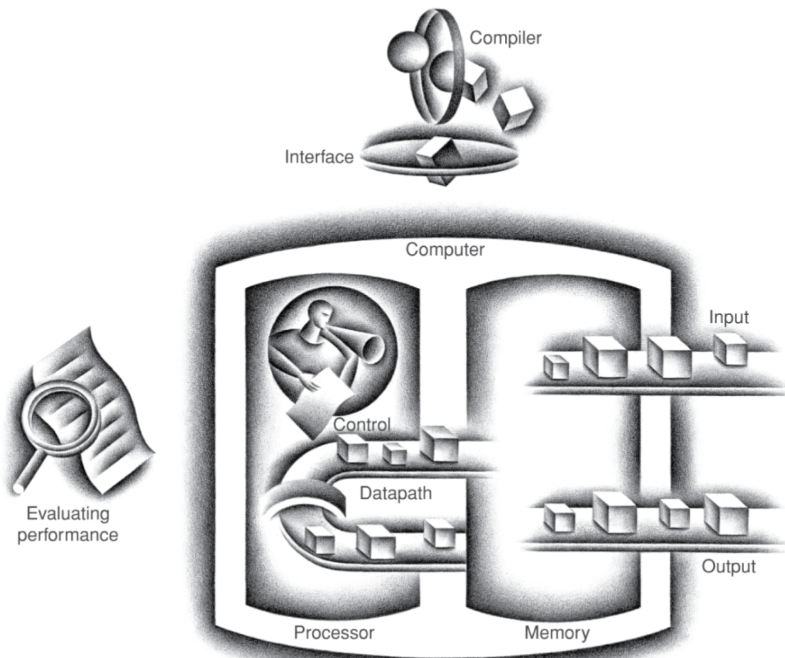
Binary machine language program (for RISC-V)

```
00000000001101011001001100010011
00000000011001010000001100110011
00000000000000110011001010000011
00000000100000110011001110000011
00000000011100110011000000100011
00000000010100110011010000100011
00000000000000001000000001100111
```

MK MORGAN KAUFMANN

Components of a Computer

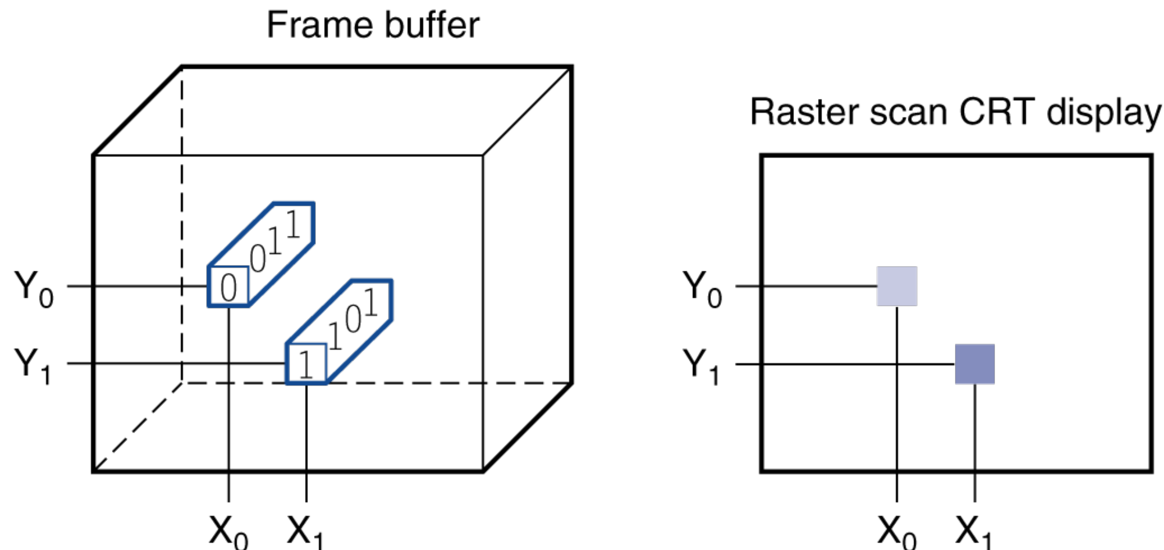
The BIG Picture



- Same components for all kinds of computer
  - Desktop, server, embedded
- Five components
  - Input, output, memory, datapath and control
- Input/output includes
  - User-interface devices
    - Display, keyboard, mouse
  - Storage devices
    - Hard disk, CD/DVD, flash
  - Network adapters
    - For communicating with other computers

Through the Looking Glass

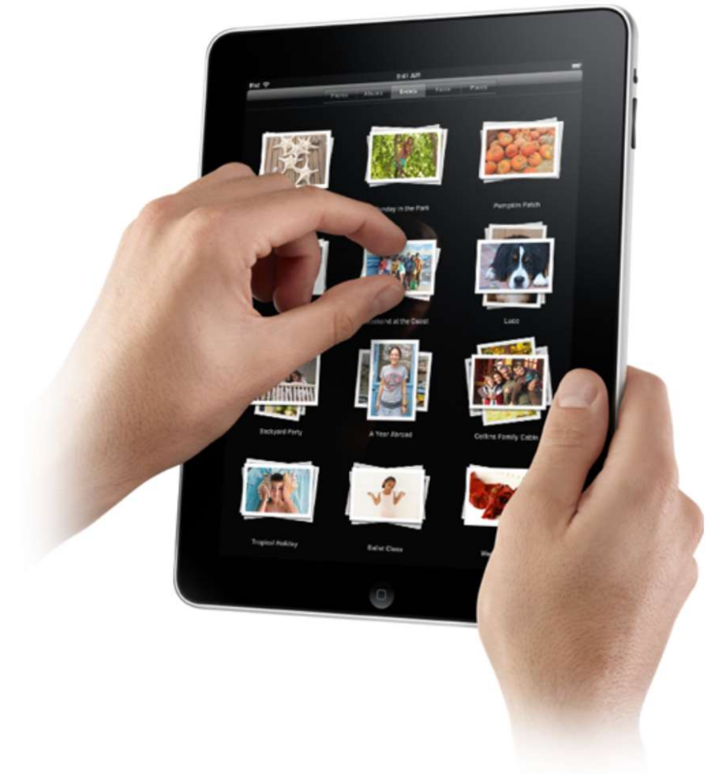
- LCD (liquid crystal display) screen: picture elements (pixels)
- An image is composed of a matrix of pixels called a bit map
  - Bit map size is based on screen size and resolution
  - 1024x768 to 2048x1536
- Hardware support for graphics
  - Frame buffer (raster refresh buffer) to store the bit map of the image to be displayed
  - Bit pattern for pixels is read out at the refresh rate



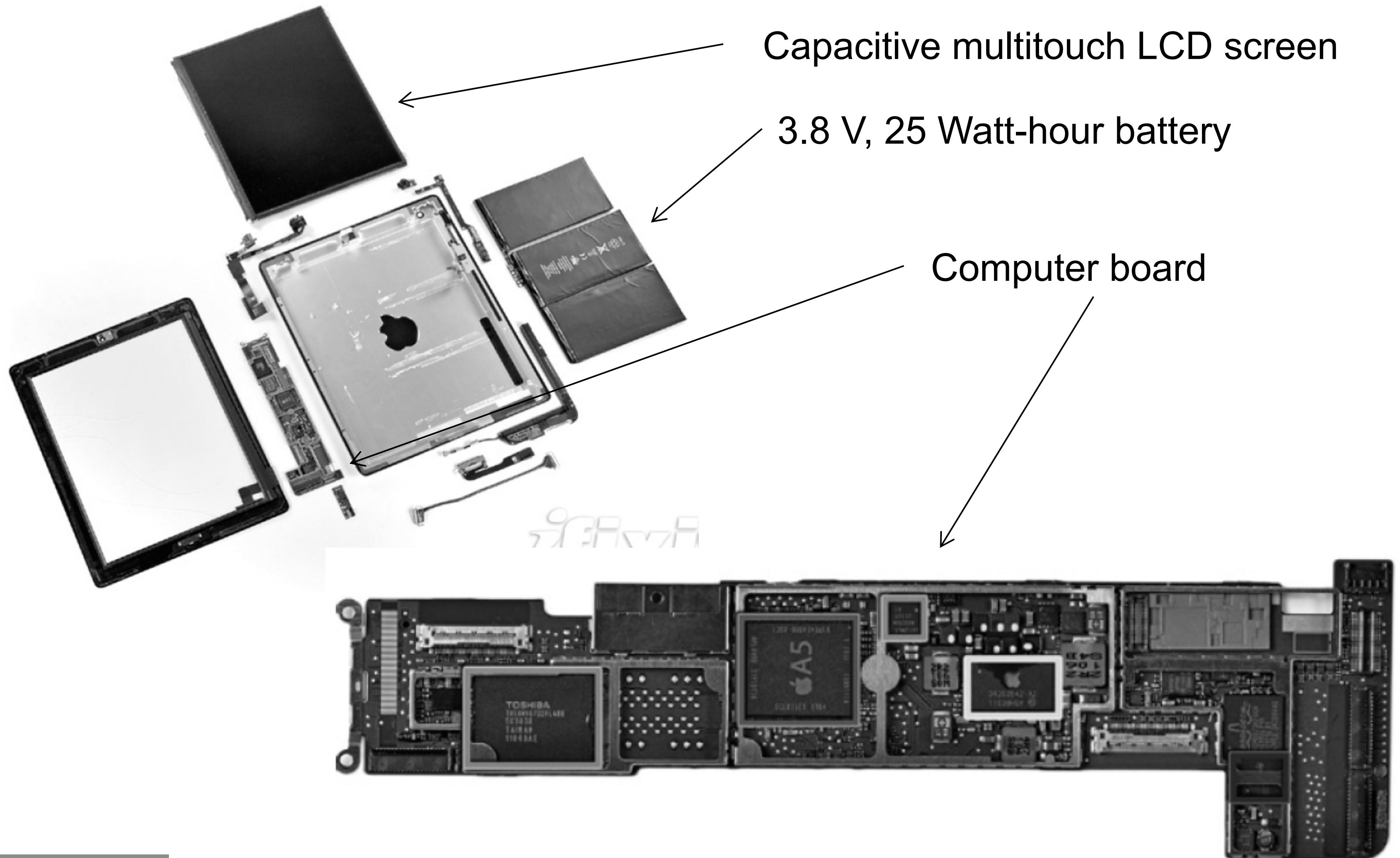
MK
MORGAN KAUFMANN

Touchscreen

- PostPC device
- Supersedes keyboard and mouse
- Resistive and Capacitive types
  - Most tablets, smart phones use capacitive
  - Capacitive allows multiple touches simultaneously



Opening the Box



Inside the Processor (CPU)

- Datapath: performs operations on data
- Control: sequences datapath, memory, ...
- Cache memory
  - Small fast SRAM memory for immediate access to data

MORGAN KAUFMANN

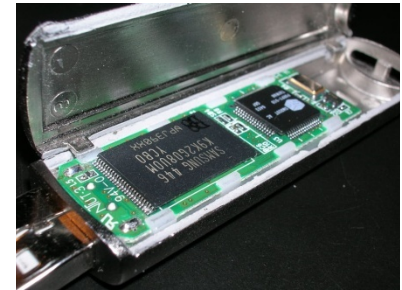
Inside the Processor

- Apple A5

MK MORGAN KAUFMANN

A Safe Place for Data (1)

- Volatile main (primary) memory
  - DRAMs: Dynamic Random Access Memory
  - Holds data and programs while running
  - Loses instructions and data when power off
- Non-volatile secondary memory
  - Magnetic disk
  - Flash memory in PMDs
  - Optical disk (CDROM, DVD)



A Safe Place for Data (2)

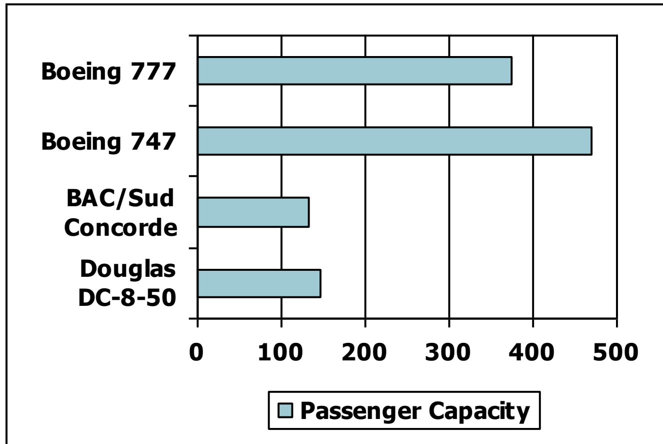
	DRAM	Magnetic Disk	Flash
Price/GByte	Expensive (5$)	Cheap (0.05 – 0.1)$	Moderate (0.75 – 1)$
Speed	Fast (50-70)ns	Slow (5-20)ms	Moderate (5-50)μs
Volatility	Volatile	Non-volatile	Non-volatile
Wearout	N/A	N/A	1,00,000-1,000,000 writes

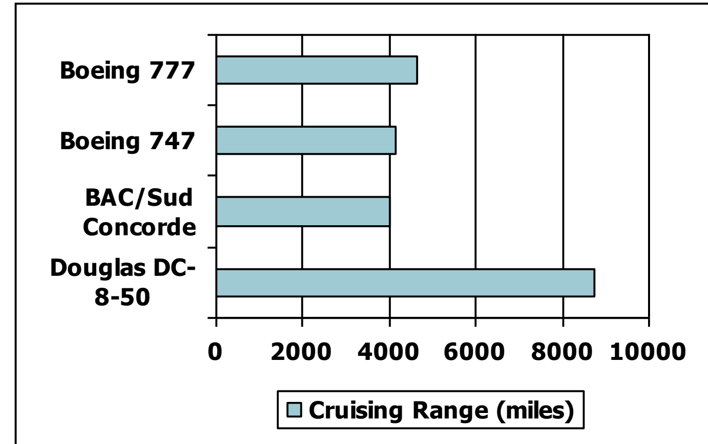
Networks

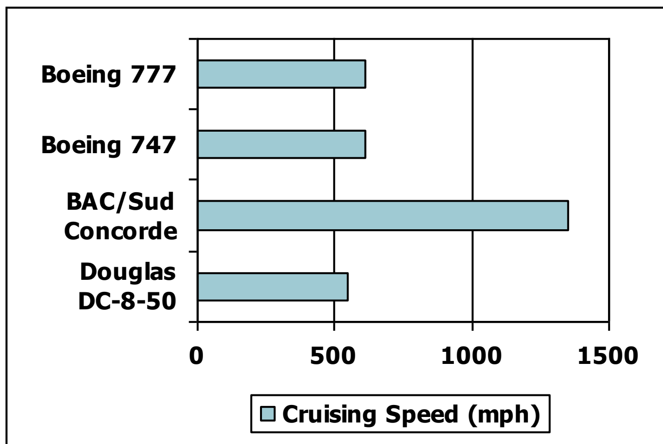
- Communication, resource sharing, nonlocal access
- Local area network (LAN): Ethernet
  - 1 Km, 40 Gbit/sec
- Wide area network (WAN): the Internet
  - Optical fiber
- Wireless network: WiFi, Bluetooth
  - 1-100 million bit/sec

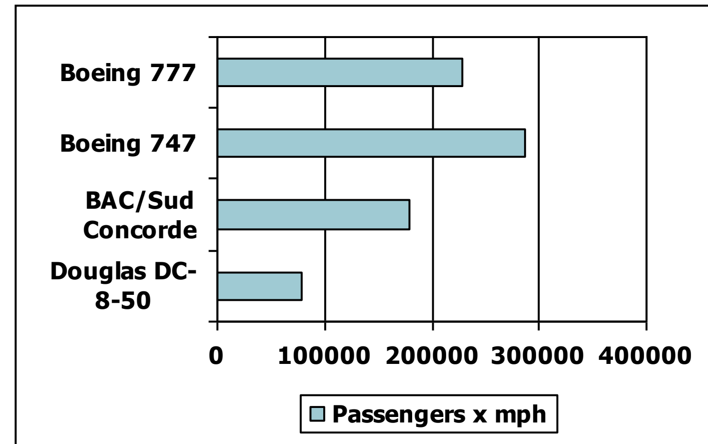
Defining Performance

- Which airplane has the best performance?









Response Time and Throughput

- Response or execution time
  - How long it takes to do a task
  - Individual users target reducing the response time
- Throughput
  - Total work done per unit time
    - e.g., tasks/transactions/… per hour
  - Datacenter managers target increasing throughput
- How are response time and throughput affected by
  - Replacing the processor with a faster version?
  - Adding more processors?
- We’ll focus on response time for now…

Relative Performance

- Define Performance = 1/Execution Time
- "X is n time faster than Y"

$$\text{Performance}_X / \text{Performance}_Y$$
$$= \text{Execution time}_Y / \text{Execution time}_X = n$$

- Example: time taken to run a program
  - 10s on A, 15s on B
  - Execution Time_B / Execution Time_A = 15s / 10s = 1.5
  - So A is 1.5 times faster than B

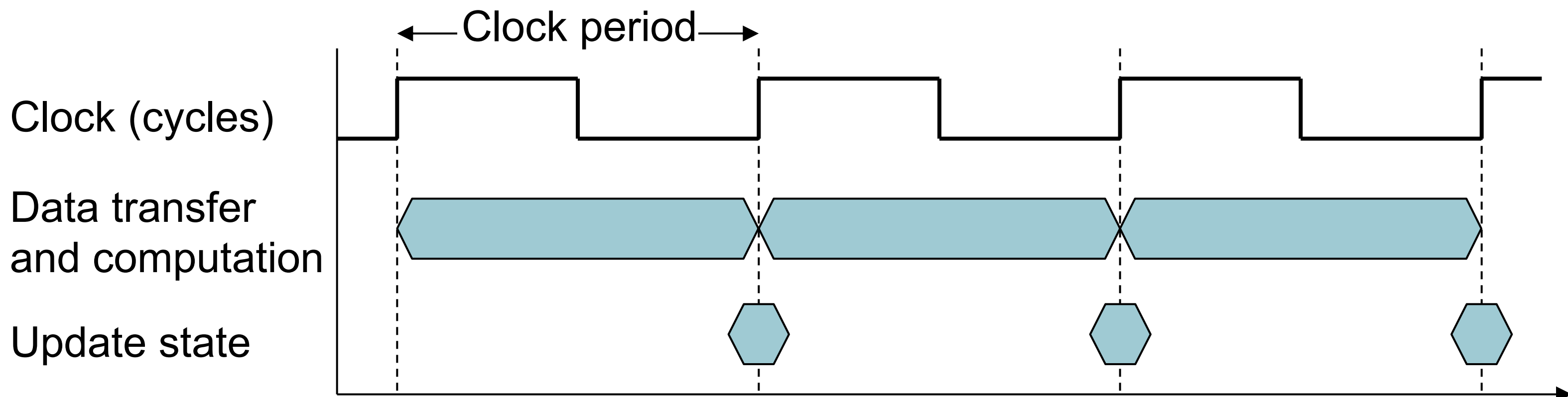
Measuring Execution Time

- Elapsed time (response or wall clock time)
  - Total response time, including all aspects
    - Processing, I/O, OS overhead, idle time
  - *System performance = 1 / Elapsed time*
- CPU time
  - Time spent processing a given job
    - Discounts I/O time, other jobs' shares
  - Comprises of:
    - User CPU time: CPU time spent in the program
    - System CPU time: CPU time spent in the OS performing tasks on behalf of the program
  - *CPU performance = 1 / User CPU time*
- Different programs are affected differently by CPU and system performance

MK MORGAN KAUFMANN

CPU Clocking

- Operation of digital hardware governed by a constant-rate clock



- Clock period: duration of a clock cycle
  - e.g., 250ps = 0.25ns = 250×10^{-12}s
- Clock frequency (rate): cycles per second
  - e.g., 4.0GHz = 4000MHz = 4.0×10^{9}Hz

CPU Time

$$\text{CPU Time} = \text{CPU Clock Cycles} \times \text{Clock Cycle Time}$$

$$= \frac{\text{CPU Clock Cycles}}{\text{Clock Rate}}$$

- Performance improved by
  - Reducing number of clock cycles
  - Increasing clock rate
  - Hardware designer must often trade off clock rate against cycle count

CPU Time Example

- Computer A: 2GHz clock, 10s CPU time
- Designing Computer B
  - Aim for 6s CPU time
  - Can do faster clock, but causes 1.2 × clock cycles
- How fast must Computer B clock be?

$$\text{Clock Rate}_B = \frac{\text{Clock Cycles}_B}{\text{CPU Time}_B} = \frac{1.2 \times \text{Clock Cycles}_A}{6\text{s}}$$

$$\text{Clock Cycles}_A = \text{CPU Time}_A \times \text{Clock Rate}_A$$

$$= 10\text{s} \times 2\text{GHz} = 20 \times 10^9$$

$$\text{Clock Rate}_B = \frac{1.2 \times 20 \times 10^9}{6\text{s}} = \frac{24 \times 10^9}{6\text{s}} = 4\text{GHz}$$

MK MORGAN KAUFMANN

Instruction Count and CPI

$$\text{Clock Cycles} = \text{Instruction Count} \times \text{Cycles per Instruction}$$

$$\text{CPU Time} = \text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time}$$

$$= \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}$$

- Instruction Count for a program
  - Determined by program, ISA and compiler
- Average cycles per instruction
  - Determined by CPU hardware
  - If different instructions have different CPI
    - Average CPI affected by instruction mix

CPI Example

- Computer A: Cycle Time = 250ps, CPI = 2.0
- Computer B: Cycle Time = 500ps, CPI = 1.2
- Same ISA and Complier
- Which is faster, and by how much?

$$\text{CPU Time}_A = \text{Instruction Count} \times \text{CPI}_A \times \text{Cycle Time}_A$$

$$= I \times 2.0 \times 250\text{ps} = I \times 500\text{ps}$$

A is faster…

$$\text{CPU Time}_B = \text{Instruction Count} \times \text{CPI}_B \times \text{Cycle Time}_B$$

$$= I \times 1.2 \times 500\text{ps} = I \times 600\text{ps}$$

$$\frac{\text{CPU Time}_B}{\text{CPU Time}_A} = \frac{I \times 600\text{ps}}{I \times 500\text{ps}} = 1.2$$

…by this much

CPI in More Detail

- If different instruction classes take different numbers of cycles

$$\text{Clock Cycles} = \sum_{i=1}^{n} (\text{CPI}_i \times \text{Instruction Count}_i)$$

- Weighted average CPI

$$\text{CPI} = \frac{\text{Clock Cycles}}{\text{Instruction Count}} = \sum_{i=1}^{n} \left(\text{CPI}_i \times \underbrace{\frac{\text{Instruction Count}_i}{\text{Instruction Count}}}_{\text{Relative frequency}} \right)$$

Relative frequency

MK MORGAN KAUFMANN

CPI Example

- Alternative compiled code sequences using instructions in classes A, B, C

Class	A	B	C
CPI for class	1	2	3
IC in sequence 1	2	1	2
IC in sequence 2	4	1	1

- Sequence 1: IC = 5
  - Clock Cycles
    $= 2 \times 1 + 1 \times 2 + 2 \times 3$
    $= 10$
  - Avg. CPI = 10/5 = 2.0
- Sequence 2: IC = 6
  - Clock Cycles
    $= 4 \times 1 + 1 \times 2 + 1 \times 3$
    $= 9$
  - Avg. CPI = 9/6 = 1.5

One More Example …

- Two compiled sequences of the same program are given below. The upper table gives the number of instructions from each type for sequence-1 and sequence-2. The lower table gives the CPI for each instruction type. Given that the two sequences are running on the **same computer**, What is the number of instructions of type B in sequence-2 that will make sequence-1 two times faster than sequence-2?

IC for each instruction class			
	A	B	C
Sequence-1	1	2	4
Sequence-2	2	?	2

	A	B	C
CPI	1	2	3

$$\frac{ET_2}{ET_1} = \frac{CPU\ Clock\ Cycles_2}{CPU\ Clock\ Cycles_1} = 2$$

$$\frac{2 \times 1 + ? \times 2 + 2\ \times 3}{1 \times 1 + 2 \times 2 + 4\ \times 3} = \frac{8 + ? \times 2}{17} = 2$$

$$8 + ? \times 2 = 34$$

$$? = \frac{34 - 8}{2} = 13$$

Performance Summary

The BIG Picture

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

- Performance depends on
  - Programming language, algorithm, compiler: affects IC and CPI
  - ISA: affects IC, CPI, and clock period (T_c)
  - Microarchitecture design: affects CPI
  - Hardware implementation and technology: affects clock period (T_c)

Concluding Remarks

- Cost/performance is improving
  - Due to underlying technology development
- Hierarchical layers of abstraction
  - In both hardware and software
- Instruction set architecture
  - The hardware/software interface
- Execution time: the best performance measure

Chapter 2

Instructions: Language of the Computer

Instruction Set

- The repertoire of instructions of a computer
- Instructions represent the computer language
  - Every language has vocabulary → Instruction Set
- Different computers have different instruction sets
  - But with many aspects in common (Why?)
    - All computers are constructed from hardware technologies based on similar underlying principles
    - There are a few basic operations that all computers must provide
- Early computers had very simple instruction sets
- Many modern computers also have simple instruction sets
- RISC: Reduced Instruction Set Computer
  - Fixed length, fast execution, and simple functionality
- CISC: Complex Instruction Set Computer
  - Variable length, slow execution, and complex functionality

Famous Commercial ISA

- MIPS is an elegant example of the instruction sets designed since the 1980s. (RISC)
- ARM instruction set from ARM Holdings plc introduced in 1985. (RISC)
  - More than 14 billion chips with ARM processors were manufactured in 2015, making them the most popular instruction sets in the world.
- Intel x86, which powers both the PC and the Cloud of the post-PC era. (CISC)

The RISC-V Instruction Set

- Used as the example throughout the book
- Developed at UC Berkeley as open ISA
- Now managed by the RISC-V Foundation (riscv.org)
- Typical of many modern ISAs
  - See RISC-V Reference Data tear-out card
- Similar ISAs have a large share of embedded core market
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …
- RISC-V has three design principles

Arithmetic Operations

- Each arithmetic instruction performs only ***one operation*** (Add, subtract, etc.) and must always have exactly ***three operands***
  - Two sources and one destination

```
add a, b, c  # a = b + c
```

- All arithmetic operations have this form
  - Example: We want to add four numbers (b, c, d, and e) and save the result in a.

```
add a, b, c
add a, a, d        → Three instructions are needed
add a, a, e
```

- ***Design Principle 1:*** **Simplicity favors regularity**
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

Arithmetic Example

- C code:

```
f = (g + h) - (i + j);
```

- Compiled RISC-V code:

```
add t0, g, h   // temp t0 = g + h
add t1, i, j   // temp t1 = i + j
sub f, t0, t1  // f = t0 - t1
```

MK MORGAN KAUFMANN

Register Operands

- Arithmetic instructions use register operands
- RISC-V has a 32 × 64-bit register file
  - Use for frequently accessed data
  - 64-bit data is called a "doubleword"
    - 32 x 64-bit general purpose registers x0 to x31
  - 32-bit data is called a "word"
- *Design Principle 2:* Smaller is faster
  - Variables in HLL are unlimited while there are only 32 registers in RISC-V
  - Designers must balance the need for more registers by the programmers and the desire to keep the clock rate fast (c.f. main memory: millions of locations)
  - Number of registers also affects the number of bits it would take to represent a specific register in the instruction format

RISC-V Registers

- x0: the constant value 0
- x1: return address
- x2: stack pointer
- x3: global pointer
- x4: thread pointer
- x5 – x7, x28 – x31: temporaries
- x8: frame pointer
- x9, x18 – x27: saved registers
- x10 – x11: function arguments/results
- x12 – x17: function arguments

Register Operand Example

- C code:

```
f = (g + h) - (i + j);
```

  - f, …, j in x19, x20, …, x23

- Compiled RISC-V code:

```
add x5, x20, x21
add x6, x22, x23
sub x19, x5, x6
```

Memory Operands

- Main memory used for composite data
  - Arrays, structures, dynamic data
- To apply arithmetic operations
  - Load values from memory into registers
  - Store result from register to memory
- RISC-V provides *data transfer* instructions to transfer data from memory to registers and vice-versa

Memory Organization

- Memory can be thought of as a single dimensional array
  - Memory[0] = 1, Memory[1] = 101, …
  - Index used with HLL

index	Data
⋮	⋮
3	100
2	10
1	101
0	1

- To access a double word (8 bytes) in memory, the instruction must provide the memory address for that doubleword
- Memory is byte addressed
  - Each address identifies an 8-bit byte
- RISC-V is Little Endian
  - Least-significant byte at least address of a word
  - *c.f.* Big Endian: most-significant byte at least address

Byte Address	Data
⋮	⋮
24	100
16	10
8	101
0	1

- RISC-V does not require words/doublewords to be aligned in memory
  - Unlike some other ISAs

Data Transfer Instructions

- Load: copies data from a memory location to a register
  - ld ≡ load doubleword
  - ld x28, 32 (x18) ≡ (x28) ← Memory [32 + (x18)]

offset — base register

- Store: copies data from a register to a memory location
  - sd ≡ store doubleword
  - sd x28, 32 (x18) ≡ Memory [32 + (x18)] ← (x28)

offset — base register

Memory Operand Example 1

- C code:

```
g = h + A[8];
```

  - g in x20, h in x21, base address of A in x22
- Compiled RISC-V code:
  - Index 8 requires offset of 64
    - 8 bytes per doubleword

```
ld  x9, 64(x22)  # load doubleword
add x20, x21, x9
```

offset

base register

Memory Operand Example 2

- C code:

```
A[12] = h + A[8];
```

  - h in x21, base address of A in x22
- Compiled RISC-V code:
  - Index 8 requires offset of 64
  - Index 12 requires offset of 96
    - 8 bytes per doubleword

```
ld      x9, 64(x22)
add     x9, x21, x9
sd      x9, 96(x22)
```

Registers vs. Memory

- Registers are faster to access than memory and consume less energy
- Operating on memory data requires loads and stores
  - More instructions to be executed
- Compiler must use registers for variables as much as possible to get better performance and conserve energy
  - Only spill to memory for less frequently used variables (i.e. register spilling)
  - Register optimization is important!

Immediate Operands

- Constant data specified in an instruction

```
addi x22, x22, 4
```

- No subtract immediate instruction
  - Just use a negative constant

  ```
  addi x22, x22, -1
  ```

- Example of the great idea “Make the common case fast”
  - Small constants are common
    - More than half of the RISC-V arithmetic instructions have constants as variables
  - Immediate operand avoids a load instruction

The Constant Zero

- RISC-V dedicates register "x0" to be hard-wired to the value zero
  - Cannot be overwritten
- Useful for common operations
  - E.g., negate the value in a register

```
sub x22, x0, x21
```

Binary Integers

- Humans use decimal system (Base = 10)
  - Digits: 0, 1, 2, …, 9
- In computers, numbers are stored as series of high and low electronic signals → Binary system (Base = 2)
  - Digits: 0, 1
  - Digit = bit
- In any number with base (r), the value of the ith digit (d)

$$value = d \times r^{i}$$

  - i starts from 0 and increases from right to left

Unsigned Binary Integers

- Given an n-bit number

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Range: 0 to $+2^n - 1$
- Example
  - $0000\ 0000 \ldots 0000\ 1011_2$
    $= 0 + \ldots + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$
    $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$
- RISC-V uses 64 bits
  - There are 2^{64} combinations
  - 0 to $(2^{64} - 1)$ → 0 to +18,446,774,073,709,551,615
  - Bit 0 is the least significant bit
  - Bit 63 is the most significant bit

2s-Complement Signed Integers

- Given an n-bit number

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Range: (-2^{n-1}) to $(+2^{n-1} - 1)$
- Example
  - $1111\ 1111\ \ldots\ 1111\ 1100_2$
    $= -1\times2^{31} + 1\times2^{30} + \ldots + 1\times2^2 + 0\times2^1 + 0\times2^0$
    $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$
- Using 64 bits: −9,223,372,036,854,775,808
  to 9,223,372,036,854,775,807

2s-Complement Signed Integers

- Bit 63 is sign bit
  - 1 for negative numbers
  - 0 for non-negative numbers
- $-(-2^{n-1})$ can't be represented
- Non-negative numbers have the same unsigned and 2s-complement representation
- Some specific numbers
  - 0: 0000 0000 … 0000
  - –1: 1111 1111 … 1111
  - Most-negative: 1000 0000 … 0000
  - Most-positive: 0111 1111 … 1111

MK MORGAN KAUFMANN

2s-Complement Characteristics

- Simple hardware can be used for both signed and unsigned numbers
  - Why not always signed?
    - Some computation deals with numbers as unsigned. For example, memory addresses start at 0 and continue to the largest address. In other words, negative addresses make no sense
- Leading 0 means positive number and leading 1 means negative number
  - So, hardware treats this bit as sign bit
- Single zero representation
- Imbalance between positive and negative numbers
- Overflow occurs when the sign bit is incorrect

Signed Negation in 2s-Complement

- First approach: Complement and add 1
  - Complement means 1 → 0, 0 → 1

$$x + \bar{x} = 1111...111_2 = -1$$
$$\bar{x} + 1 = -x$$

- Second approach:
  - Start from the right
  - Search for the first bit with value 1
  - Complement all bits to the left of the rightmost bit with value 1
- Example: negate +2
  - $+2 = 0000\ 0000\ \ldots\ 0010_2$
  - $-2 = 1111\ 1111\ \ldots\ 1101_2 + 1$
    $= 1111\ 1111\ \ldots\ 1110_2$
- Why 2s-Complement is called like this?

$$x + (-x) = 10000\ldots.000_2 = 2^n$$
$$-x = 2^n - x$$

Sign Extension

- Representing a number using more bits
  - Preserve the numeric value
- Replicate the sign bit to the left
  - c.f. unsigned values: extend with 0s
- Examples: 8-bit to 16-bit
  - +2: 0000 0010 => 0000 0000 0000 0010
  - –2: 1111 1110 => 1111 1111 1111 1110
- In RISC-V instruction set
  - `lb`: sign-extend loaded byte
  - `lbu`: zero-extend loaded byte

Representing Instructions

- Instructions are encoded in binary
  - Called machine code
- RISC-V instructions
  - Encoded as 32-bit instruction words
  - Small number of formats encoding operation code (opcode), register numbers, …
  - Regularity!

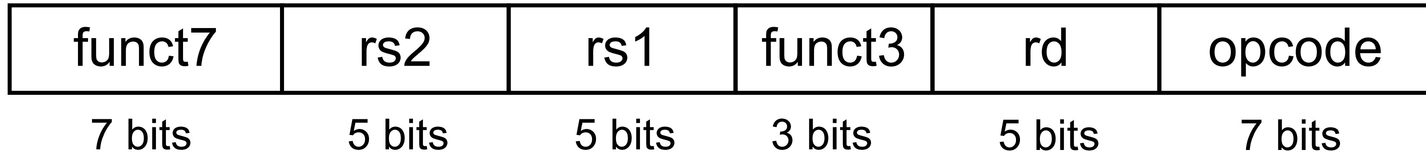
Hexadecimal

- Base 16
  - Compact representation of bit strings
  - 4 bits per hex digit

0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

- Example: eca8 6420
  - 1110 1100 1010 1000 0110 0100 0010 0000

RISC-V R-format Instructions



- Instruction fields
  - opcode: operation code
  - rd: destination register number
  - funct3: 3-bit function code (additional opcode)
  - rs1: the first source register number
  - rs2: the second source register number
  - funct7: 7-bit function code (additional opcode)

R-format Example

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

```
add x9,x20,x21
```

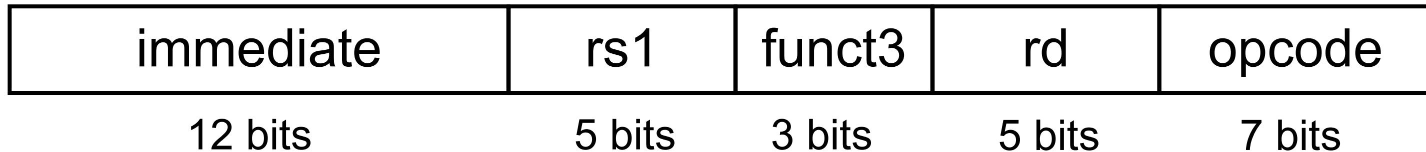
0	21	20	0	9	51

0000000	10101	10100	000	01001	0110011

$0000\ 0001\ 0101\ 1010\ 0000\ 0100\ 1011\ 0011_{two} =$

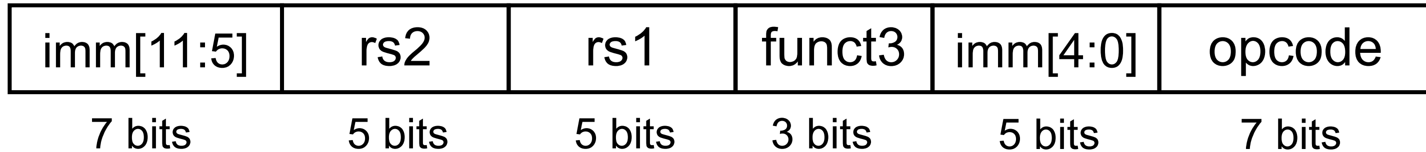
$015A04B3_{16}$

RISC-V I-format Instructions



- Immediate arithmetic and load instructions
  - rs1: source or base address register number
  - immediate: constant operand, or offset added to base address
    - 2s-complement, **sign extended**, (-2^{11}) to ($+2^{11} - 1$)
  - The load doubleword instruction can refer to any doubleword within a region of $\pm 2^{11}$ or 2048 bytes ($\pm 2^{8}$ or 256 doublewords) of the base address in the base register rs1
- *Design Principle 3:* Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible

RISC-V S-format Instructions



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

Instructions Opcodes

Instruction	Format	funct7	rs2	rs1	funct3	rd	opcode
add (add)	R	0000000	reg	reg	000	reg	0110011
sub (sub)	R	0100000	reg	reg	000	reg	0110011

Instruction	Format	immediate	rs1	funct3	rd	opcode
addi (add immediate)	I	constant	reg	000	reg	0010011
ld (load doubleword)	I	address	reg	011	reg	0000011

Instruction	Format	immed -iate	rs2	rs1	funct3	immed -iate	opcode
sd (store doubleword)	S	address	reg	reg	011	address	0100011

- Opcode helps to distinguish which format should be used

Instruction Representation Example

- Translate the C-statement below to RISC-V machine code:
  - A[30] = h + A[30] + 1;
  - Assume that x10 contains the base address of A and h is mapped to x21
- Solution:
  - First: translate to RISC-V assembly

```
ld x9, 240 (x10)
add x9, x21, x9
addi x9, x9, 1
sd x9, 240 (x10)
```

  - Second: translate to RISC-V machine code

immediate	rs1	funct3	rd	opcode
240	10	3	9	3

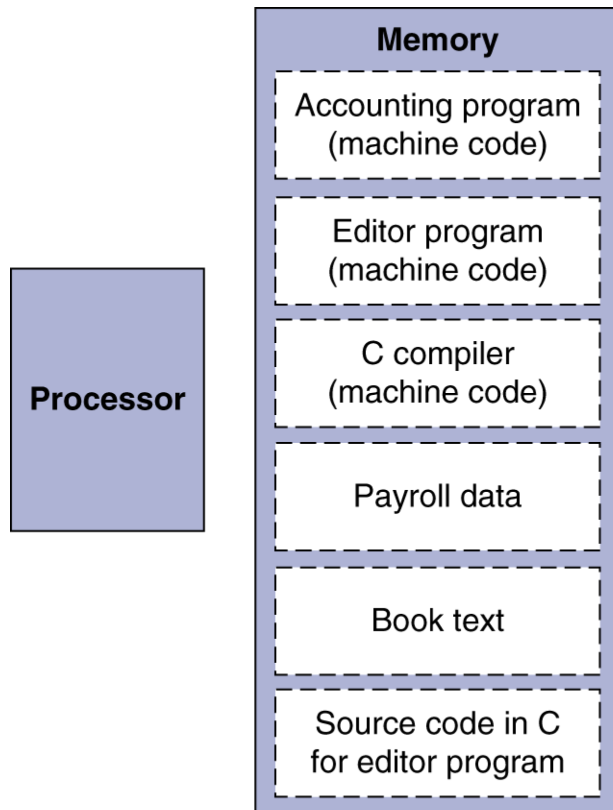
funct7	rs2	rs1	funct3	rd	opcode
0	9	21	0	9	51

immediate	rs1	funct3	rd	opcode
1	9	0	9	19

immediate[11:5]	rs2	rs1	funct3	immediate[4:0]	opcode
7	9	10	3	16	35

Stored Program Computers

The BIG Picture



- Instructions represented in binary, just like data
- Instructions and data stored in memory
- Programs can operate on programs
  - e.g., compilers, linkers, …
- Binary compatibility allows compiled programs to work on different computers
  - Standardized ISAs

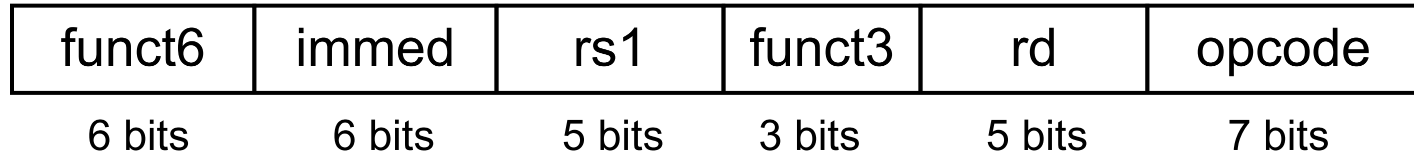
Logical Operations

- Instructions for bitwise manipulation

Operation	C	Java	RISC-V
Shift left	<<	<<	`sll, slli`
Shift right	>>	>>>	`srl, srli`
Bit-by-bit AND	&	&	`and, andi`
Bit-by-bit OR	|	|	`or, ori`
Bit-by-bit XOR	^	^	`xor, xori`
Bit-by-bit NOT	~	~	`xori`

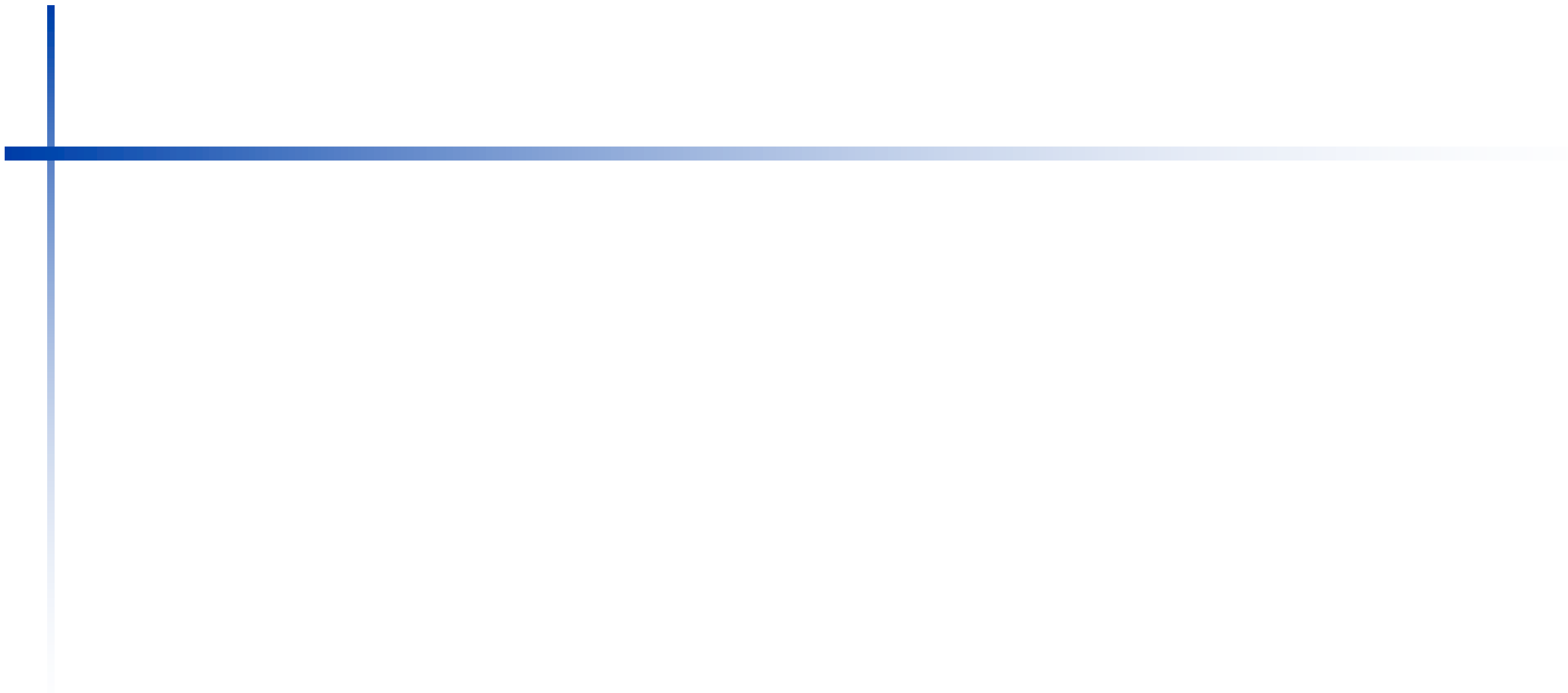
- Useful for extracting and inserting groups of bits in a word

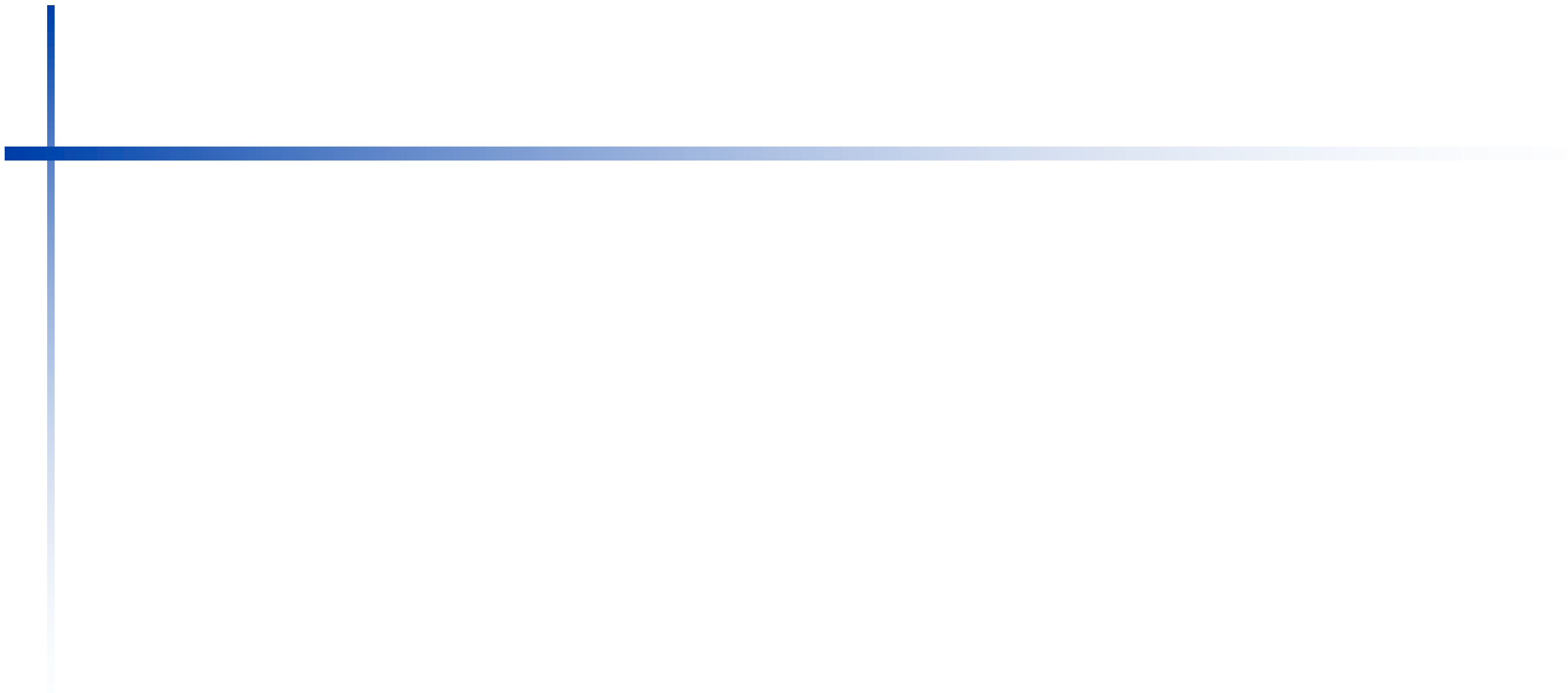
Shift Immediate Operations



funct6	immed	rs1	funct3	rd	opcode
6 bits	6 bits	5 bits	3 bits	5 bits	7 bits

- immed: how many positions to shift
- Shift left logical
  - Shift left and fill with 0 bits
  - `slli` by *i* bits multiplies by 2^i
- Shift right logical
  - Shift right and fill with 0 bits
  - `srli` by *i* bits divides by 2^i (unsigned only)





AND Operations

- Useful to mask bits in a word
  - Select some bits, clear others to 0

```
and x9,x10,x11
```

x10	00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000
x11	00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000
x9	00000000 00000000 00000000 00000000 00000000 00000000 00001100 00000000

- There is an AND Immediate instruction (andi)
  - Sign-extension for the immediate value

```
andi x9, x10, 0xFFF
```

OR Operations

- Useful to include bits in a word
  - Set some bits to 1, leave others unchanged

```
or x9,x10,x11
```

x10	00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000
x11	00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000
x9	00000000 00000000 00000000 00000000 00000000 00000000 00111101 11000000

- There is an OR Immediate instruction (ori)
  - Sign-extension for the immediate value

```
ori x9, x10, 0xFFF
```

XOR Operations

- Differencing operation
  - Invert some bits, leave others unchanged

```
xor x9,x10,x11  // NOT operation
```

x10	00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000
x11	11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111
x9	11111111 11111111 11111111 11111111 11111111 11111111 11110010 00111111

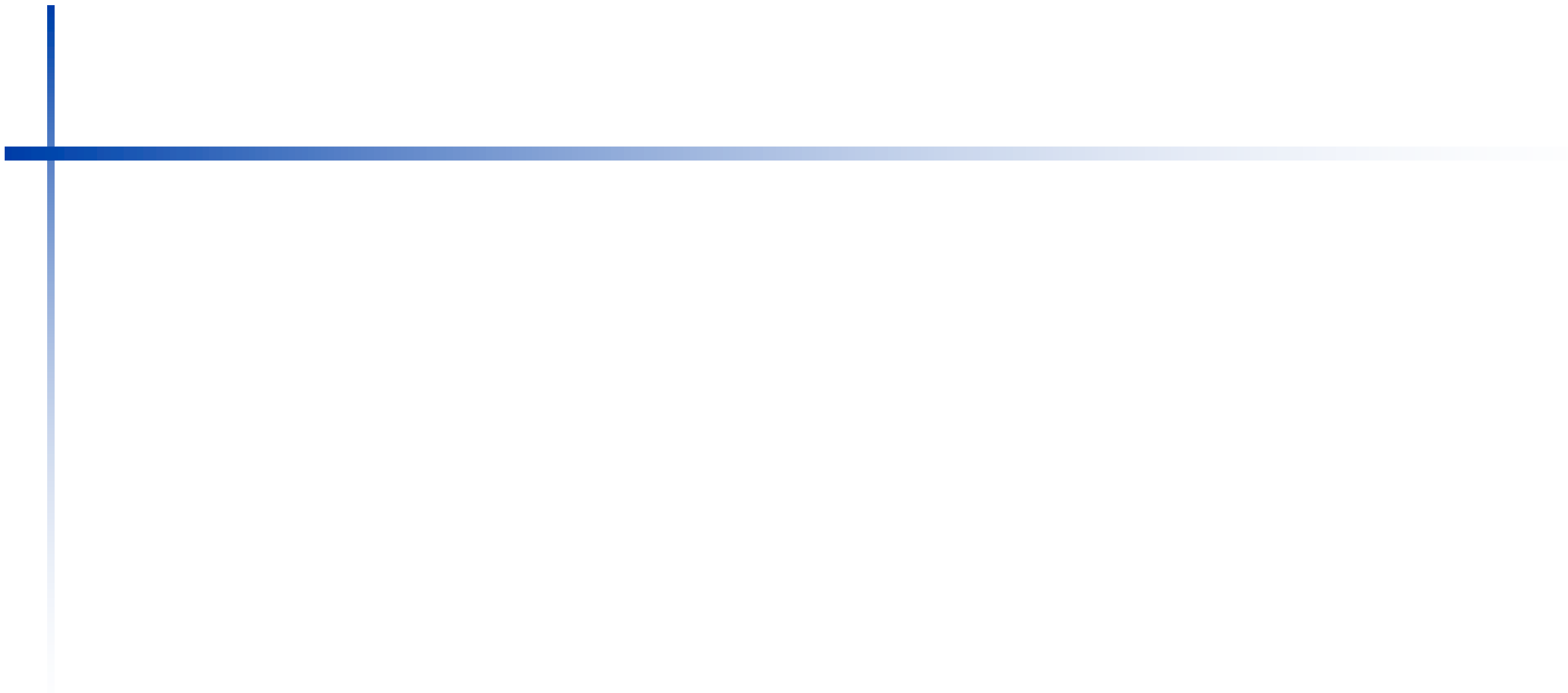
- There is an XOR Immediate instruction (xori)
  - Sign-extension for the immediate value

```
xori x9, x10, 0xFFF
```

MK MORGAN KAUFMANN

Conditional Operations

- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially
- `beq rs1, rs2, L1`
  - if (rs1 == rs2) branch to instruction labeled L1
- `bne rs1, rs2, L1`
  - if (rs1 != rs2) branch to instruction labeled L1



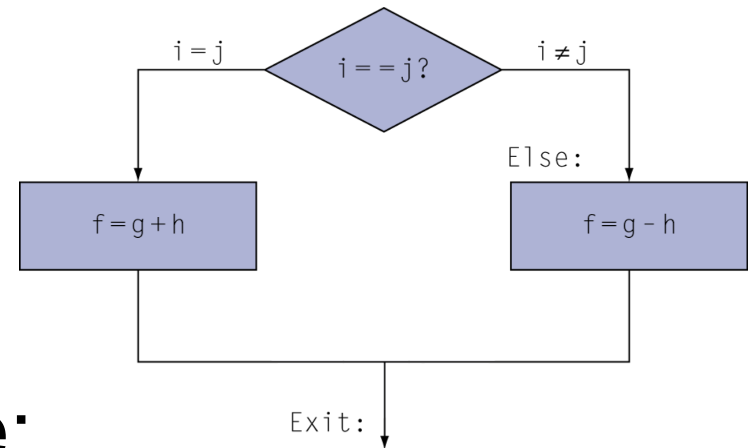
MORGAN KAUFMANN

Compiling If Statements

- C code:

```
if (i==j) f = g+h;
else f = g-h;
```

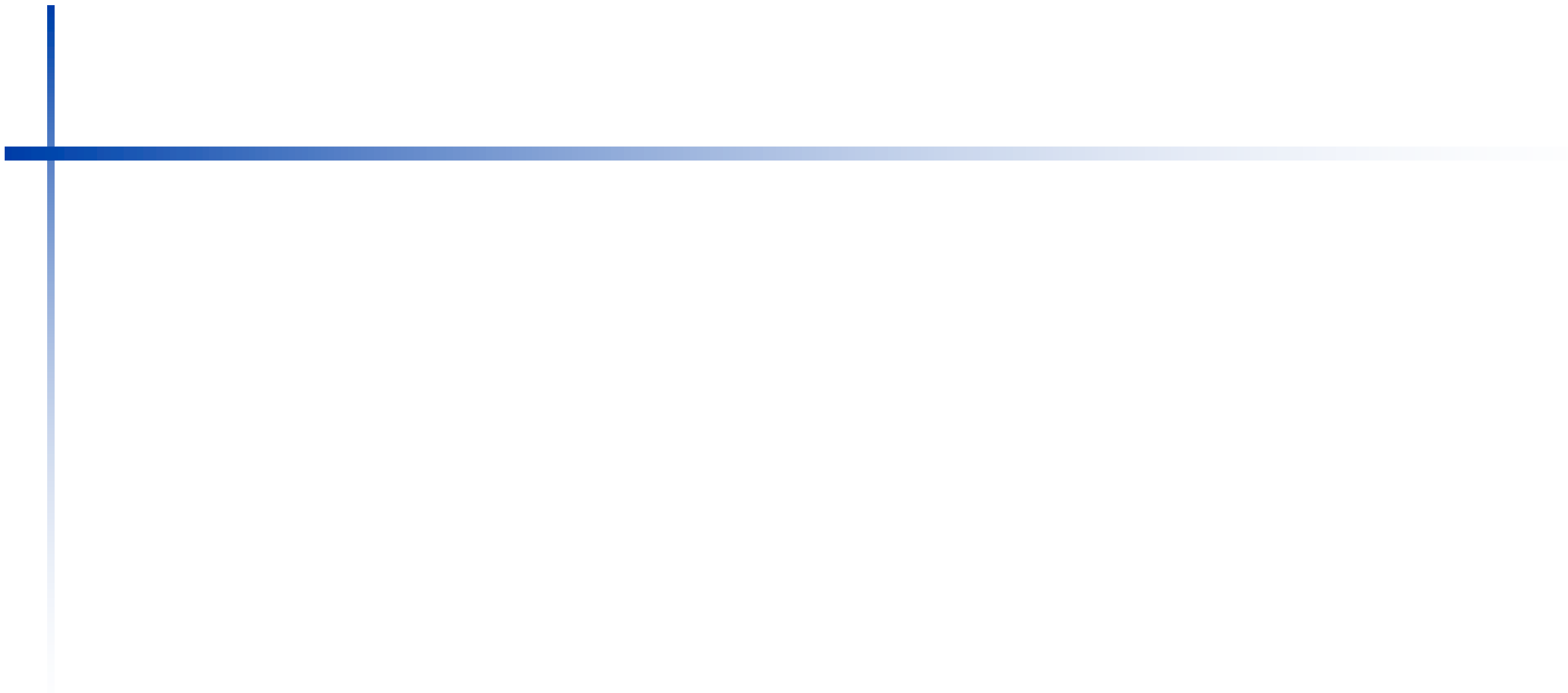
  - f, g, … in x19, x20, …



- Compiled RISC-V code:

```
      bne x22, x23, Else
      add x19, x20, x21
      beq x0,x0,Exit // unconditional
Else: sub x19, x20, x21
Exit: …
```

Assembler calculates addresses



Compiling Loop Statements

- C code:

```
while (save[i] == k) i += 1;
```

  - i in x22, k in x24, address of save in x25

- Compiled RISC-V code:

```
Loop: slli x10, x22, 3
      add  x10, x10, x25
      ld   x9, 0(x10)
      bne  x9, x24, Exit
      addi x22, x22, 1
      beq  x0, x0, Loop
Exit: …
```

More Conditional Operations

- `blt rs1, rs2, L1`
  - if (rs1 < rs2) branch to instruction labeled L1
- `bge rs1, rs2, L1`
  - if (rs1 >= rs2) branch to instruction labeled L1
- Example
  - if (a > b) a += 1;
  - a in x22, b in x23

```
    bge  x23, x22, Exit      // branch if b >= a
    addi x22, x22, 1
Exit:
```

Signed vs. Unsigned

- Signed comparison: blt, bge
- Unsigned comparison: bltu, bgeu
- Example
  - x22 = 1111 1111 1111 1111 1111 1111 1111 1111
  - x23 = 0000 0000 0000 0000 0000 0000 0000 0001
  - `x22 < x23 // signed`
    - –1 < +1
  - `x22 > x23 // unsigned`
    - +4,294,967,295 > +1

Bounds Check Shortcut

- Treating signed numbers as if they were unsigned gives us a low cost way of checking if $0 \leq x < y$, which matches the index out-of-bounds check for arrays.

Example

Use this shortcut to reduce an index-out-of-bounds check: branch to `IndexOutOfBounds` if `x20` ≥ `x11` or if `x20` is negative.

Answer

The checking code just uses unsigned greater than or equal to do both checks:

```
  bgeu x20, x11, IndexOutOfBounds // if x20 >= x11 or x20 < 0,
goto IndexOutOfBounds
```

MK MORGAN KAUFMANN

Compiling Loop Statements

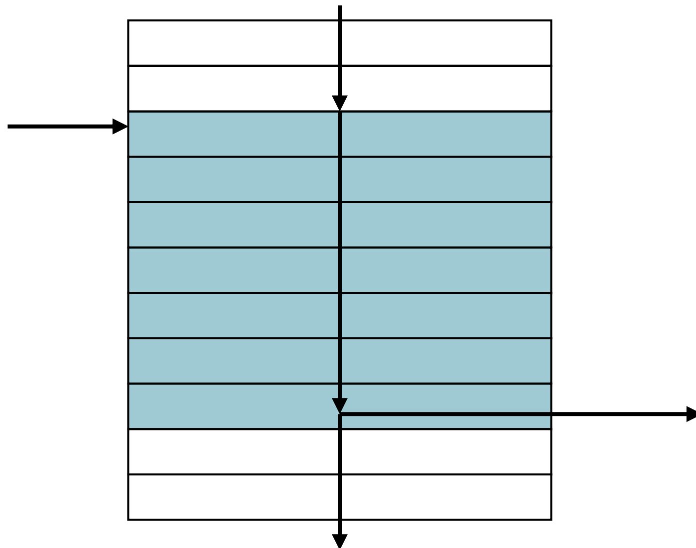
- For Loop (C code):

```
for (i = 0; i < 10; i++)
save[i] = save[i] * 2
```

- Case/Switch:
  - Simplest way is via sequence of conditional tests → chain of if-else statements
  - Or use branch address table (branch table)
    - Array of double words containing addresses that corresponds to labels in the code
    - The program loads the appropriate entry from the branch table into a register.
    - Then use jump-and-link register (jalr) instruction (Unconditional)

Basic Blocks

- A basic block is a sequence of instructions with
  - No embedded branches (except at end)
  - No branch targets (except at beginning)



- A compiler identifies basic blocks for optimization
- An advanced processor can accelerate execution of basic blocks

Procedure Calling

- Steps required
  1. Place parameters in registers x10 to x17
  2. Transfer control to procedure
  3. Acquire storage for procedure
  4. Perform procedure’s operations
  5. Place result in register for caller
  6. Return to place of call (address in x1)

Procedure Call Instructions

- Procedure call: jump and link

  `jal x1, ProcedureLabel`

  - Address of following instruction (PC+4) put in x1
  - Jumps to target address
  - Can also be used for unconditional branch
    - e.g., jal x0, Label (x0 cannot be changed)

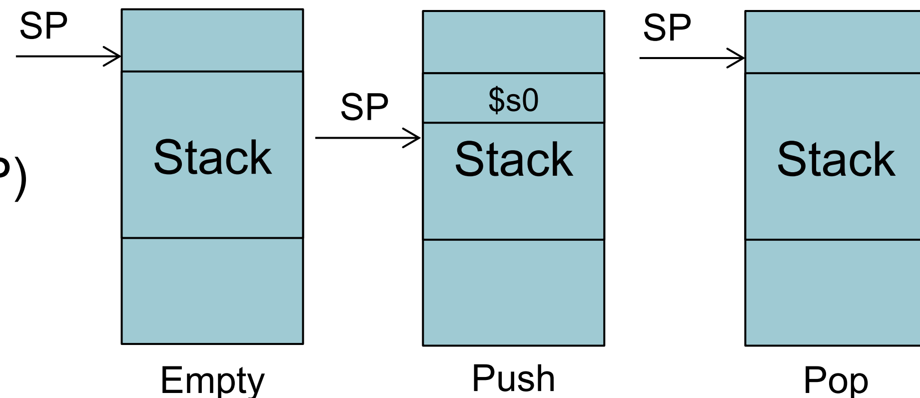
- Procedure return: jump and link register

  `jalr x0, 0(x1)`

  - Like jal, but jumps to 0 + address in x1
  - Use x0 as rd (x0 cannot be changed)
  - Can also be used for computed jumps
    - e.g., for case/switch statements

Using More Registers: Stack

- Stack is needed to:
  - Spill registers during procedure execution
    - More registers are needed for execution other than the eight argument registers
    - Saving return address or arguments (Non-leaf procedures)
  - Define local arrays or structures inside the procedure
- Stack is part of the program memory space and it grows in the direction of decreasing addresses (i.e. from higher to lower addresses
  - Last-In First-Out (LIFO)
  - Access using stack pointer (SP) which is register x2
  - SP always points to the top of the stack
  - Push → Store in Stack (Subtract from SP)
  - Pop → Load from Stack (Add to SP)



Leaf Procedure Example

- C code:

```
long long int leaf_example (
    long long int g, long long int h,
    long long int i, long long int j) {
  long long int f;
  f = (g + h) - (i + j);
  return f;
}
```

  - Arguments g, …, j in x10, …, x13
  - f in x20
  - temporaries x5, x6
  - Result in x10

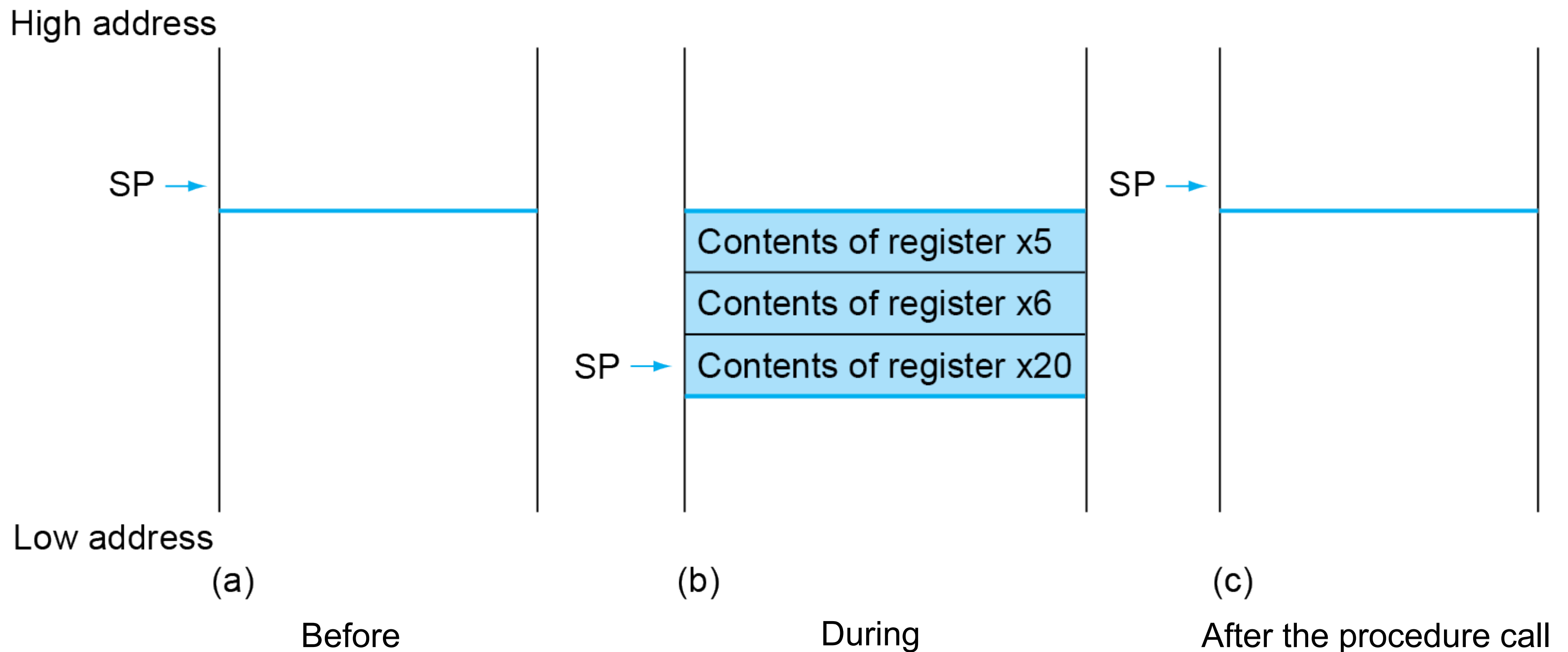
MK MORGAN KAUFMANN

Leaf Procedure Example

- RISC-V code:

```
leaf_example:
  addi sp,sp,-24        Save x5, x6, x20 on stack
  sd   x5,16(sp)
  sd   x6,8(sp)
  sd   x20,0(sp)
  add  x5,x10,x11       x5 = g + h
  add  x6,x12,x13       x6 = i + j
  sub  x20,x5,x6        f = x5 – x6
  addi x10,x20,0        copy f to return register
  ld   x20,0(sp)        Restore x5, x6, x20 from stack
  ld   x6,8(sp)
  ld   x5,16(sp)
  addi sp,sp,24
  jalr x0,0(x1)         Return to caller
```

Local Data on the Stack



Register Usage

- x5 – x7, x28 – x31: temporary registers
  - Not preserved by the callee
- x8 – x9, x18 – x27: saved registers
  - If used, the callee saves and restores them
- **In the previous example, the caller does not expect x5 and x6 to be preserved. Hence, we can drop two stores and two loads.**
- **We still must save and restore x20**

Non-Leaf Procedures

- Procedures that call other procedures
- For nested call, caller needs to save on the stack:
  - Its return address
  - Any arguments and temporaries needed after the call
- Restore from the stack after the call

Non-Leaf Procedure Example

- C code:

```
long long int fact (long long int n)
{
  if (n < 1) return 1;
  else return n * fact(n - 1);
}
```

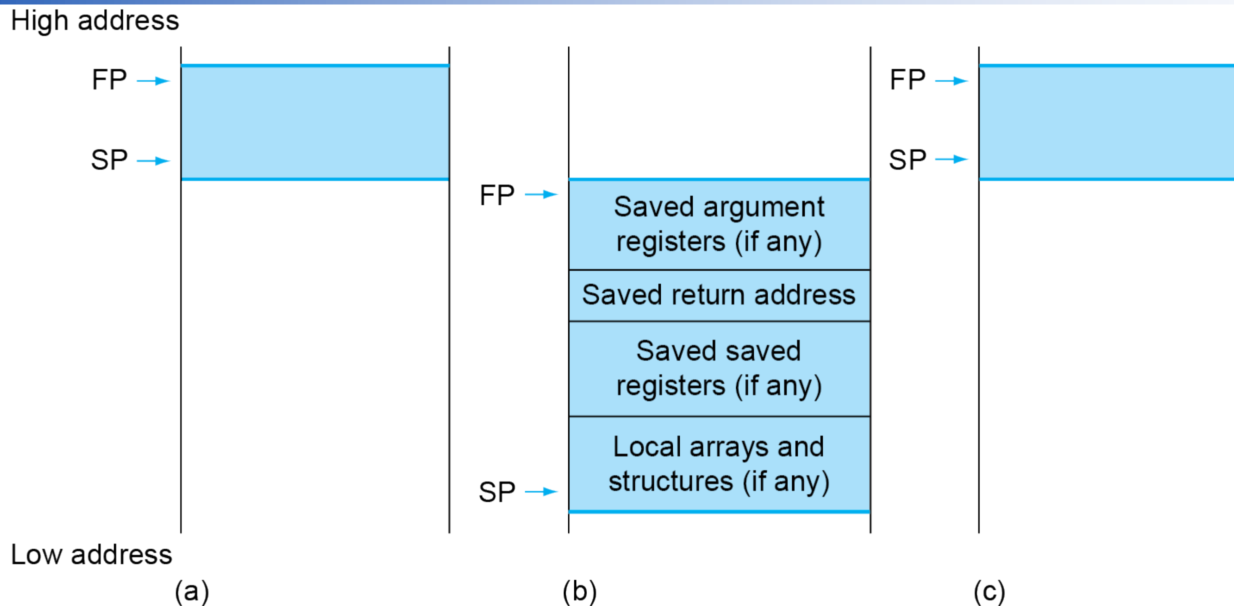
  - Argument n in x10
  - Result in x10

Non-Leaf Procedure Example

- RISC-V code:

```
fact:
    addi sp,sp,-16          Save return address and n on stack
    sd   x1,8(sp)
    sd   x10,0(sp)
    addi x5,x10,-1          x5 = n - 1
    bge  x5,x0,L1           if n >= 1, go to L1
    addi x10,x0,1           Else, set return value to 1
    addi sp,sp,16           Pop stack, don't bother restoring values
    jalr x0,0(x1)           Return
L1: addi x10,x10,-1         n = n - 1
    jal  x1,fact            call fact(n-1)
    addi x6,x10,0           move result of fact(n - 1) to x6
    ld   x10,0(sp)          Restore caller's n
    ld   x1,8(sp)           Restore caller's return address
    addi sp,sp,16           Pop stack
    mul  x10,x10,x6         return n * fact(n-1)
    jalr x0,0(x1)           return
```

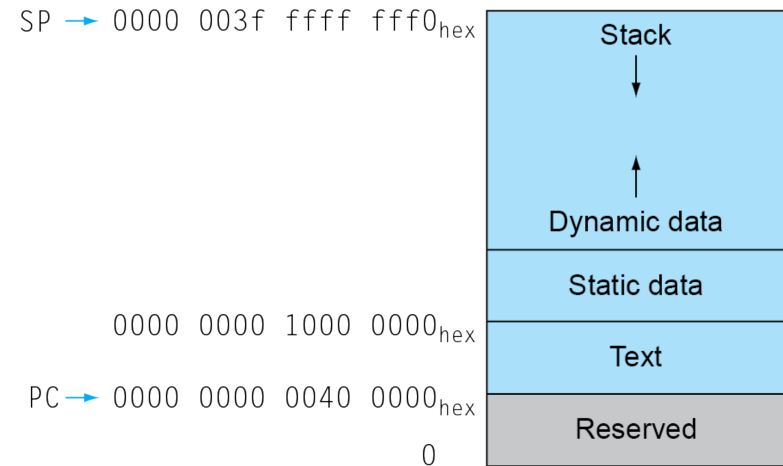
Local Data on the Stack



- Local data allocated by callee
  - Local to the procedure but don't fit in the registers (e.g. local arrays and structures)
  - Similar to C automatic variables
- Procedure frame (activation record)
  - The part of the stack containing the procedure saved registers and local variables
  - The Frame Pointer (FP), which is register x8, points to the 1st double word of the procedure frame and offers a stable base within a procedure for local memory references

Memory Layout

- Text: program code
- Static data: global variables
  - e.g., static variables in C, constant arrays and strings
  - x3 (global pointer) initialized to address allowing ±offsets into this segment
- Dynamic data: heap
  - E.g., malloc in C, new in Java
  - De-allocation in C using free(); otherwise, there will be memory leak or dangling pointers
  - Java uses automatic garbage collection
- Stack: automatic storage



Iteration Vs. Recursion

- Some recursive procedures can be implemented iteratively without recursion
- Example: C code:

```
long long int sum (long
   long int n, long long int
   acc)
{
if (n > 0)
   return sum(n – 1, acc +
   n);
else
   return acc;
}
```

- `n is x10, acc is x11, and result in x12`

Iteration	Recursion

Sum:
bge x0, x10, Exit
add x11, x11, x10
addi x10, x10, -1
jal x0, Sum
Exit:
add x12, x11, x0
jalr x0, 0(x1)
	Sum:
	addi sp, sp, -8
	sd x1, 0(sp)
	bge x0, x10, Else
	add x11, x11, x10
	addi x10, x10, -1
	jal x1, Sum
	beq x0, x0, Exit
	Else:
	addi x12, x11, 0
	Exit:
	ld x1, 0(sp)
	addi sp, sp, 8
	jalr x0, 0(x1)

RISC-V Register Conventions

Name	Register number	Usage	Preserved on call?
x0	0	The constant value 0	n.a.
x1 (ra)	1	Return address (link register)	yes
x2 (sp)	2	Stack pointer	yes
x3 (gp)	3	Global pointer	yes
x4 (tp)	4	Thread pointer	yes
x5-x7	5–7	Temporaries	no
x8-x9	8–9	Saved	yes
x10-x17	10–17	Arguments/results	no
x18-x27	18–27	Saved	yes
x28-x31	28–31	Temporaries	no

- Example of "Make the Common Case Fast": 12 saved, 7 temporary, and 8 argument is sufficient most of the time

Character Data

- Byte-encoded character sets
  - ASCII: 128 characters
    - ASCII: American Standard Code for Information Interchange
    - 95 graphic, 33 control
    - Size of (1000000000) in ASCII = 10 char * 8 bits = 80 bits
    - Size of (1000000000) in Binary = 32 bits
  - Latin-1: 256 characters
    - ASCII, +96 more graphic characters
- Unicode: 32-bit character set
  - Used in Java, C++ wide characters, …
  - Most of the world's alphabets, plus symbols
  - UTF-8, UTF-16: variable-length encodings

Byte/Halfword/Word Operations

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

String Copy Example

- C code:
  - Null-terminated string

```
void strcpy (char x[], char y[])
{ size_t i;
  i = 0;
  while ((x[i]=y[i])!='\0')
    i += 1;
}
```

  - Base addresses of arrays x, y are in x10, x11
  - i is in x19

String Copy Example

- RISC-V code:

```
strcpy:
     addi sp,sp,-8        // adjust stack for 1 doubleword
     sd   x19,0(sp)       // push x19
     add  x19,x0,x0       // i=0
L1:  add  x5,x19,x11      // x5 = addr of y[i]
     lbu  x6,0(x5)        // x6 = y[i]
     add  x7,x19,x10      // x7 = addr of x[i]
     sb   x6,0(x7)        // x[i] = y[i]
     beq  x6,x0,L2        // if y[i] == 0 then exit
     addi x19,x19,1       // i = i + 1
     jal  x0,L1           // next iteration of loop
L2:  ld   x19,0(sp)       // restore saved x19
     addi sp,sp,8         // pop 1 doubleword from stack
     jalr x0,0(x1)        // and return
```

32-bit Constants (1)

- Most constants are small
  - 12-bit immediate is sufficient
- For the occasional 32-bit constant: `lui rd, constant`
  - Copies 20-bit constant to bits [31:12] of rd
  - Extends bit 31 to bits [63:32]
  - Clears bits [11:0] of rd to 0
- Example: Write RISC-V instructions to load 0x003D0500 in x19.

```
lui x19, 976  // 0x003D0
```

0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0011 1101 0000	0000 0000 0000

```
addi x19,x19,128  // 0x500
```

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

32-bit Constants (2)

- Example: Write RISC-V instructions to load 0x003D0800 in x19.

```
lui x19, 977   // 0x003D1
```

0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0011 1101 0001	0000 0000 0000

```
addi x19,x19,-2048  // 0x800
```

0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0011 1101 0000	1000 0000 0000

Branch Addressing

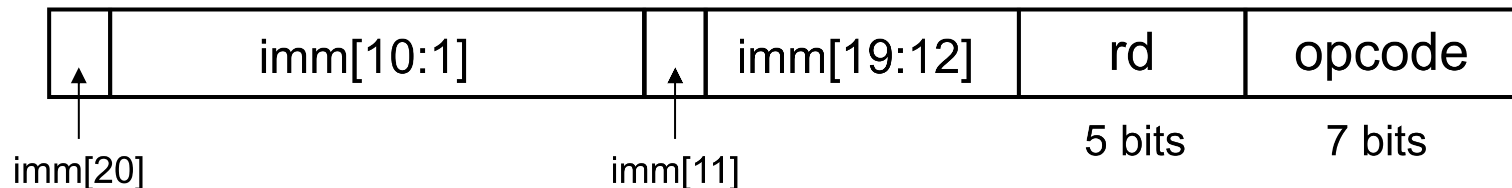
- Branch instructions specify
  - Opcode, two registers, target address
- Most branch targets are near branch
  - Forward or backward
- SB format:

imm[12]	imm [10:5]	rs2	rs1	funct3	imm [4:1]	imm[11]	opcode

- PC-relative addressing
  - Target address = PC + immediate × 2
- The immediate represents the offset in **halfwords**

Unconditional Jump-and-Link Addressing

- Jump and link (jal) target uses 20-bit immediate for larger range
- UJ format: Only (jal) uses this format



- PC-relative addressing
  - Target address = PC + immediate × 2
- The immediate represents the offset in **halfwords**
- The unusual encoding in SB and UJ formats simplifies datapath design but complicates assembly

PC-relative Addressing

- If absolute addresses were to fit in the 20-bit immediate field, then no program could be bigger than 2^{20} bytes, which is far too small to be a realistic option today
- An alternative is to specify a register that would always be added to the branch/jal offset:
  - Allows the program to be as large as 2^{64} bytes
  - PC contains the address of the current instruction
  - PC is the ideal choice
- The RISC-V architects wanted to support the possibility of instructions that are only 2 bytes long (i.e. the offset is in halfwords)
  - Maximum offset for branch is $\pm 2K\ halfwords = \pm 4K\ Bytes$
  - Maximum offset for jal is $\pm 512K\ halfwords = \pm 1M\ Bytes$

Branch Offset in Machine Language

- Loop code from earlier example
  - Assume Loop at location 80000

```
Loop:slli x10, x22, 3
 add x10, x10, x25
 ld x9, 0(x10)
 bne x9, x24, Exit
 addi x22, x22, 1
 beq x0, x0, Loop
Exit:
```

Address	Instruction					
80000	0000000	00011	10110	001	01010	0010011
80004	0000000	11001	01010	000	01010	0110011
80008	0000000	00000	01010	011	01001	0000011
80012	0000000	11000	01001	001	01100	1100011
80016	0000000	00001	10110	000	10110	0010011
80020	1111111	00000	00000	000	01101	1100011

Long Jumps and Branching Far Away

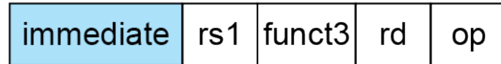
- For long jumps, e.g., to 32-bit absolute address of a procedure
  - lui: load address[31:12] to temp register
  - jalr: add address[11:0] and jump to target
- If branch target is too far to encode with 12-bit offset, assembler rewrites the code
  - Example

```
        beq x10,x0, L1
                  ↓
        bne x10,x0, L2
        jal x0, L1
L2:     …
```

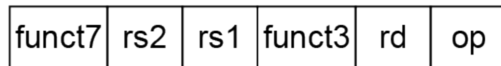
MK MORGAN KAUFMANN

RISC-V Addressing Summary

1. Immediate addressing



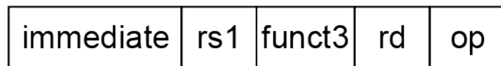
2. Register addressing



Registers

Register

3. Base addressing



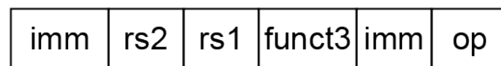
Register

+

Memory

Byte Halfword Word Doubleword

4. PC-relative addressing



PC

+

Memory

Word

RISC-V Encoding Summary

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

Chapter 3

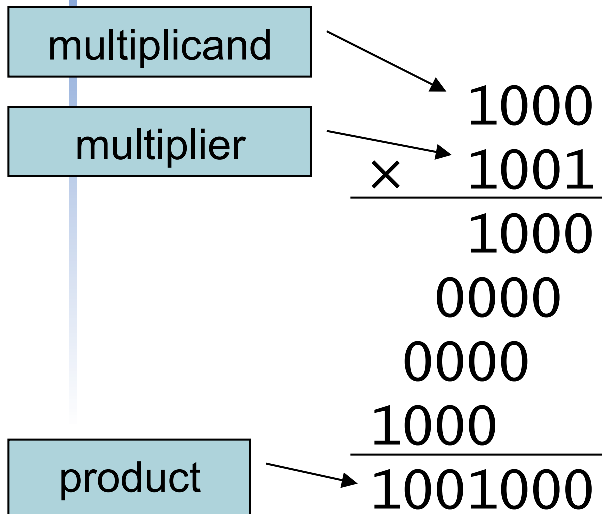
Arithmetic for Computers

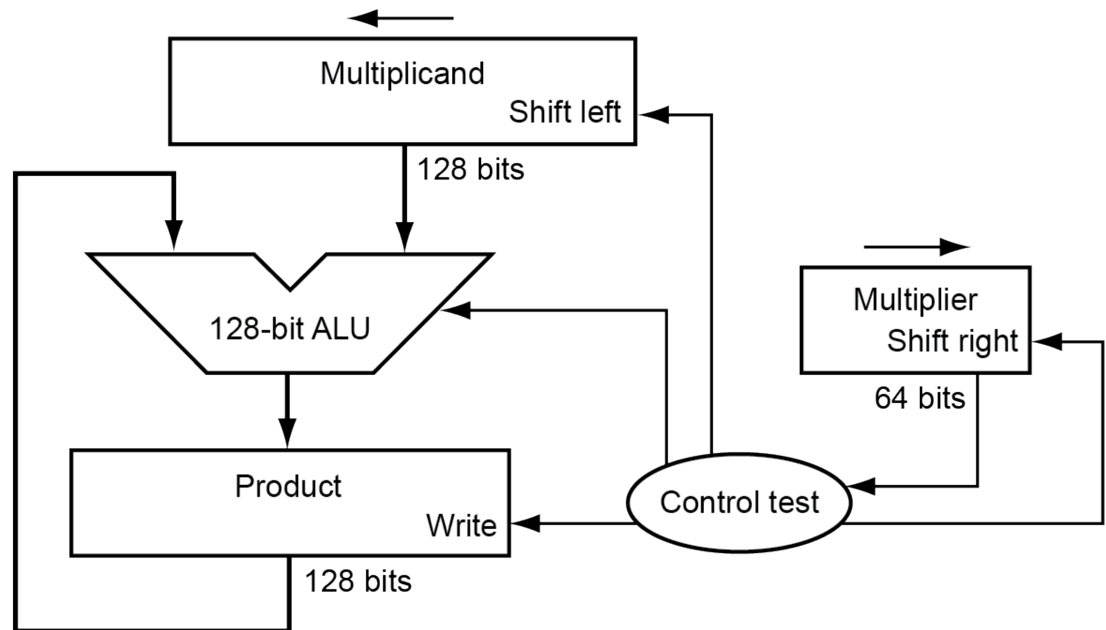
Arithmetic for Computers

- Operations on integers
  - Addition and subtraction
  - Multiplication and division
  - Dealing with overflow
- Floating-point real numbers
  - Representation and operations

Multiplication

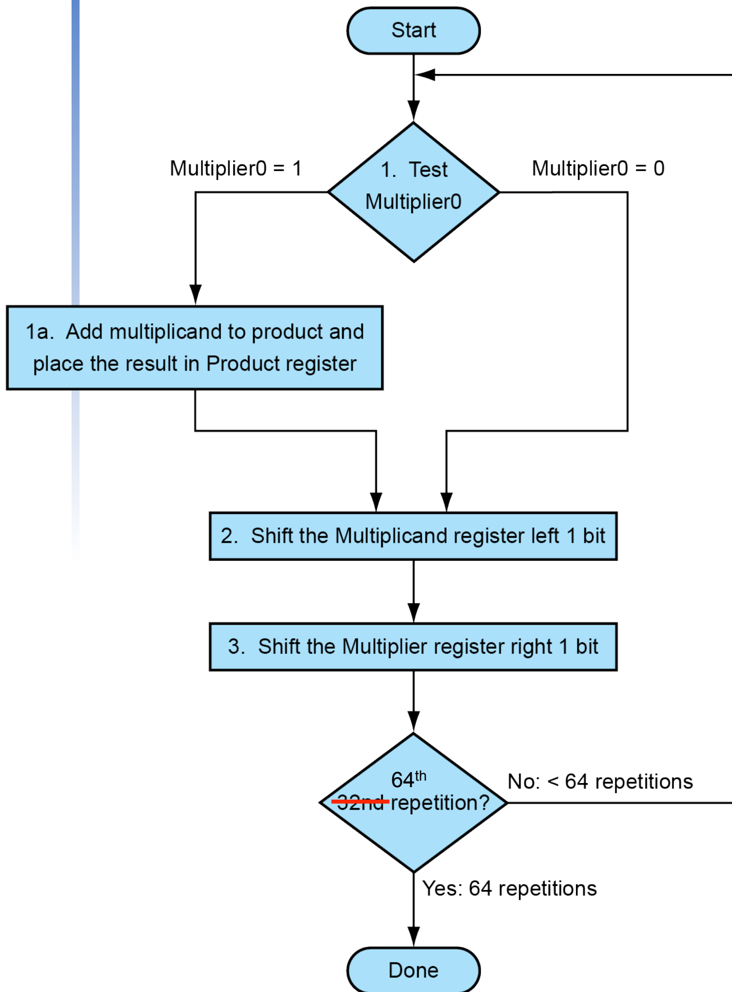
- Start with long-multiplication approach

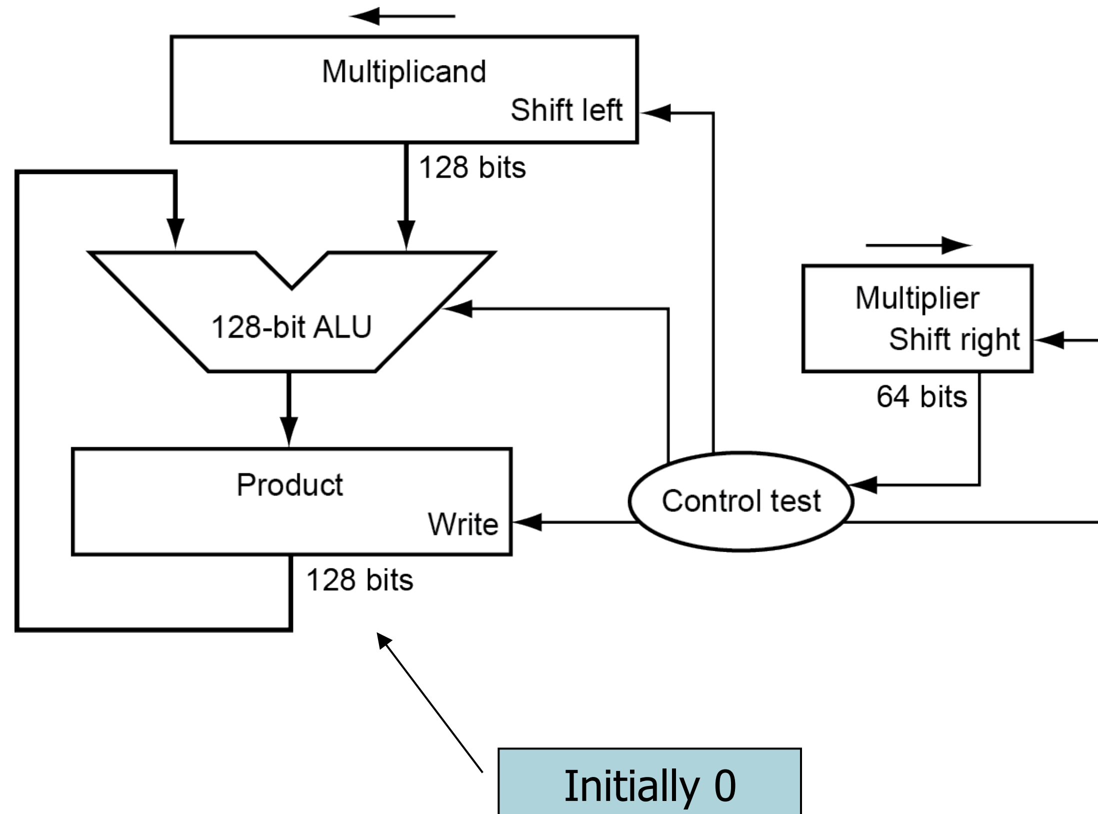




Length of product is the sum of operand lengths

Sequential Multiplication Hardware





MK MORGAN KAUFMANN

Sequential Multiplication Example

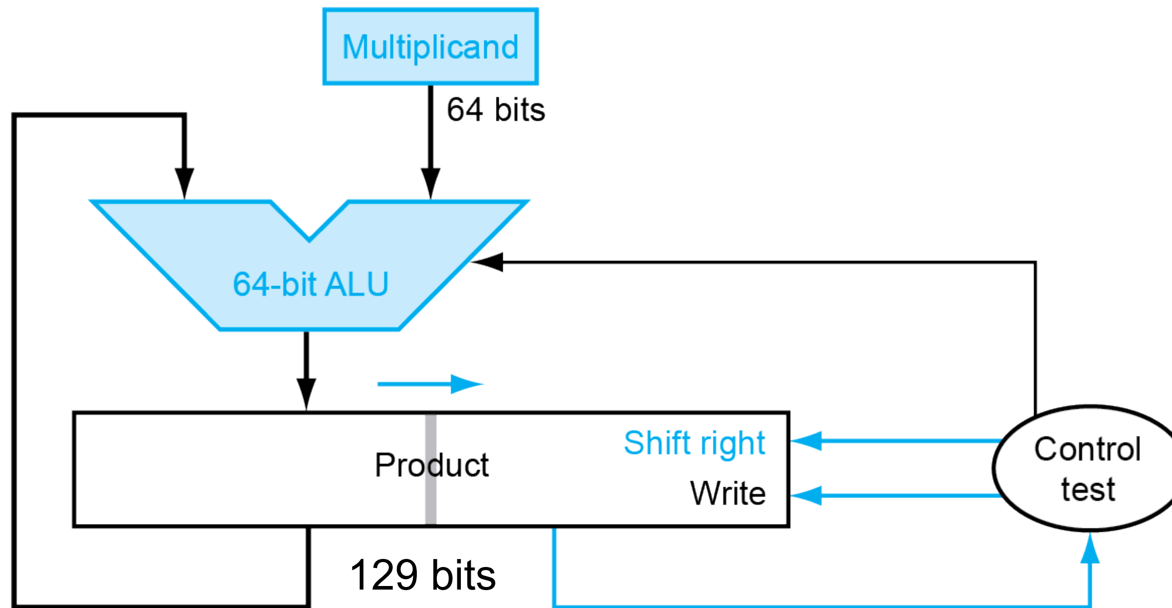
- 2_{ten} X 3_{ten}
  - 4-bit multiplication → Product Size = 8-bit
  - Multiplicand = 2_{ten}= 0000 0010_{two}
  - Multiplier = 3_{ten}= 0011_{two}

Iteration	Step	Multiplier	Multiplicand	Product
0	Initialization	0011	0000 0010	0000 0000
1	1a. Prod = Prod + Multiplicand 2.SLL Multiplicand 3.SRL Multiplier	0011 0011 0001	0000 0010 0000 0100 0000 0100	0000 0010 0000 0010 0000 0010
2	1a. Prod = Prod + Multiplicand 2.SLL Multiplicand 3.SRL Multiplier	0001 0001 0000	0000 0100 0000 1000 0000 1000	0000 0110 0000 0110 0000 0110
3	2.SLL Multiplicand 3.SRL Multiplier	0000 0000	0001 0000 0001 0000	0000 0110 0000 0110
4	2.SLL Multiplicand 3.SRL Multiplier	0000 0000	0010 0000 0010 0000	0000 0110 0000 0110

- **Worst Case Delay = 4 iteration X 3 steps = 12 cycles**
  - **For 64-bit multiplication → worst case delay = 64 X 3 = 192 cycles**

Optimized Multiplier

- Perform steps in parallel: add/shift



- One cycle per partial-product addition
  - That's ok, if frequency of multiplications is low

Optimized Multiplication Example

- 2_{ten} X 3_{ten}
  - 4-bit multiplication → Product Size = 9-bit
  - Multiplicand = 2_{ten}= 0010_{two}
  - ***No dedicated Multiplier register***

Iteration	Step	Multiplicand	Product
0	Initialization	0010	0 0000 0011
1	Prod (MS half) = Prod (MS half)+ Multiplicand ***Then*** SRL Product	0010	0 0010 0011 0 0001 0001
2	Prod (MS half) = Prod (MS half)+ Multiplicand ***Then*** SRL Product	0010	0 0011 0001 0 0001 1000
3	SRL Product	0010	0 0000 1100
4	SRL Product	0010	0 0000 0110

- **Delay = 4 iteration = 4 cycles**
  - **For 64-bit multiplication → delay = 64 cycles**
  - **That's ok, if frequency of multiplications is low**

Signed Multiplication (1)

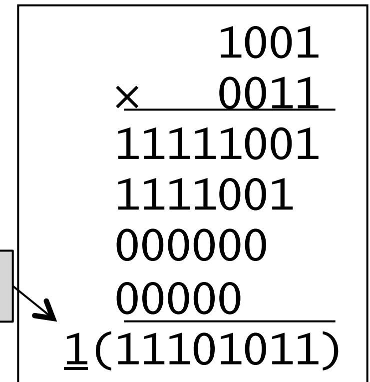
- Unsigned numbers are different from signed numbers

```
    1011      (11)ten   (-5)ten
×   1101      (13)ten   (-3)ten
----------
    1011
   0000      ------    --------
  1011       (143)ten  (15)ten
 1011   ✓               ✗
----------
 10001111
```

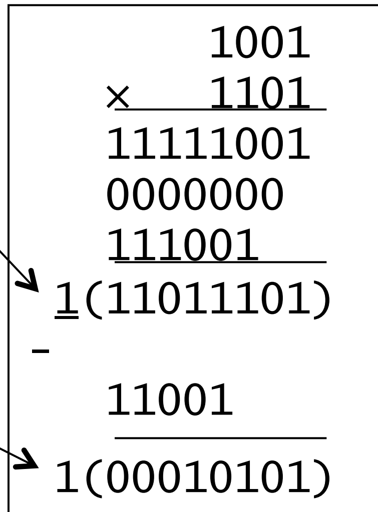
- Approach1: Convert multiplicand and multiplier to positive numbers
  - ***Negate the product if original signs are different***
  - ***Only 63 iterations are needed***

Signed Multiplication (2)

- Approach2:
  - ***Sign-Extend the partial products***
  - ***Subtract the last partial product when multiplier is negative***
- Examples:
  - $-7 \times 3 = -21_{ten} = (11101011)_{two}$
  - $7 \times -3 = -21_{ten} = (11101011)_{two}$
  - $-7 \times -3 = 21_{ten} = (00010101)_{two}$

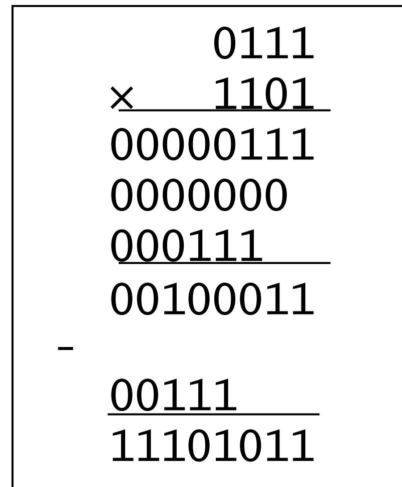


Ignore carry



Ignore carry

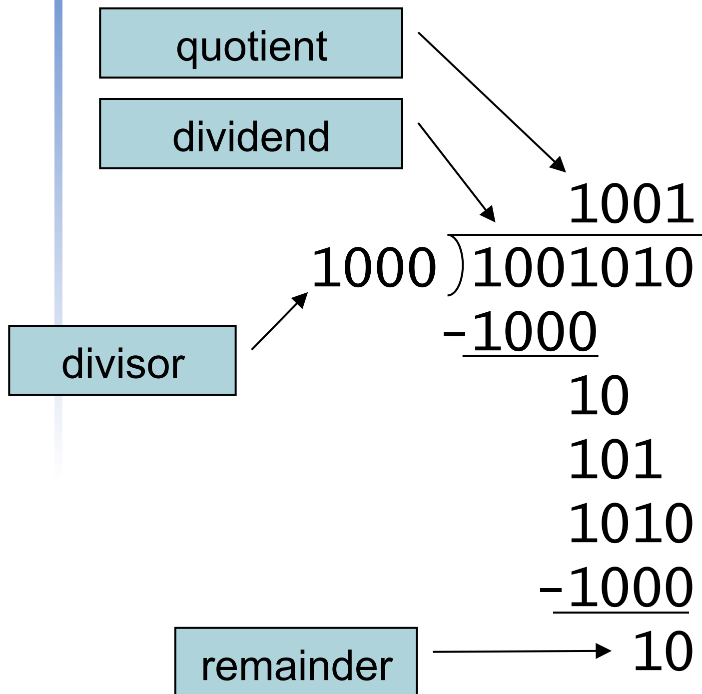
Ignore carry



RISC-V Multiplication

- Four multiply instructions:
  - mul: multiply
    - Gives the lower 64 bits of the product
  - mulh: multiply high
    - Gives the upper 64 bits of the product, assuming the operands are signed
  - mulhu: multiply high unsigned
    - Gives the upper 64 bits of the product, assuming the operands are unsigned

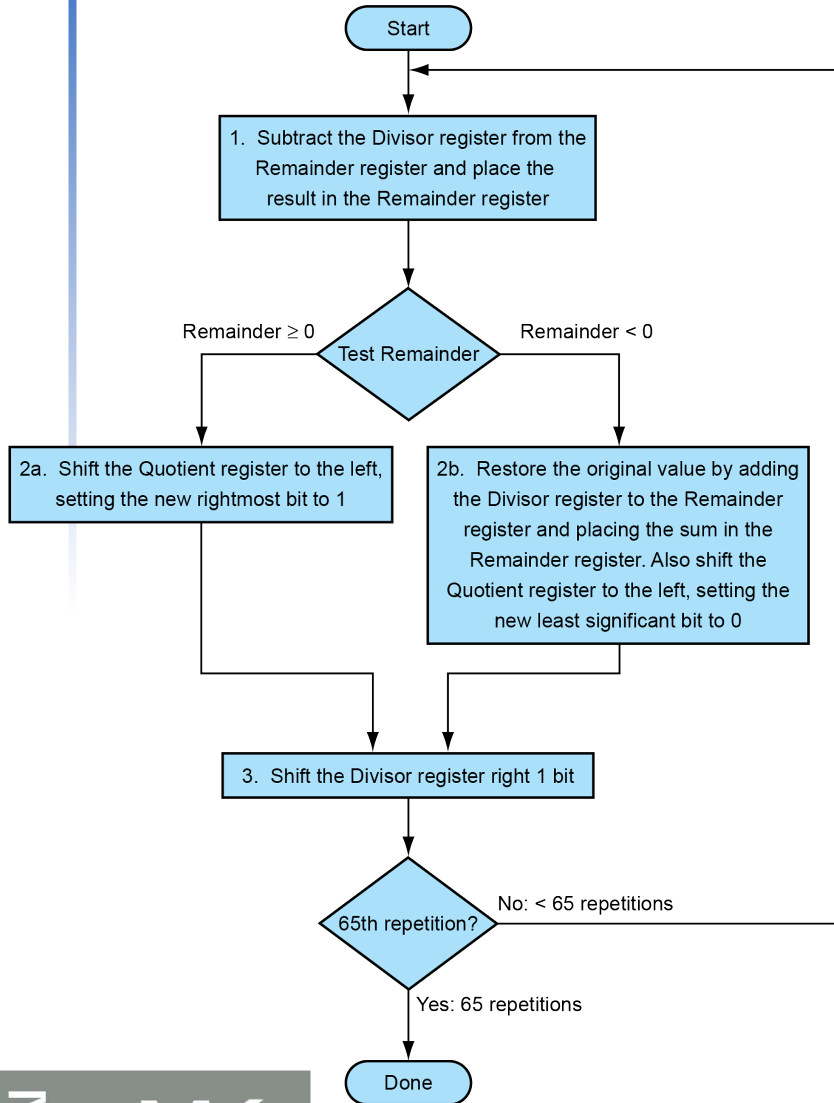
Division

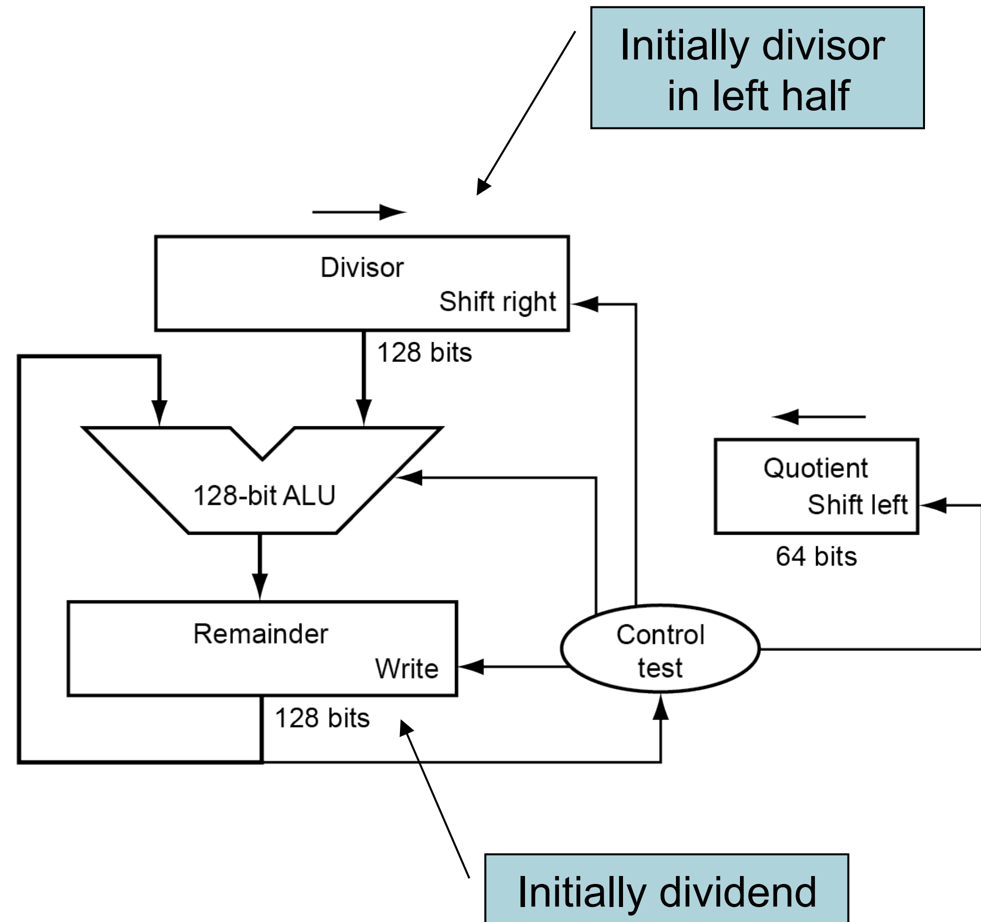


n-bit operands yield *n*-bit quotient and remainder

- Check for 0 divisor
- *Dividend = (Divisor x Quotient) + Remainder*
- Long division approach
  - If divisor ≤ dividend bits
    - 1 bit in quotient, subtract
  - Otherwise
    - 0 bit in quotient, bring down next dividend bit
- Restoring division
  - Do the subtract, and if remainder goes < 0, add divisor back
- Signed division
  - Divide using absolute values
  - Adjust sign of quotient and remainder as required

Division Hardware





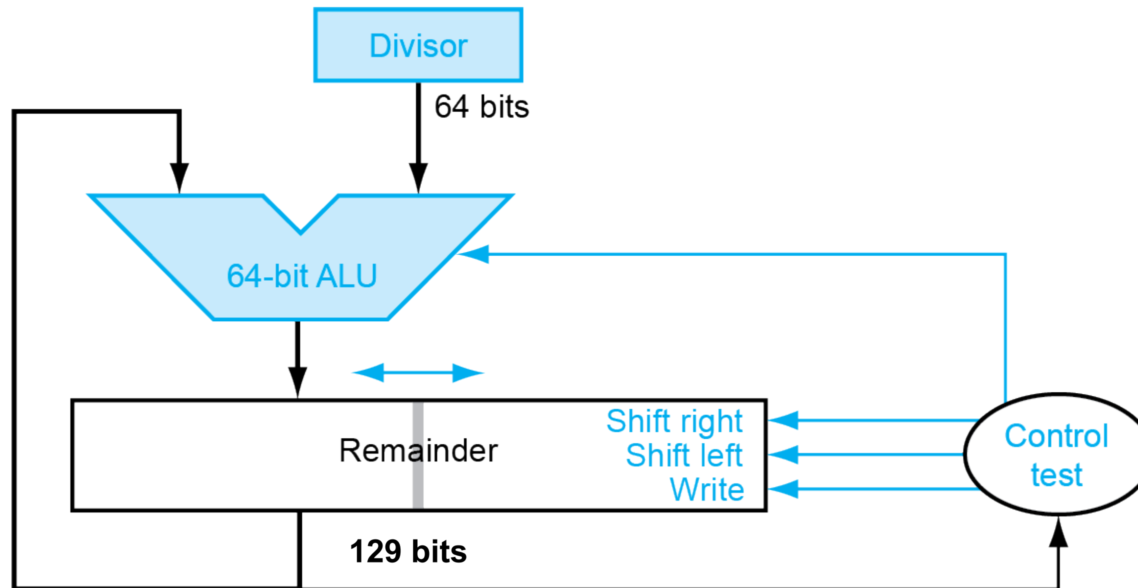
Division Example

- $7_{ten} / 2_{ten}$
  - 4-bit division
    - Quotient = 4-bit
    - Remainder = 4-bit
  - ***No dedicated Dividend register***

Iteration	Step	Quotient (Q)	Divisor	Remainder (R)
0	Initialization	0000	0010 0000	0000 0111
1	1. R = R - Divisor 2b. R = R + Divisor, SLL Q using 0 3.SRL Divisor using 0	0000 0000 0000	0010 0000 0010 0000 0001 0000	1110 0111 0000 0111 0000 0111
2	1. R = R - Divisor 2b. R = R + Divisor, SLL Q using 0 3.SRL Divisor using 0	0000 0000 0000	0001 0000 0001 0000 0000 1000	1111 0111 0000 0111 0000 0111
3	1. R = R - Divisor 2b. R = R + Divisor, SLL Q using 0 3.SRL Divisor using 0	0000 0000 0000	0000 1000 0000 1000 0000 0100	1111 1111 0000 0111 0000 0111
4	1. R = R - Divisor 2a. SLL Q using 1 3.SRL Divisor using 0	0000 0001 0001	0000 0100 0000 0100 0000 0010	0000 0011 0000 0011 0000 0011
5	1. R = R - Divisor 2a. SLL Q using 1 3.SRL Divisor using 0	0001 0011 0011	0000 0010 0000 0010 0000 0001	0000 0001 0000 0001 0000 0001

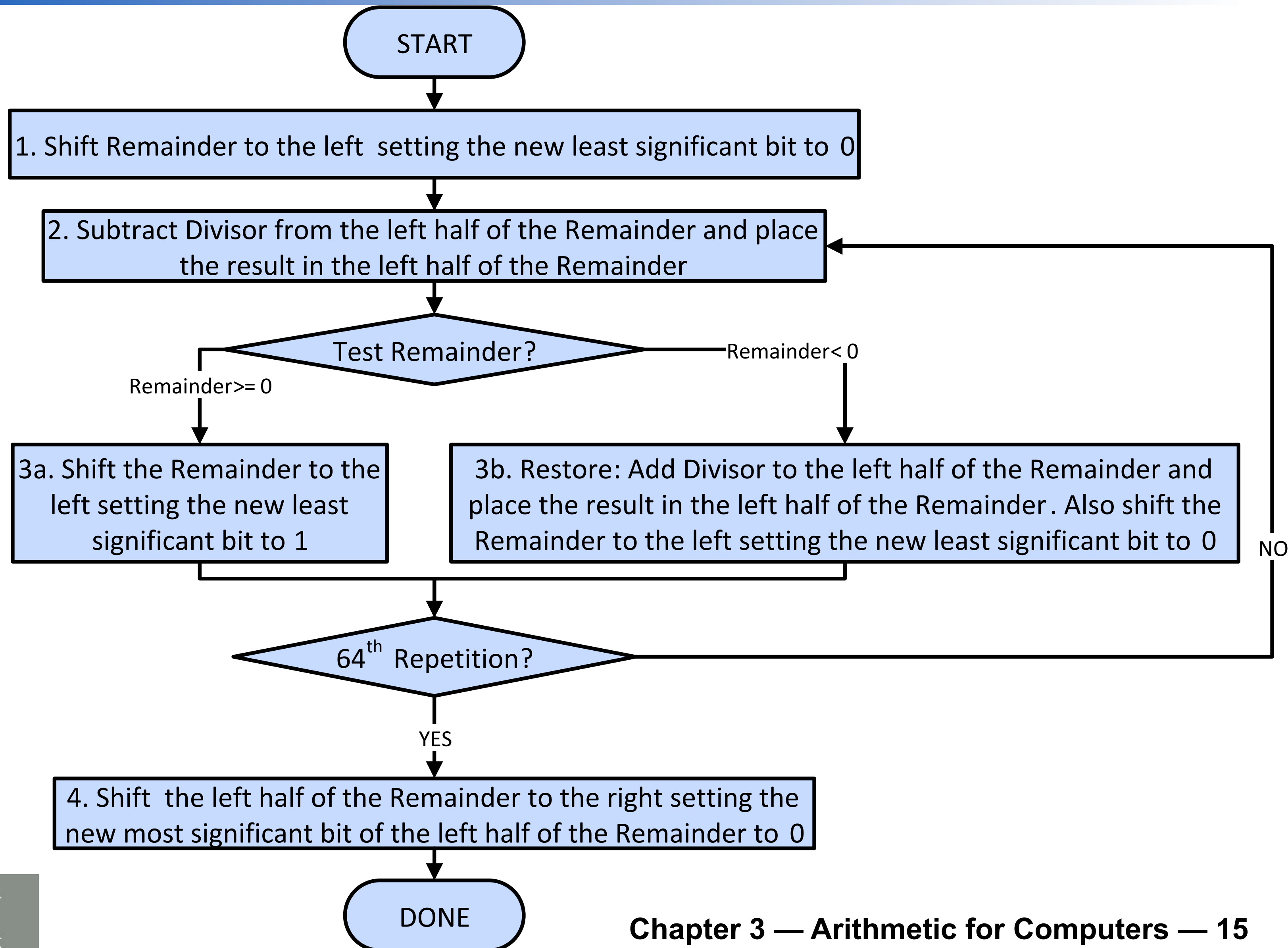
- Delay = 5 iteration X 3 steps = 15 cycles
  - **For 64-bit division → delay = 65 X 3 = 195 cycles**

Optimized Divider



- One cycle per partial-remainder subtraction
- Looks a lot like a multiplier!
  - Same hardware can be used for both

Optimized Divider Flow Chart



Optimized Divider Example

- 7_{ten} / 2_{ten}
  - 4-bit division → Quotient and Remainder are 4-bit each
  - Divisor = 2_{ten}= 0010_{two}
  - ***No dedicated Dividend or Quotient register***

Iteration	Step	Divisor	Remainder
0	Initialization ***Then*** 1. SLL Remainder using 0	0010	0 0000 0111 0 0000 1110
1	2. Left_Half _Remainder = Left_Half _Remainder - Divisor 3b. Restore Left_Half_Remainder ***Then*** SLL Remainder using 0	0010	0 0001 1100
2	2. Left_Half _Remainder = Left_Half _Remainder - Divisor 3b. Restore Left_Half_Remainder ***Then*** SLL Remainder using 0	0010	0 0011 1000
3	2. Left_Half _Remainder = Left_Half _Remainder - Divisor 3a. SLL Remainder using 1	0010	0 0011 0001
4	2. Left_Half _Remainder = Left_Half _Remainder - Divisor 3a. SLL Remainder using 1 4. SRL Left_Half_Remainder using 0	0010	0 0001 0011

- ***Quotient*** is in the least significant 4-bit of the Remainder register and the ***remainder*** is in bits 4, 5, 6, and 7 of the Remainder register
- **Delay = 4 iteration = 4 cycles → For 64-bit division: delay = 64 cycles**

Signed Division

- Convert Dividend and Divisor to positive, divide and ***then negate the Quotient*** if the ***original signs disagree***
- ***What about the Remainder sign?***
  - Use the general equation: Dividend = (Divisor X Quotient) + Remainder
  - Ex1: 7/2 → Q = 3
    - (2 X 3) + R = 7 → R = 1
  - Ex2: -7/2 → Q = -3
    - (2 X -3) + R = -7 → R = -1
  - Ex3: 7/-2 → Q = -3
    - (-2 X -3) + R = 7 → R = 1
  - Ex4: -7/-2 → Q = 3
    - (-2 X 3) + R = -7 → R = -1
- ***Observation:*** Remainder always have the same sign as the Dividend

RISC-V Division

- Four instructions:
  - div, rem: signed divide, remainder
  - divu, remu: unsigned divide, remainder

Floating Point

- Floating-point numbers = numbers with fractions
  - Called Real numbers in math
- Representation for non-integral numbers
  - Including very small and very large numbers
- ***Scientific notation: (…xxxx.yyyy….* r* ^{exponent}*)***
  - ***Normalized: Single non-zero digit to the left of the radix point***
- Like scientific notation
  - $–2.34 \times 10^{56}$ ← normalized
  - $+0.002 \times 10^{–4}$ ← not normalized
  - $+987.02 \times 10^{9}$ ← not normalized
- In binary
  - $\pm 1.xxxxxxx_2 \times 2^{yyyy}$
  - For simplicity, we show the exponent ("*yyyy*") in decimal
- Types `float` and `double` in C

Floating Point Standard

- Defined by IEEE Std 754-1985
- Developed in response to divergence of representations
  - Portability issues for scientific code
- Now almost universally adopted
- Two representations
  - Single precision (32-bit)
  - Double precision (64-bit)

IEEE Floating-Point Format

single: 8 bits
double: 11 bits

single: 23 bits
double: 52 bits

$$x = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

If we number the bits of the fraction from left to right: f_1, f_2, f_3, … then

$$x = (-1)^S \times (1 + (f_1 \times 2^{-1} + f_2 \times 2^{-2} + f_3 \times 2^{-3} + \ldots)) \times 2^{(\text{Exponent} - \text{Bias})}$$

- S: sign bit (0 ⇒ non-negative, 1 ⇒ negative)
- Normalize significand: 1.0 ≤ |significand| < 2.0
  - Always has a leading pre-binary-point 1 bit, so no need to represent it explicitly (hidden bit)
  - Significand is Fraction with the “1.” restored
- Exponent: excess representation: actual exponent + Bias
  - Ensures exponent is unsigned
  - Single: Bias = 127; Double: Bias = 1203

Single-Precision Range

- Exponents 00000000 and 11111111 reserved
- Smallest value
  - Exponent: 00000001
    $\Rightarrow$ actual exponent = 1 – 127 = –126
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- Largest value
  - exponent: 11111110
    $\Rightarrow$ actual exponent = 254 – 127 = +127
  - Fraction: 111…11 $\Rightarrow$ significand ≈ 2.0
  - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

Double-Precision Range

- Exponents 0000…00 and 1111…11 reserved
- Smallest value
  - Exponent: 00000000001
    $\Rightarrow$ actual exponent = 1 – 1023 = –1022
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
- Largest value
  - Exponent: 11111111110
    $\Rightarrow$ actual exponent = 2046 – 1023 = +1023
  - Fraction: 111…11 $\Rightarrow$ significand ≈ 2.0
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

Floating-Point Example

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

Floating-Point Example

- What number is represented by the single-precision float

  11000000101000…00

  - S = 1
  - Fraction = $01000\ldots00_2$
  - Fxponent = $10000001_2 = 129$
- $x = (-1)^1 \times (1 + 01_2) \times 2^{(129 - 127)}$

  $= (-1) \times 1.25 \times 2^2$

  $= -5.0$

MK MORGAN KAUFMANN

IEEE 754 Features

- ***Overflow*** in FP means that exponent is too large to fit in its field
  - Single-precision: actual exponent > 127
  - Double-precision: actual exponent > 1023
- ***Underflow*** in FP means that exponent is too small to fit in its field
  - Single-precision: actual exponent < -126
  - Double-precision: actual exponent < -1022
- IEEE 754 standard simplifies FP comparison:
  - First, compare the sign-bit
  - Second, compare the biased exponent → unsigned comparison
  - Finally, compare the fraction

Single-Precision		**Double-Precision**		**Object Represented**
Exponent	**Fraction**	**Exponent**	**Fraction**	
0	0	0	0	Zero
0	Non-Zero	0	Non-Zero	± Denormalized Number
255	0	2047	0	± ∞
255	Non-Zero	2047	Non-Zero	Not a Number (NAN)
1-254	Anything	1-2046	Anything	± Normalized Number

Floating-Point Addition

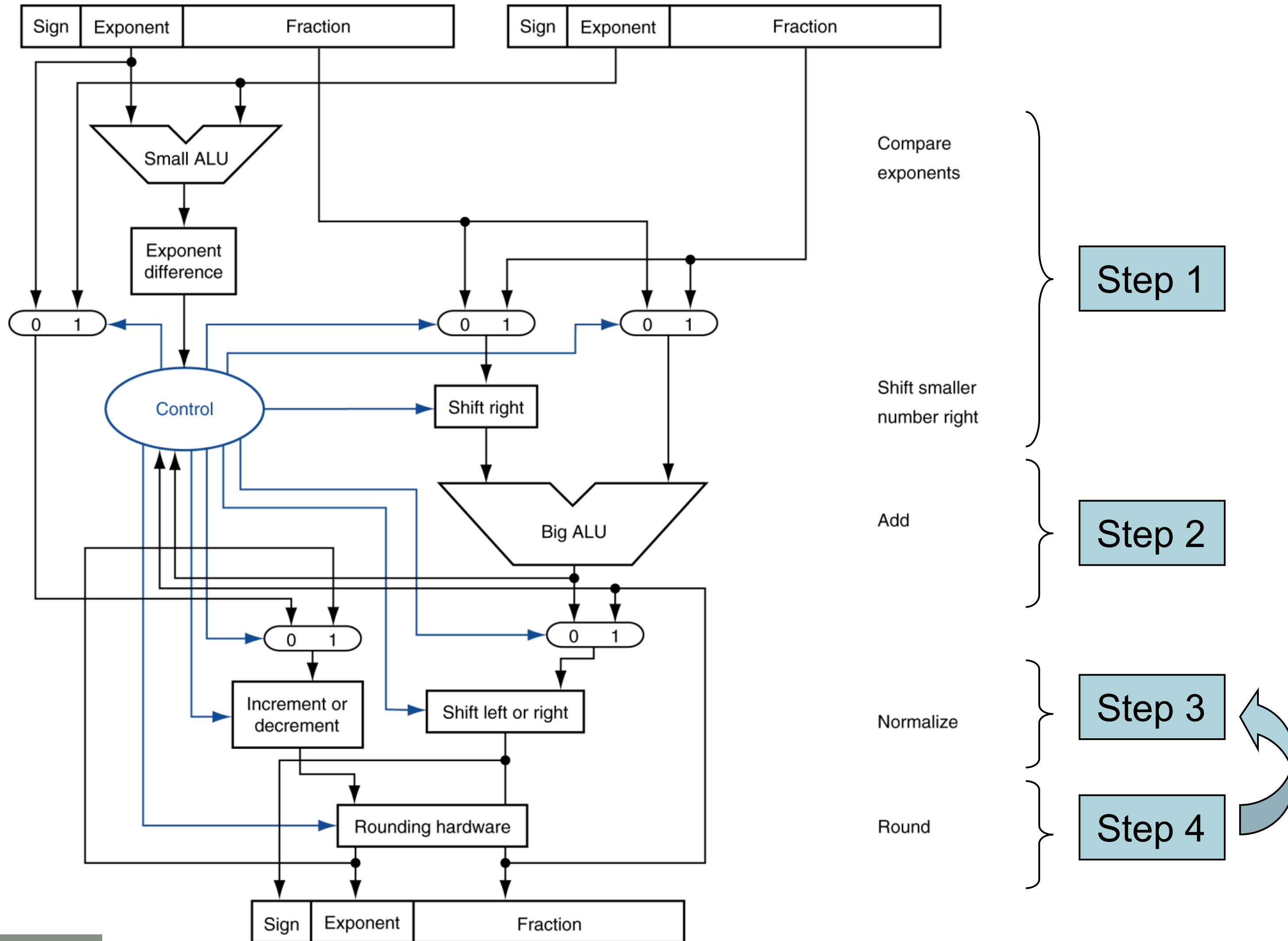
- Now consider a 4-digit binary example
  - $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$ (0.5 + –0.4375)
- 1. Align binary points
  - Shift number with smaller exponent
  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$
- 2. Add significands
  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1} = 0.001_2 \times 2^{-1}$
- 3. Normalize result & check for over/underflow
  - $1.000_2 \times 2^{-4}$, with no over/underflow
- 4. Round and renormalize if necessary
  - $1.000_2 \times 2^{-4}$ (no change) = 0.0625

MK MORGAN KAUFMANN

FP Adder Hardware

- Much more complex than integer adder
- Doing it in one clock cycle would take too long
  - Much longer than integer operations
  - Slower clock would penalize all instructions
- FP adder usually takes several cycles
  - Can be pipelined

FP Adder Hardware



Floating-Point Multiplication

- Now consider a 4-digit binary example
  - $1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}$ (0.5×-0.4375)
- 1. Add exponents
  - Unbiased: $-1 + -2 = -3$
  - Biased: $(-1 + 127) + (-2 + 127) = -3 + 254 - 127 = -3 + 127$
- 2. Multiply significands
  - $1.000_2 \times 1.110_2 = 1.110_2 \Rightarrow 1.110_2 \times 2^{-3}$
- 3. Normalize result & check for over/underflow
  - $1.110_2 \times 2^{-3}$ (no change) with no over/underflow
- 4. Round and renormalize if necessary
  - $1.110_2 \times 2^{-3}$ (no change)
- 5. Determine sign: +ve × –ve ⇒ –ve
  - $-1.110_2 \times 2^{-3} = -0.21875$

FP Arithmetic Hardware

- FP multiplier is of similar complexity to FP adder
  - But uses a multiplier for significands instead of an adder
- FP arithmetic hardware usually does
  - Addition, subtraction, multiplication, division, reciprocal, square-root
  - FP ↔ integer conversion
- Operations usually takes several cycles
  - Can be pipelined

FP Instructions in RISC-V

- Separate FP registers: f0, …, f31
  - double-precision
  - single-precision values stored in the lower 32 bits
- FP instructions operate only on FP registers
  - Programs generally don't do integer ops on FP data, or vice versa
  - More registers with minimal code-size impact
- FP load and store instructions
  - `flw, fld`
  - `fsw, fsd`
  - The base registers for floating-point data transfers which are used for addresses remain integer registers

MK MORGAN KAUFMANN

FP Instructions in RISC-V

- Single-precision arithmetic
  - `fadd.s, fsub.s, fmul.s, fdiv.s, fsqrt.s`
    - e.g., `fadds.s f2, f4, f6`
- Double-precision arithmetic
  - `fadd.d, fsub.d, fmul.d, fdiv.d, fsqrt.d`
    - e.g., `fadd.d f2, f4, f6`
- Single- and double-precision comparison
  - `feq.s, flt.s, fle.s`
  - `feq.d, flt.d, fle.d`
  - Result is 0 (comparison is false) or 1 (comparison is true) in integer destination register
    - Use beq, bne to branch on comparison result

MORGAN KAUFMANN

FP Example: °F to °C

- C code:

```
float f2c (float fahr) {
  return ((5.0/9.0)*(fahr - 32.0));
}
```

  - fahr in f10, result in f10, literals in global memory space

- Compiled RISC-V code:

```
f2c:
  flw    f0,const5(x3)  // f0 = 5.0f
  flw    f1,const9(x3)  // f1 = 9.0f
  fdiv.s f0, f0, f1  // f0 = 5.0f / 9.0f
  flw    f1,const32(x3) // f1 = 32.0f
  fsub.s f10,f10,f1  // f10 = fahr – 32.0
  fmul.s f10,f0,f10  // f10 = (5.0f/9.0f) * (fahr–32.0f)
  jalr   x0,0(x1)    // return
```

Accurate Arithmetic

- IEEE Std 754 specifies additional rounding control
  - Extra bits of precision (guard, round, sticky)
  - Choice of rounding modes (4 modes):
    - Always round-up (towards +∞)
    - Always round-down (towards -∞)
    - Truncate
    - Round to the nearest
      - ***What to do when the number is exactly halfway in between?***
        - **Round to the nearest, ties to even: If the least significant bit retained is odd then add one, otherwise truncate (Default Mode)**
        - **Round to the nearest, ties away from zero: It is rounded to the nearest value above (for positive numbers) or below (for negative numbers)**
- Allows programmer to fine-tune numerical behavior of a computation
- Not all FP units implement all options
- Trade-off between hardware complexity, performance, and market requirements

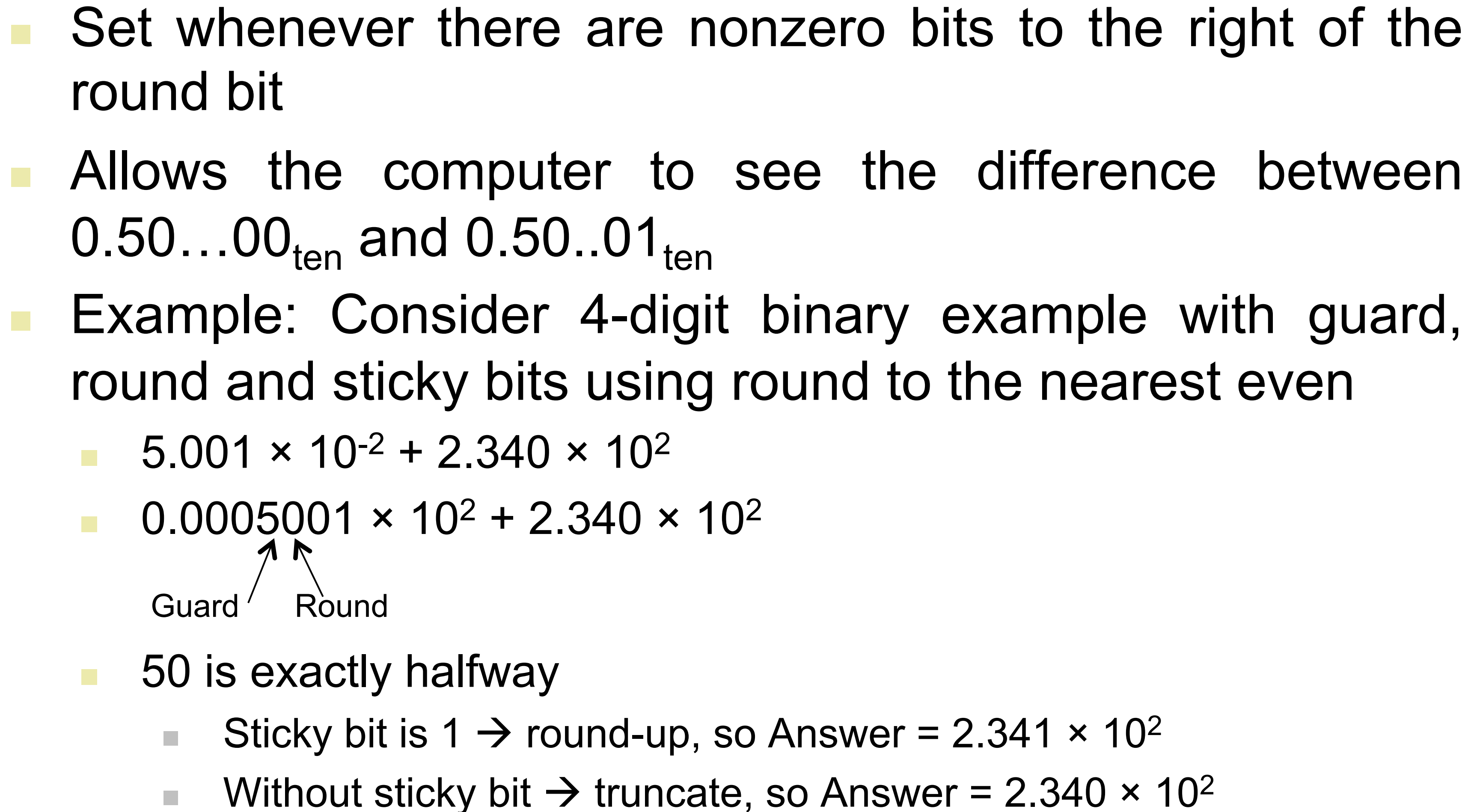
Rounding Example with Guard and Round bits

- Consider a 3-digit binary example using round to the nearest
  - $2.56 \times 10^1 + 2.34 \times 10^3$
- ***Solution with guard and round*** (two extra bits on the right during intermediate additions):
  - $0.0256 \times 10^3 + 2.34 \times 10^3 = 2.3656 \times 10^3$

Guard Round

  - 0-49: round-down, 51-99: round-up
  - So, answer is 2.37×10^3
- ***Solution without guard and round*** (truncate during intermediate additions):
  - $0.02 \times 10^3 + 2.34 \times 10^3 = 2.36 \times 10^3$

Sticky Bit



- Set whenever there are nonzero bits to the right of the round bit
- Allows the computer to see the difference between $0.50\ldots00_{ten}$ and $0.50..01_{ten}$
- Example: Consider 4-digit binary example with guard, round and sticky bits using round to the nearest even
  - $5.001 \times 10^{-2} + 2.340 \times 10^{2}$
  - $0.0005001 \times 10^{2} + 2.340 \times 10^{2}$

  - 50 is exactly halfway
    - Sticky bit is 1 → round-up, so Answer = 2.341×10^{2}
    - Without sticky bit → truncate, so Answer = 2.340×10^{2}

Concluding Remarks

- Bits have no inherent meaning
  - Interpretation depends on the instructions applied
- Computer representations of numbers
  - Finite range and precision
  - Need to account for this in programs

Concluding Remarks

- ISAs support arithmetic
  - Signed and unsigned integers
  - Floating-point approximation to reals
- Bounded range and precision
  - Operations can overflow and underflow

Chapter 4

The Processor

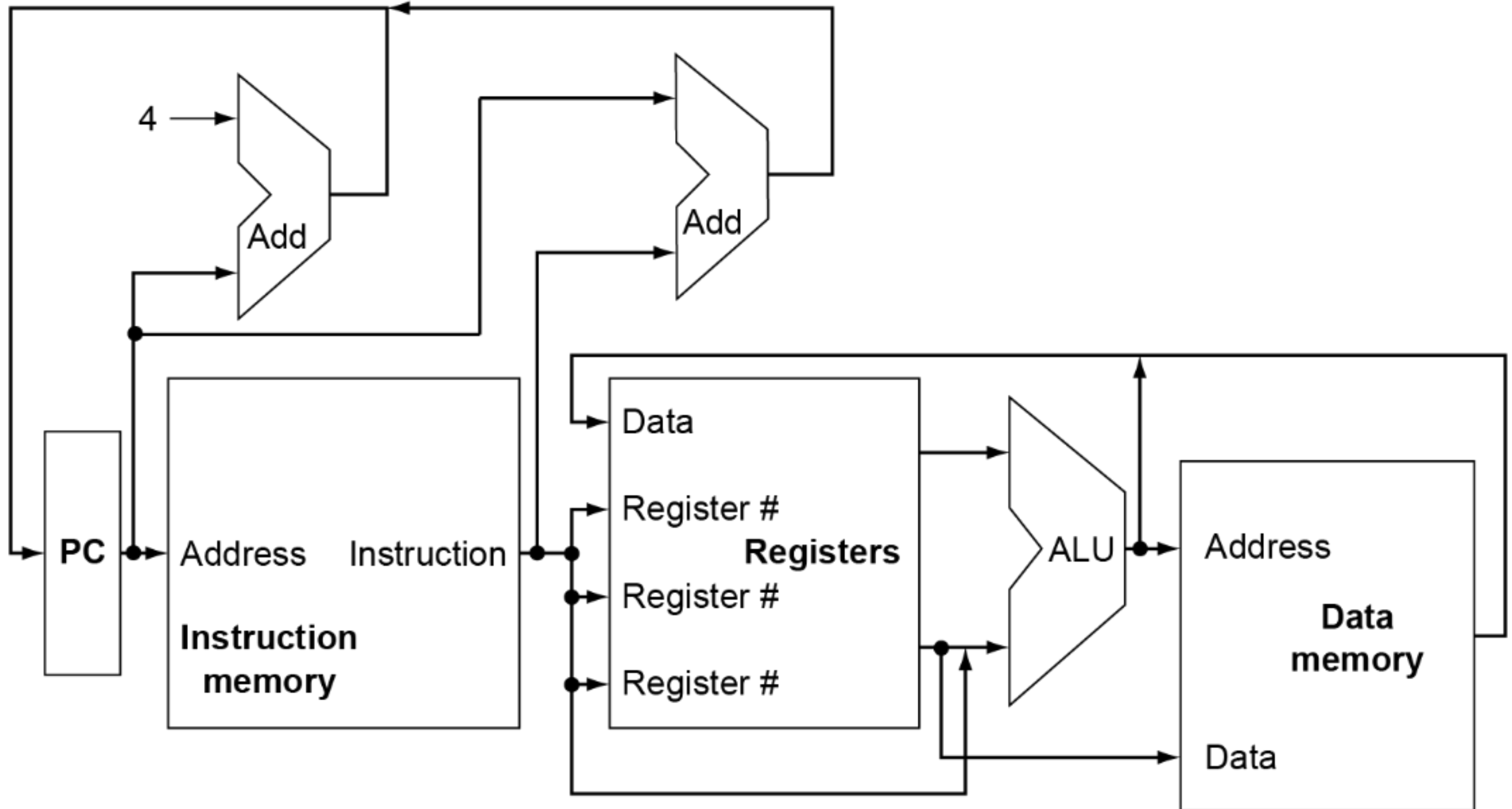
Introduction

- CPU performance factors
  - Instruction count
    - Determined by ISA and compiler
  - CPI and Cycle time
    - Determined by CPU hardware
- We will examine two RISC-V implementations
  - A simplified version (i.e. Single-Cycle implementation)
  - Multi-Cycle implementation
  - A more realistic pipelined version
- Simple subset, shows most aspects
  - Memory reference: `ld`, `sd`
  - Arithmetic/logical: `add`, `sub`, `and`, `or`
  - Control transfer: `beq`

Instruction Execution

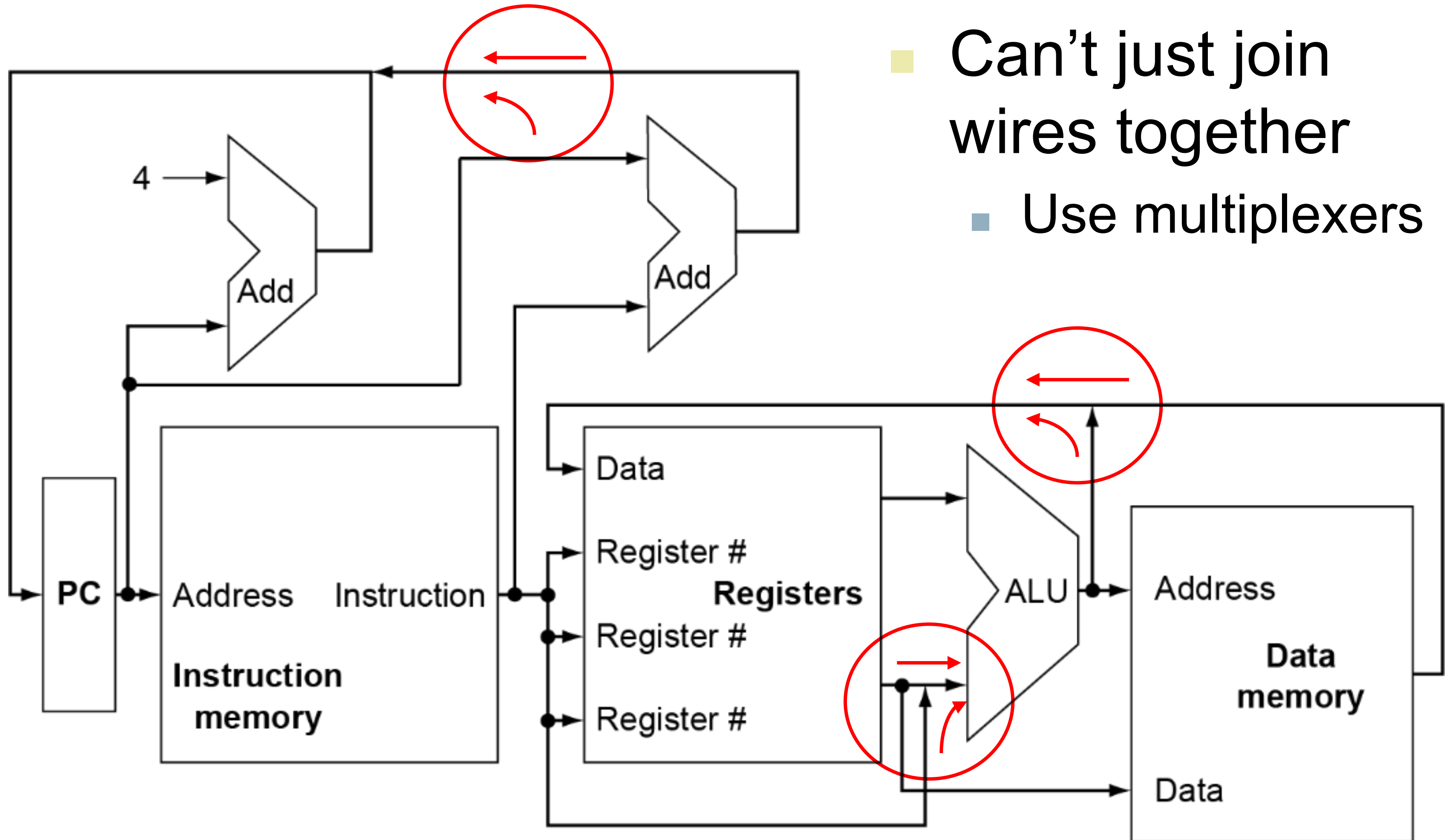
- PC → instruction memory, fetch instruction
- Register numbers → register file, read registers
- Depending on instruction class
  - Use ALU to calculate
    - Arithmetic result
    - Memory address for load/store
    - Branch comparison
  - Access data memory for load/store
  - PC ← target address or PC + 4

CPU Overview



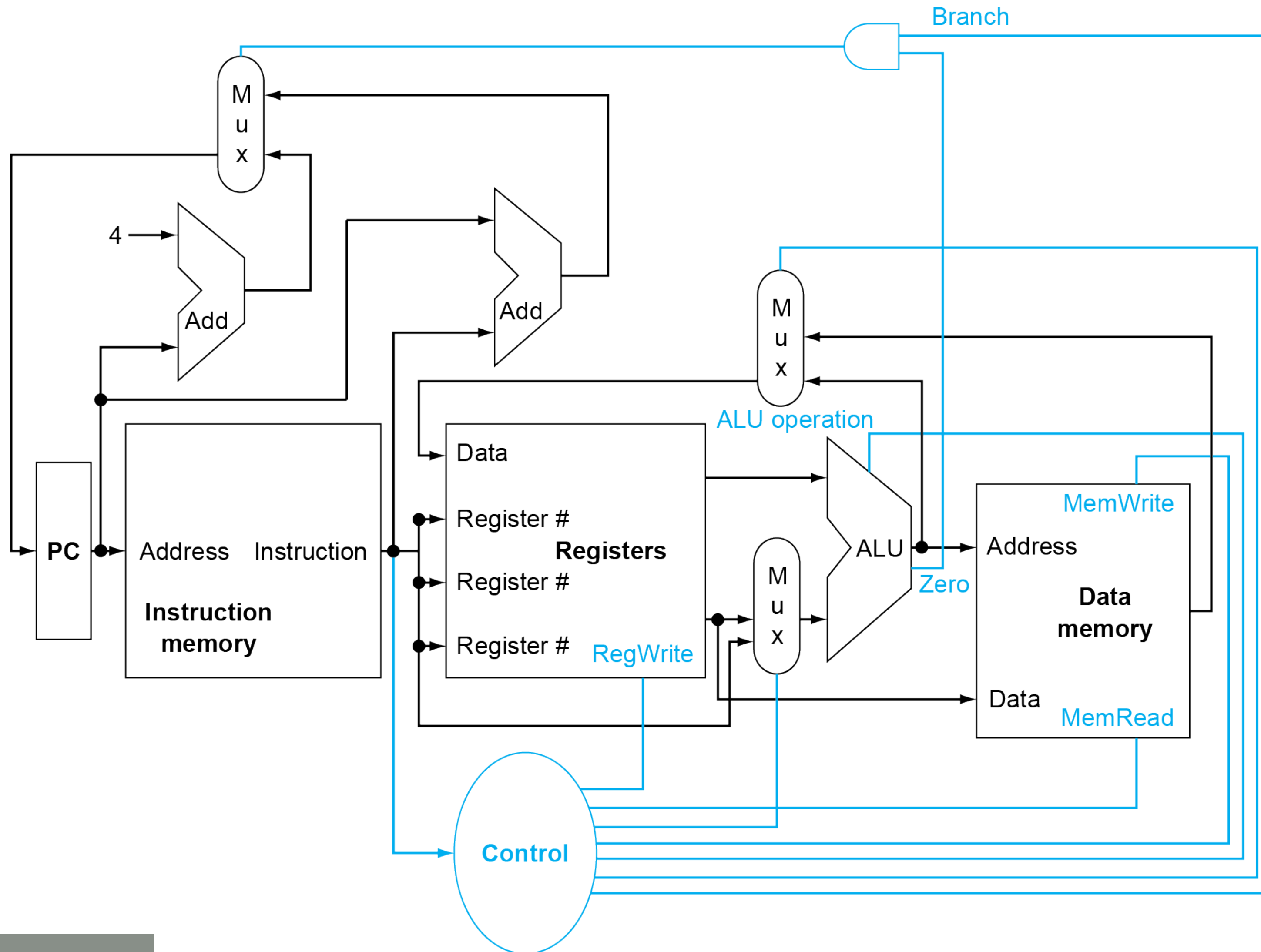
MORGAN KAUFMANN

Multiplexers



- Can’t just join wires together
  - Use multiplexers

Control

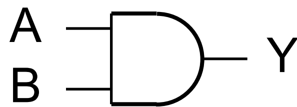


Logic Design Basics

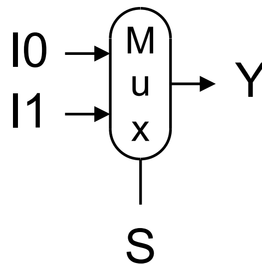
- Information encoded in binary
  - Low voltage = 0, High voltage = 1
  - One wire per bit
  - Multi-bit data encoded on multi-wire buses
- Combinational element
  - Operate on data
  - Output is a function of input
- State (sequential) elements
  - Store information

Combinational Elements

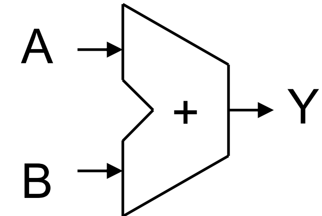
- AND-gate
  - Y = A & B



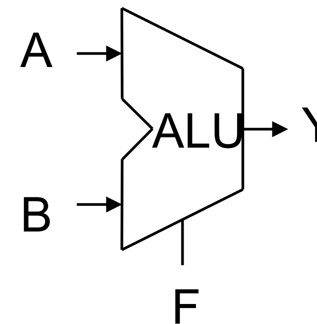
- Multiplexer
  - Y = S ? I1 : I0



- Adder
  - Y = A + B

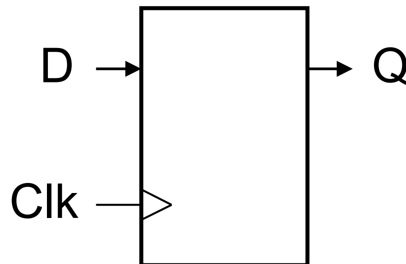


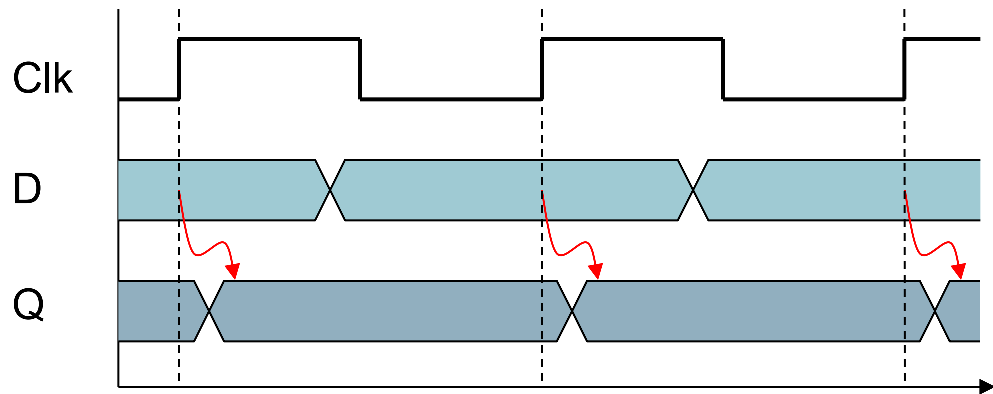
- Arithmetic/Logic Unit
  - Y = F(A, B)



Sequential Elements

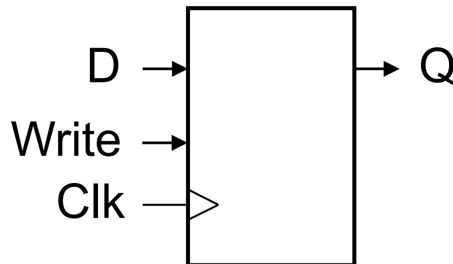
- Register: stores data in a circuit
  - Uses a clock signal to determine when to update the stored value
  - Edge-triggered: update when Clk changes from 0 to 1

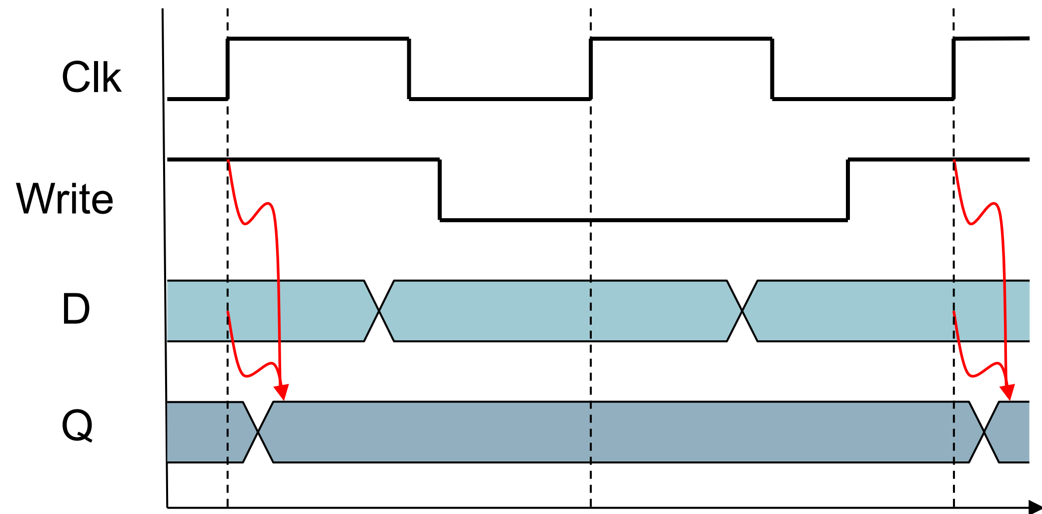




Sequential Elements

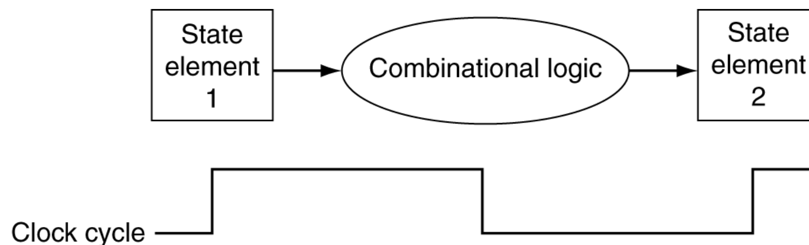
- Register with write control
  - Only updates on clock edge when write control input is 1
  - Used when stored value is required later

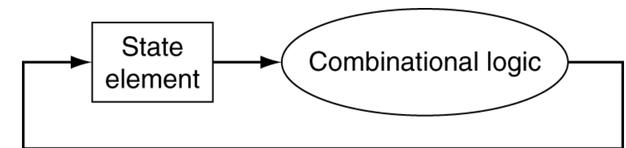




Clocking Methodology

- ***Clocking methodology*** defines when signals can be read or written
  - This is important because if a signal is to be read and written simultaneously, then the value read could correspond to the old value, new value, or a mix
- Most CPUs use edge-triggered clocking methodology
  - This methodology allows us to read the contents of a state element at the beginning of the clock cycle, send the value through Combinational logic during the clock cycle, then write the output to the same state element or another state element at the end of the clock cycle
  - Longest Combinational logic delay determines clock period

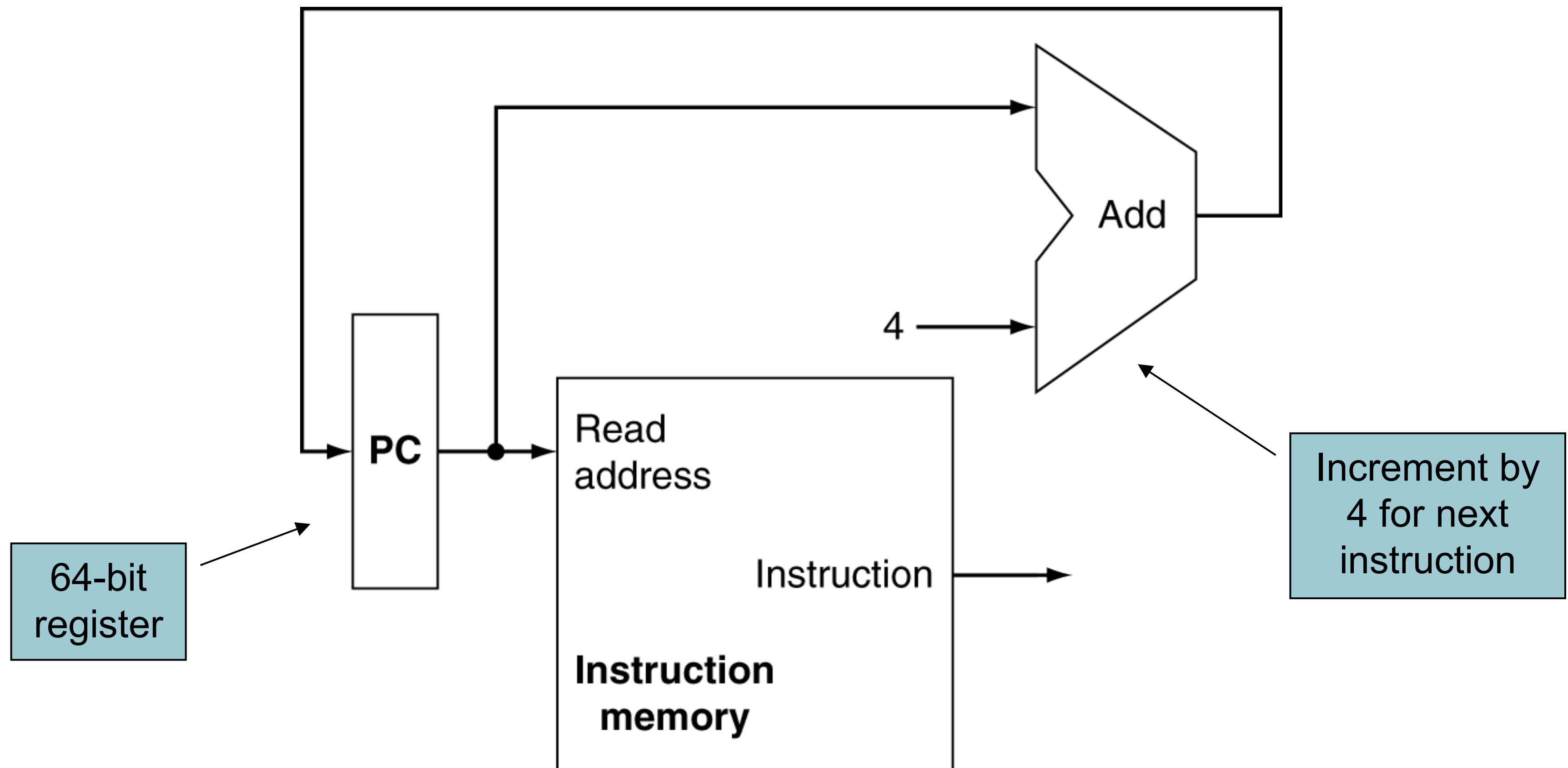




Building Datapath for Single-Cycle CPU

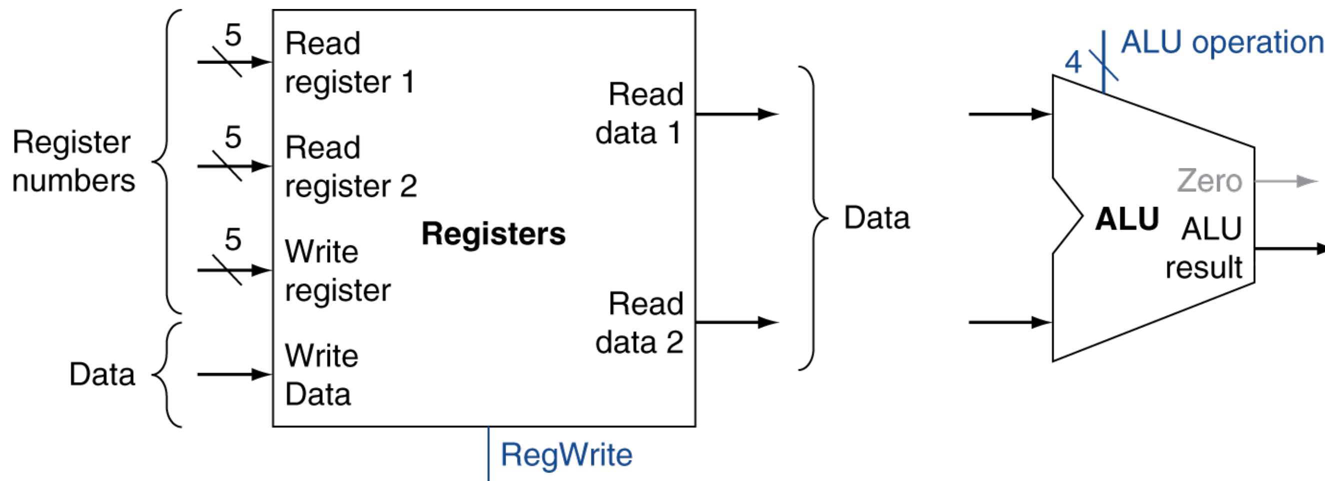
- Every instruction is executed in one cycle
  - No datapath resource can be used more than once per instruction → any element needed more than once must be duplicated
- Datapath
  - Elements that process data and addresses in the CPU
    - Registers, ALUs, mux's, memories, …
- We will build a RISC-V datapath incrementally
  - Refining the overview design

Instruction Fetch

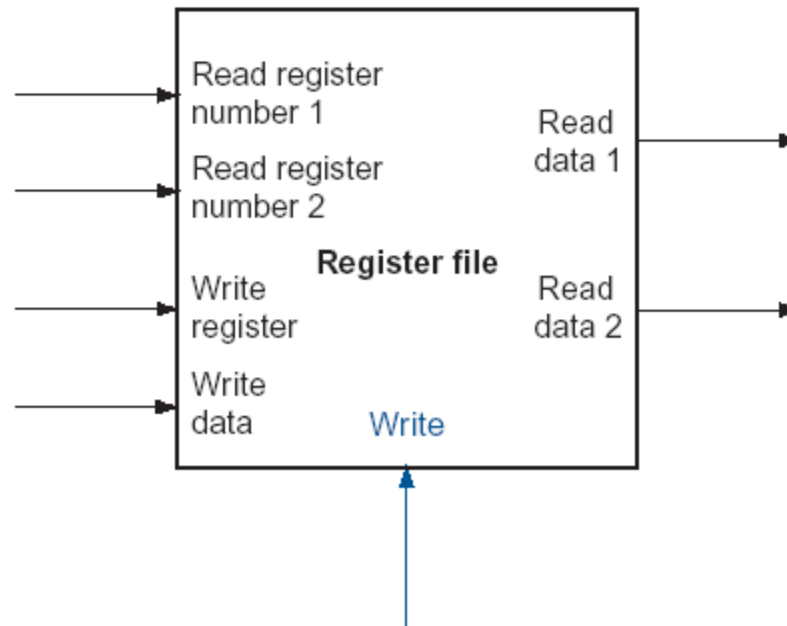


R-Format Instructions

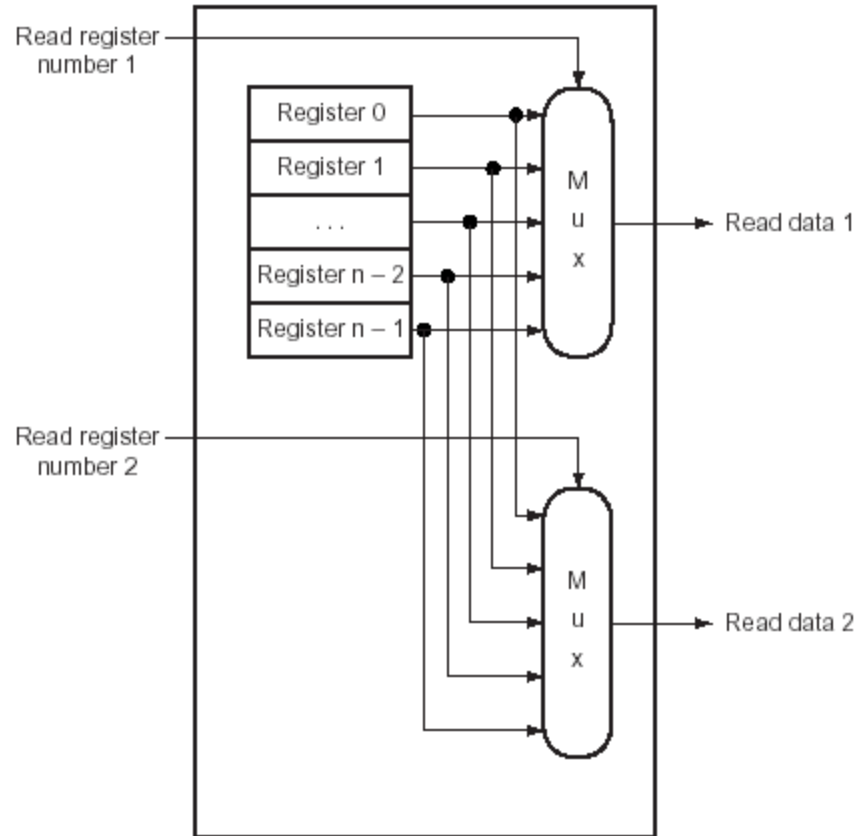
- Read two register operands
- Perform arithmetic/logical operation
- Write register result



Register File

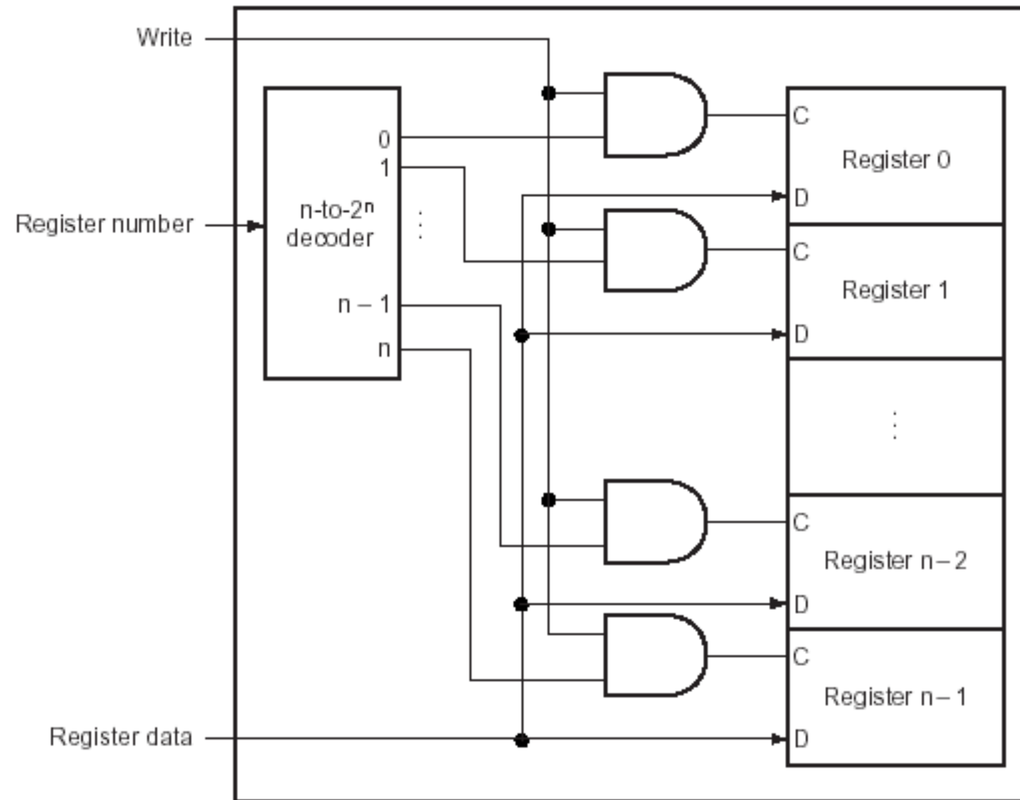


Register File – Read Ports



- Use two 64-bit 32-to-1 multiplexers whose control lines are the register numbers.

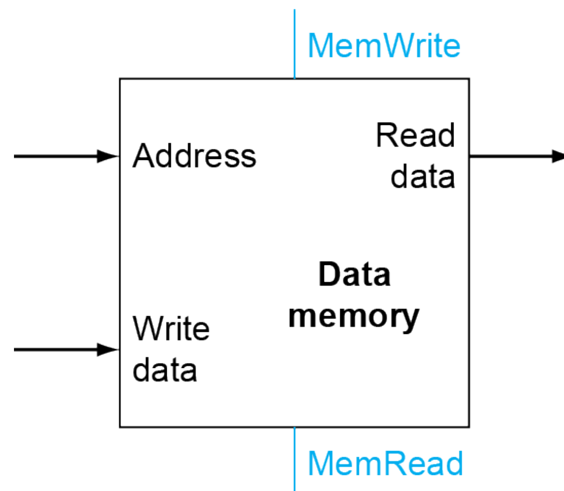
Register File – Write Port



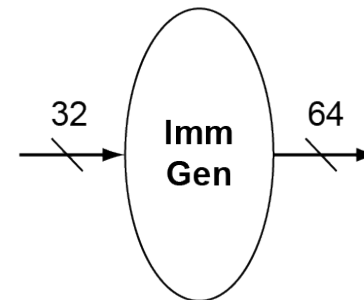
- We need 5-to-32 decoder in addition to the Write signal to generate actual write signal
- The register data is common to all registers

Load/Store Instructions

- Read register operands
- Calculate address using 12-bit offset
  - Use ALU, but sign-extend offset
- Load: Read memory and update register
- Store: Write register value to memory



a. Data memory unit

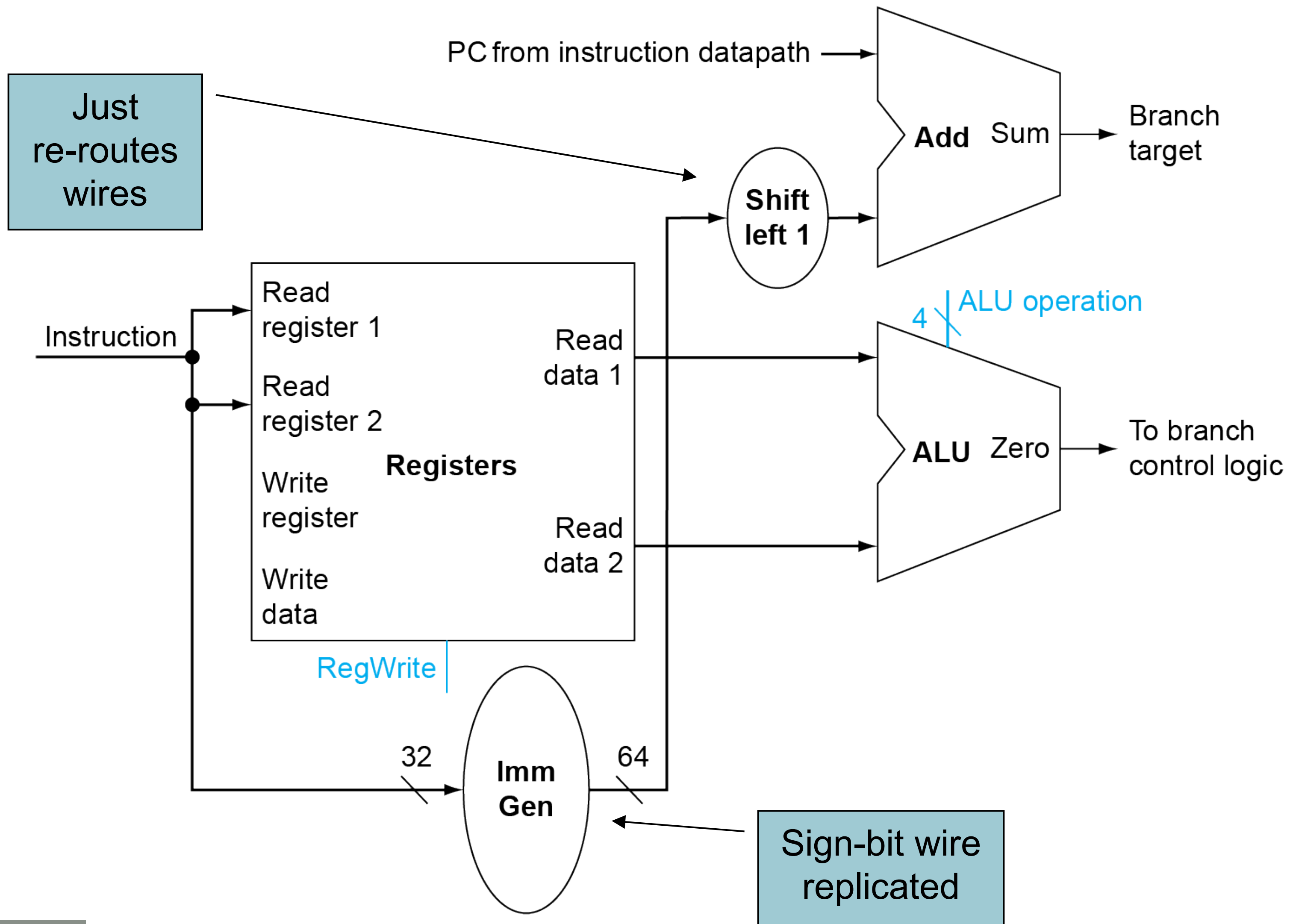


b. Immediate generation unit

Branch Instructions

- Read register operands
- Compare operands
  - Use ALU, subtract and check Zero output
  - Taken Branch: Condition is true → Jump to branch target
  - Not-Taken Branch: Condition is false → Execute instruction next to branch
- Calculate target address
  - Sign-extend displacement
  - Shift left 1 place (halfword displacement)
  - Add to PC value

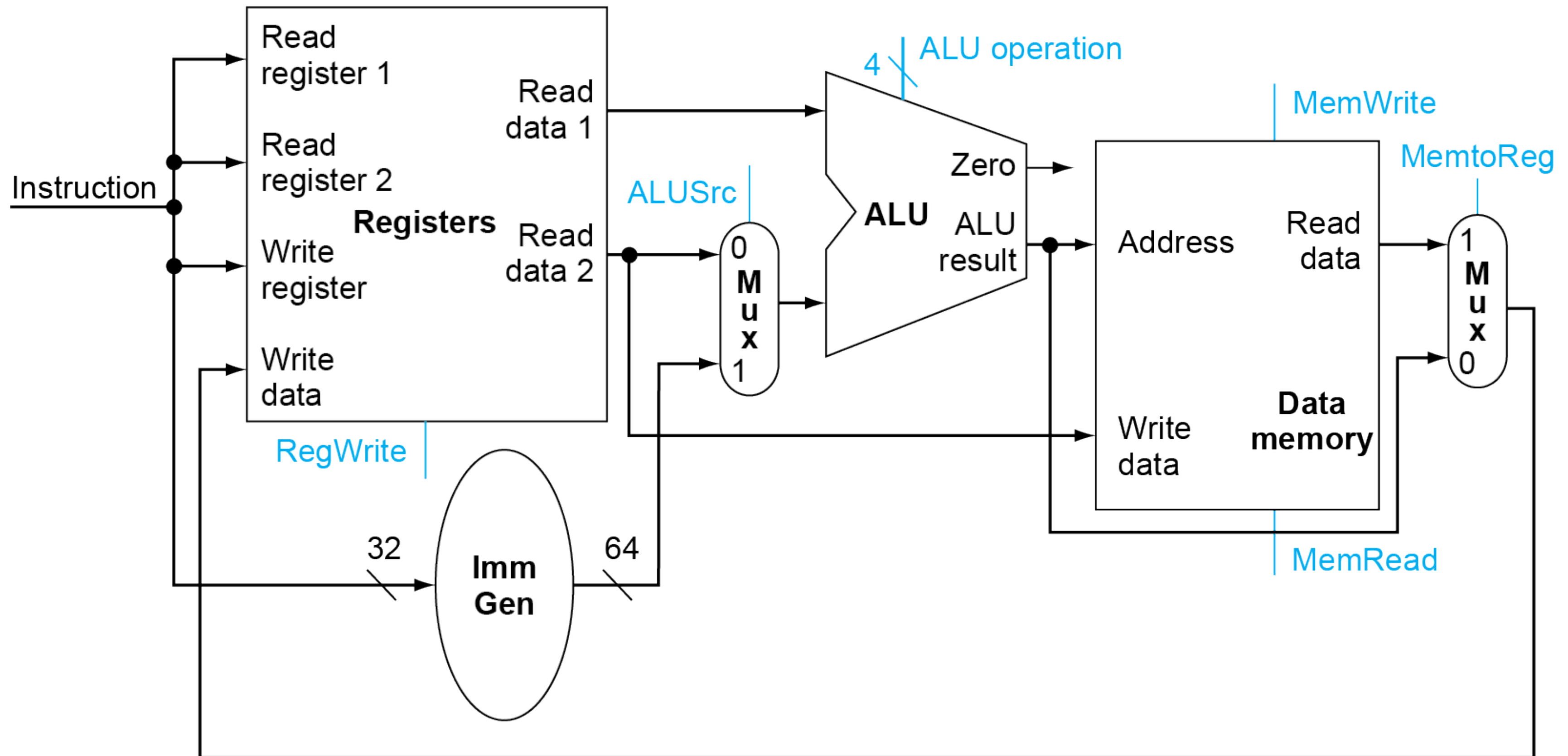
Branch Instructions



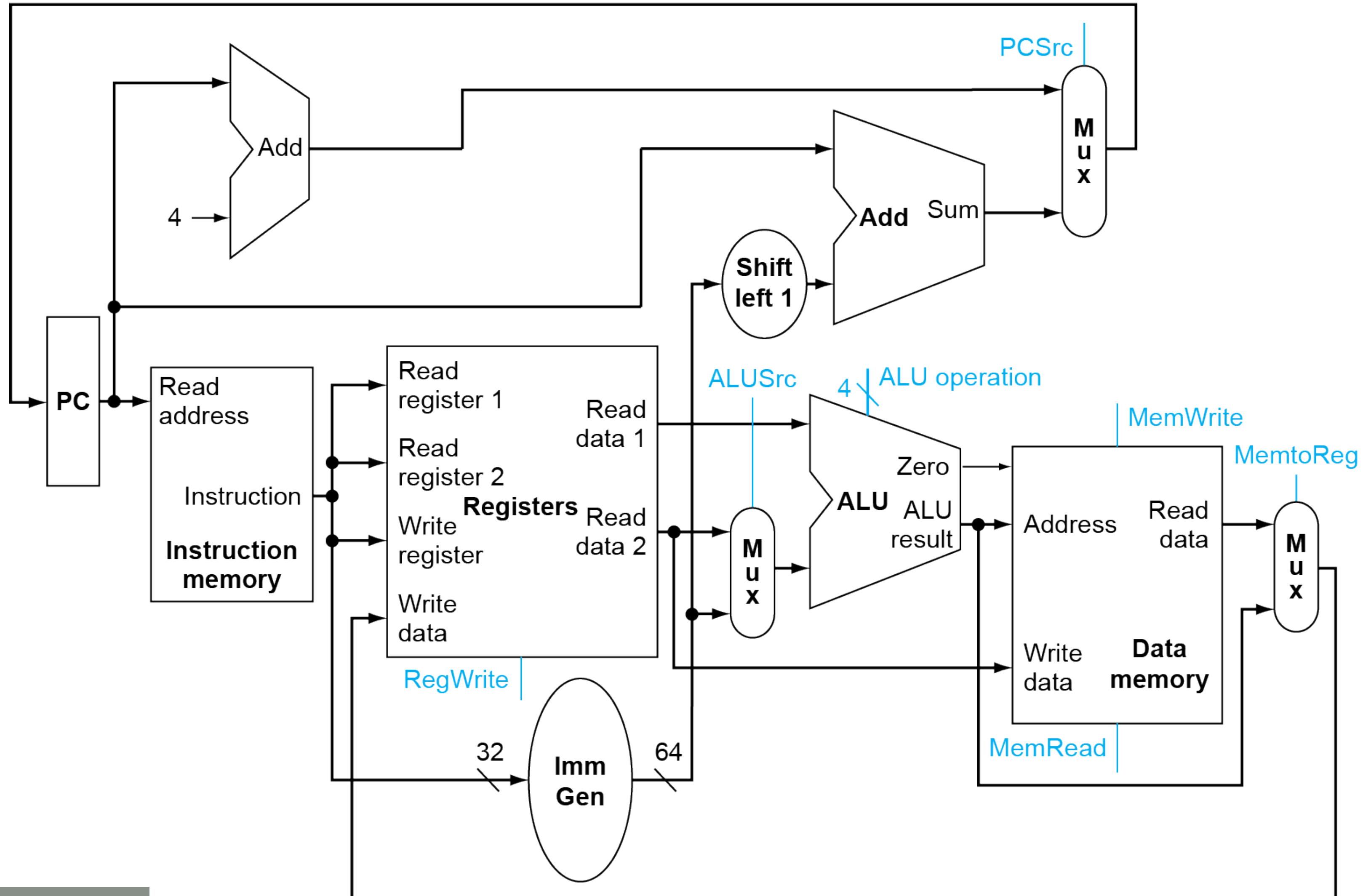
Composing the Elements

- First-cut data path does an instruction in one clock cycle
  - Each datapath element can only do one function at a time
  - Hence, we need separate instruction and data memories. Also we need separate Adders for (PC + 4) and BTA calculation beside the main ALU
- Use multiplexers where alternate data sources are used for different instructions

R-Type/Load/Store Datapath



Full Datapath



ALU Control

- ALU used for
  - Load/Store: F = add
  - Branch: F = subtract
  - R-type: F depends on opcode

ALU control	Function
0000	AND
0001	OR
0010	add
0110	subtract

ALU Control

- Assume 2-bit ALUOp derived from opcode
  - Combinational logic derives ALU control

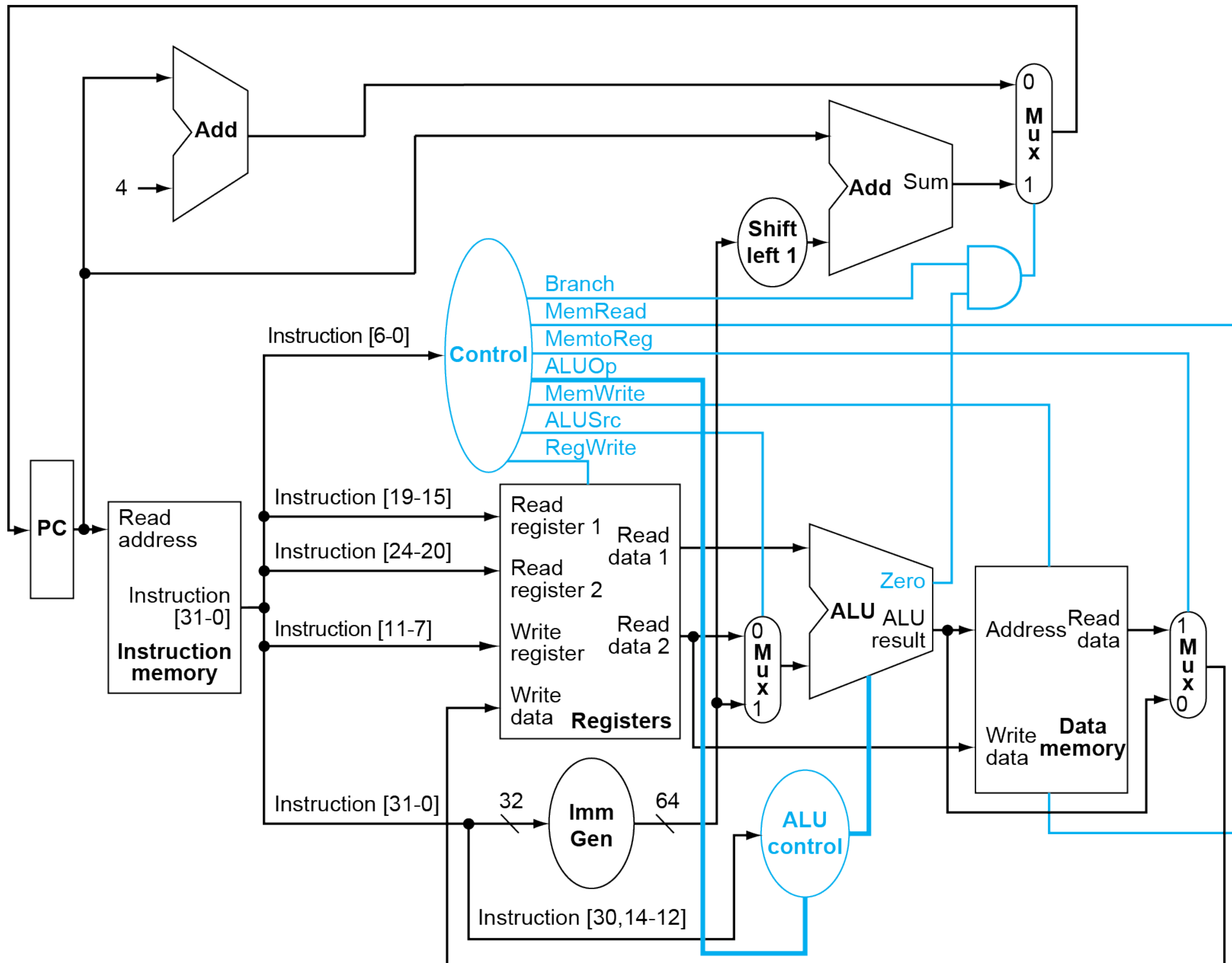
Instruction opcode	ALUOp	Operation	Funct7 field	Funct3 field	Desired ALU action	ALU control input
ld	00	load doubleword	XXXXXXX	XXX	add	0010
sd	00	store doubleword	XXXXXXX	XXX	add	0010
beq	01	branch if equal	XXXXXXX	XXX	subtract	0110
R-type	10	add	0000000	000	add	0010
R-type	10	sub	0100000	000	subtract	0110
R-type	10	and	0000000	111	AND	0000
R-type	10	or	0000000	110	OR	0001

The Main Control Unit

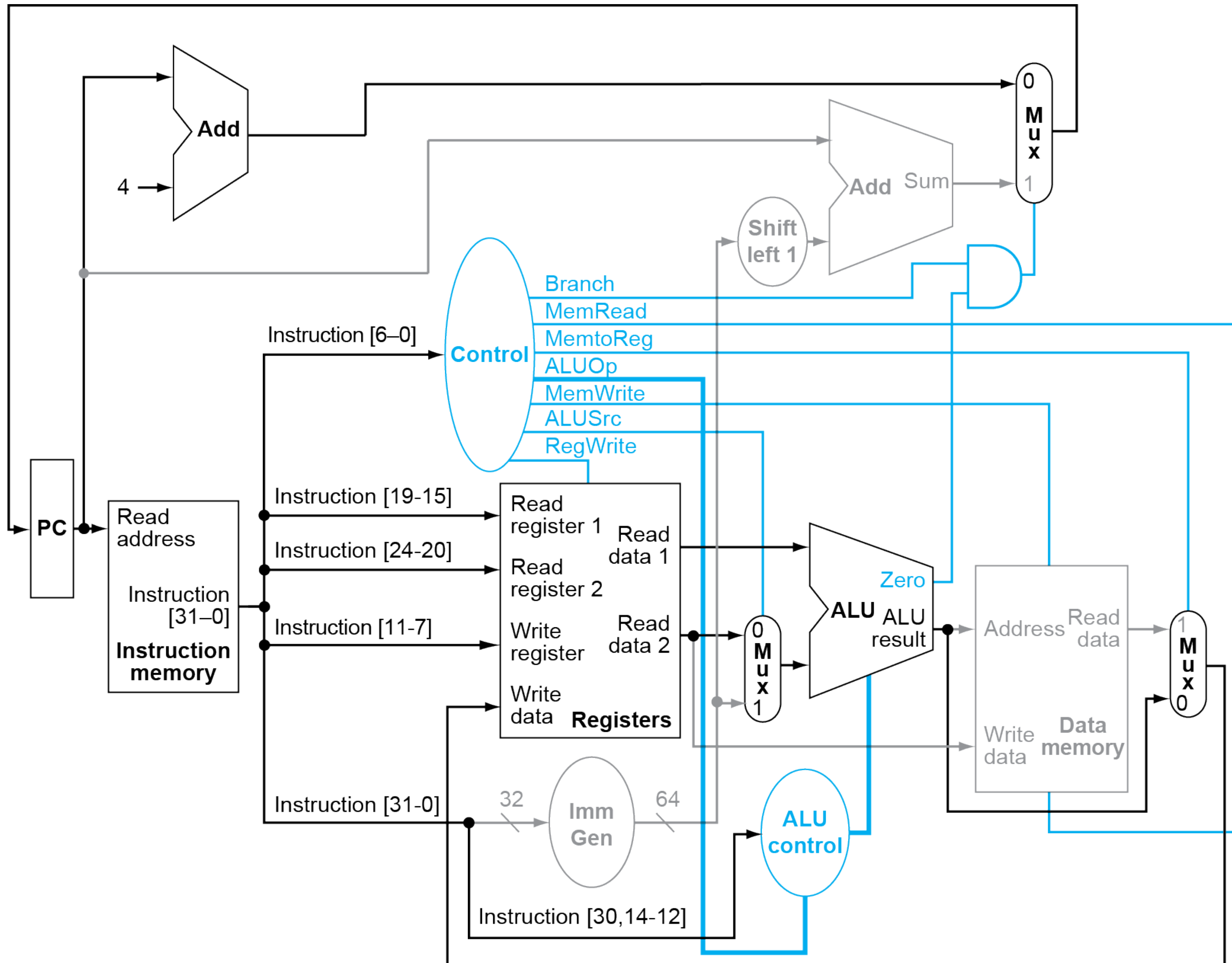
- Control signals derived from instruction's opcode

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

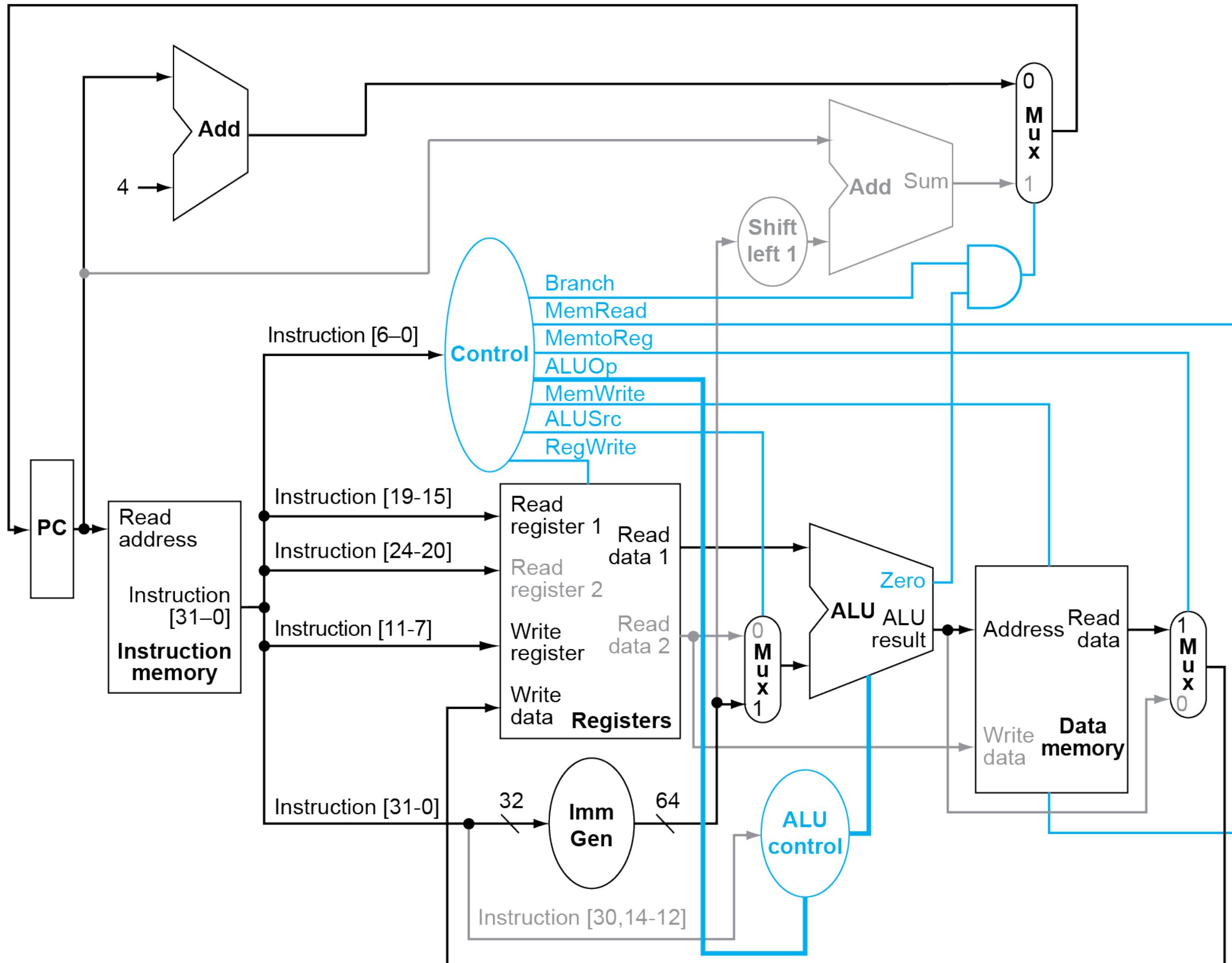
Datapath With Control



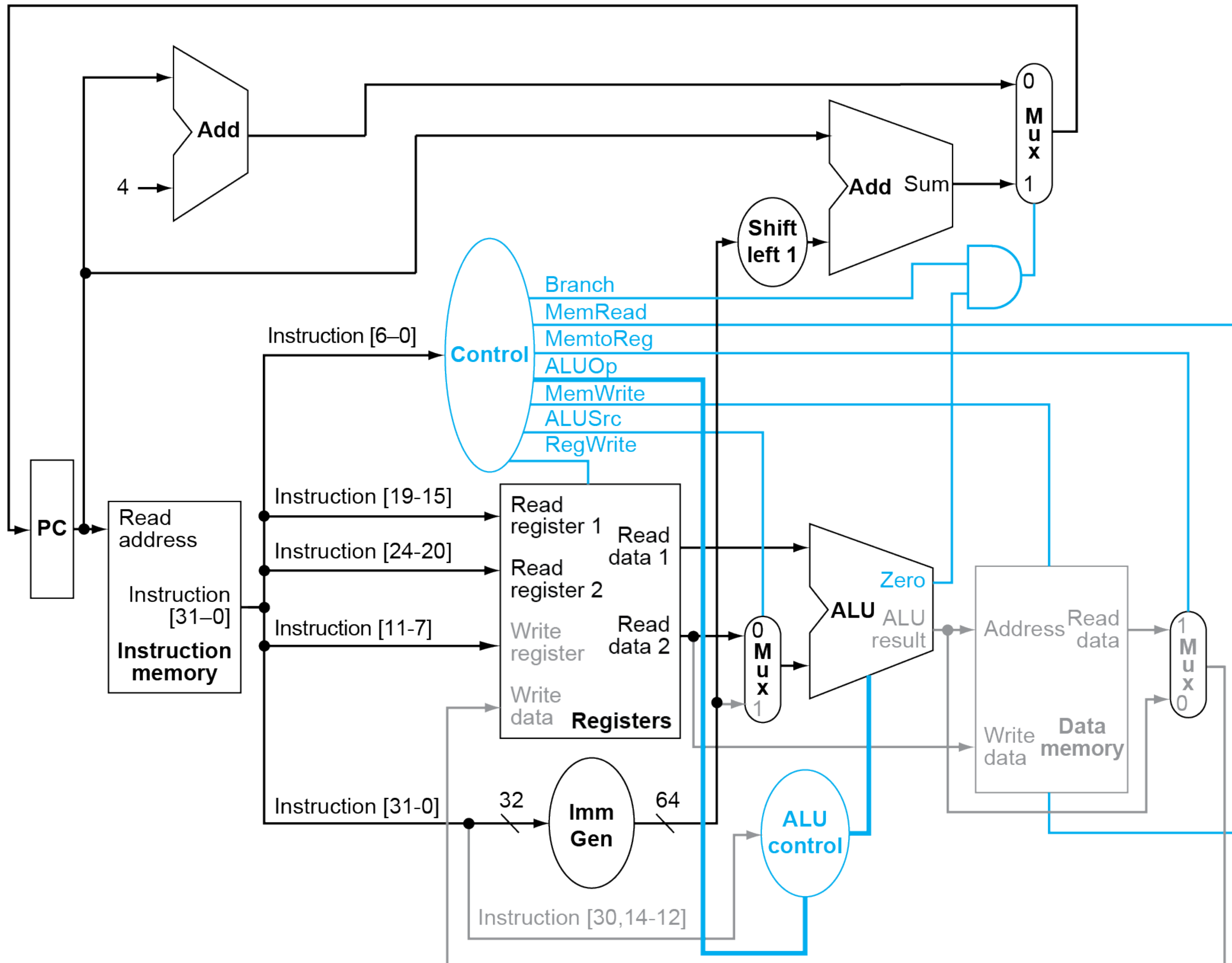
R-Type Instruction



Load Instruction



BEQ Instruction



Control Signals

- 6 single-bit control signals plus 2-bit ALUop signal
  - Total of 8 control signals
- Asserted means that signal is logically high
- Deasserted means that signal is logically low

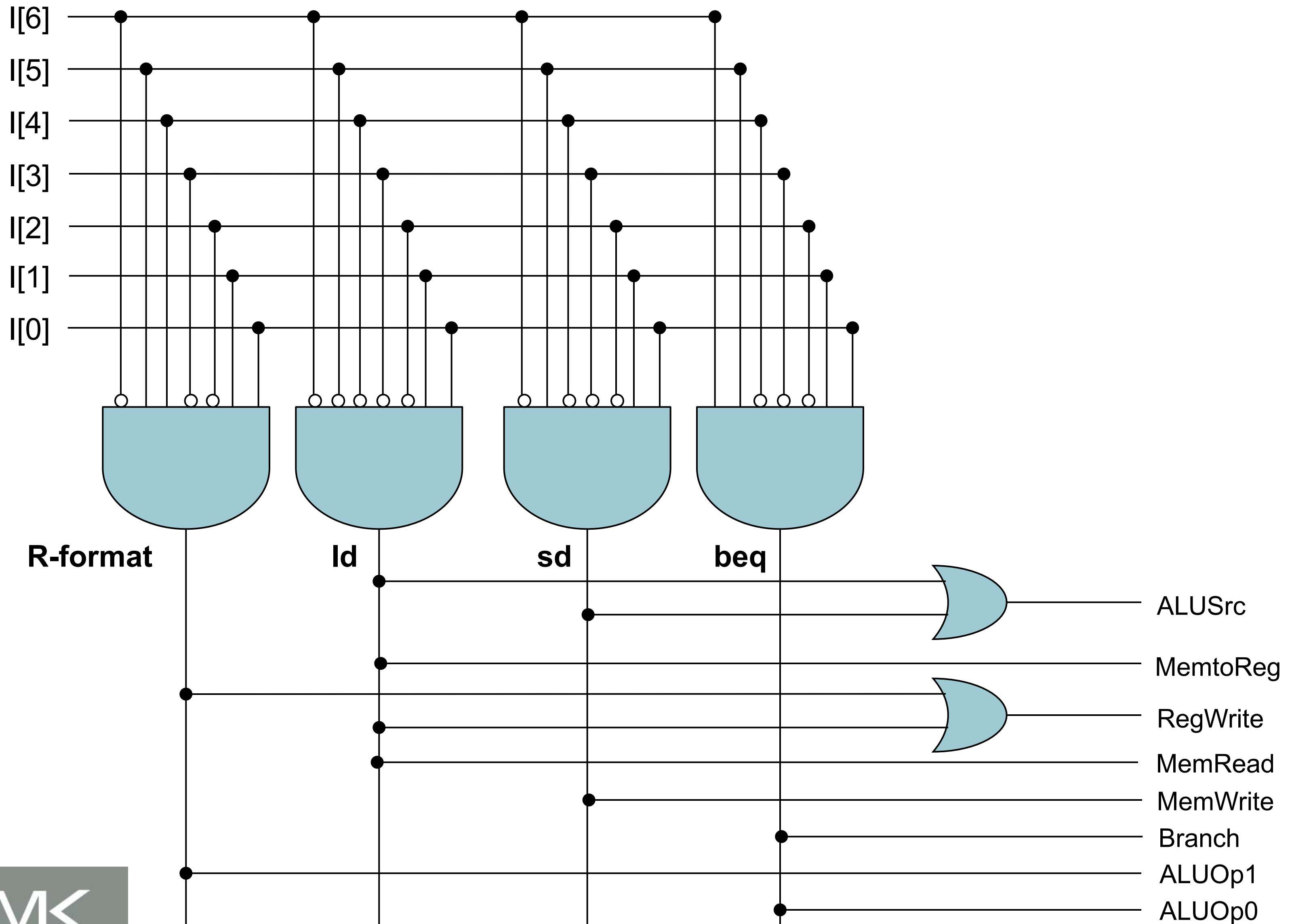
Signal name	Effect when deasserted	Effect when asserted
RegWrite	None.	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extended, 12 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None.	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None.	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

Control Unit

- The input is the Opcode field (7 bits) from the instruction register
- The output is 8 control signals
  - Control signals can be set solely based on the opcode field, except PCSrc (i.e. PCSrc = Branch AND Zero)

Input or output	Signal name	R-format	ld	sd	beq
Inputs	I[6]	0	0	0	1
	I[5]	1	0	1	1
	I[4]	1	0	0	0
	I[3]	0	0	0	0
	I[2]	0	0	0	0
	I[1]	1	1	1	1
	I[0]	1	1	1	1
Outputs	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

Control Unit Design

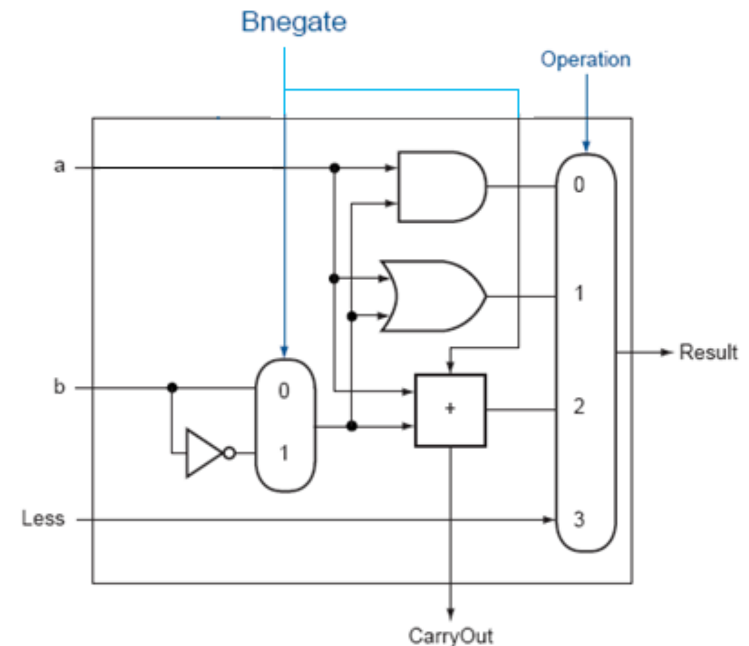


MK MORGAN KAUFMANN

ALU Control

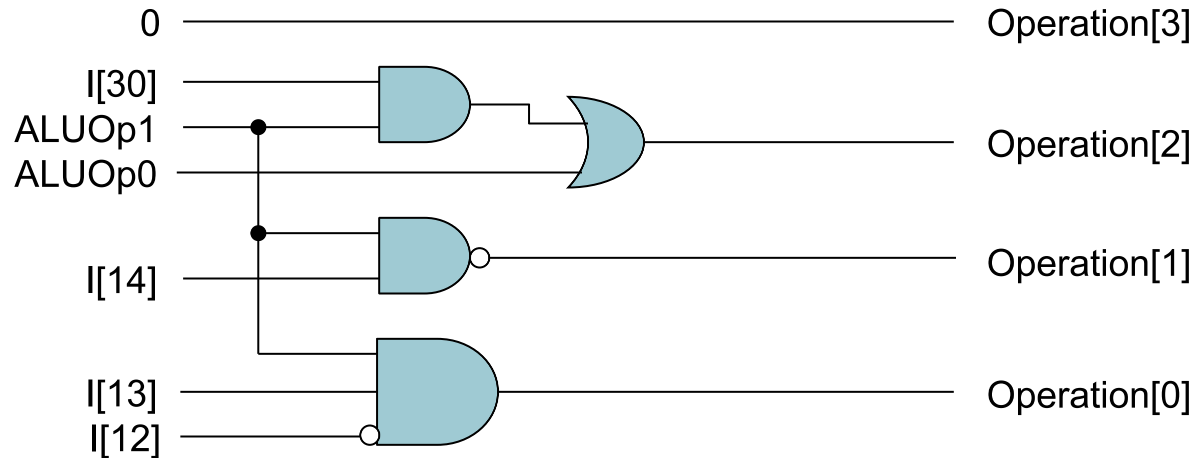
- The ALU control has two inputs:
  1. ALUOp (2 bits) from the control unit
  2. Funct3 and Funct7 fields from the instruction register
- The ALU control has a 4-bit output

Function	Ainvert Operation [3]	Bnegate Operation [2]	Operation [1:0]
and	0	0	00
or	0	0	01
add	0	0	10
sub	0	1	10



ALU Control

ALUOp		Funct7 field							Funct3 field			
ALUOp1	ALUOp0	I[31]	I[30]	I[29]	I[28]	I[27]	I[26]	I[25]	I[14]	I[13]	I[12]	Operation
0	0	X	X	X	X	X	X	X	X	X	X	0010
0	1	X	X	X	X	X	X	X	X	X	X	0110
1	0	0	0	0	0	0	0	0	0	0	0	0010
1	0	0	1	0	0	0	0	0	0	0	0	0110
1	0	0	0	0	0	0	0	0	1	1	1	0000
1	0	0	0	0	0	0	0	0	1	1	0	0001



Performance Issues

- Longest delay determines clock period
  - Critical path: load instruction
  - Instruction memory → register file → ALU → data memory → register file
- Not feasible to vary period for different instructions
- Violates design principle
  - Making the common case fast
- We will improve performance by pipelining

MK MORGAN KAUFMANN

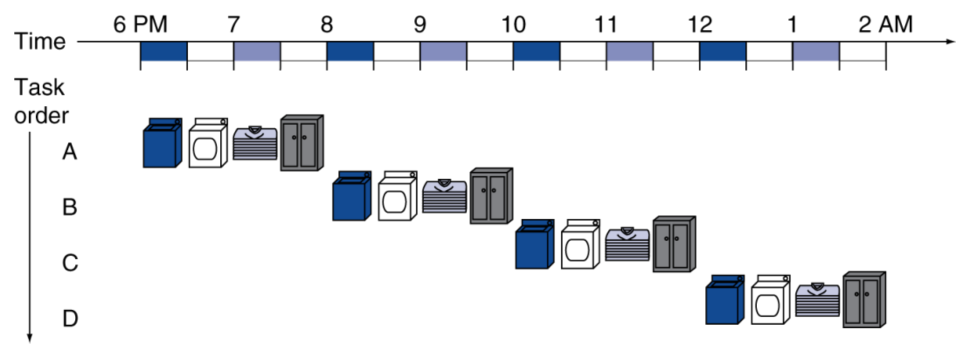
Pipelining

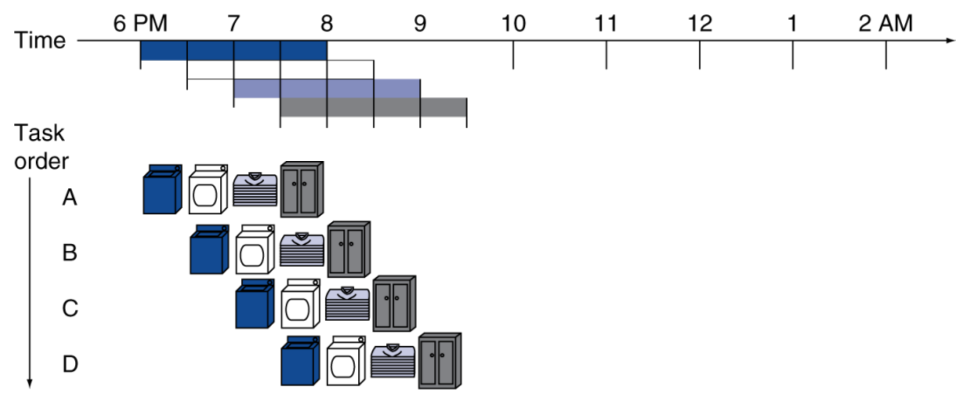
- An implementation technique in which multiple instructions are overlapped in execution
  - C.f. Single-Cycle and Multi-Cycle implementations in which only a single instruction is executing at any time
- *Today, pipelining is nearly universal*

MK MORGAN KAUFMANN

Pipelining Analogy

- Pipelined laundry: overlapping execution
  - Parallelism improves performance





- Four loads:
  - Speedup = 8/3.5 = 2.3
- Non-stop:
  - Speedup = 2n/0.5n + 1.5 ≈ 4 = number of stages

RISC-V Pipeline

- Five stages, one step per stage
  1. IF: Instruction fetch from memory
  2. ID: Instruction decode & register read
  3. EX: Execute operation or calculate address
  4. MEM: Access memory operand
  5. WB: Write result back to register

- Stages can operate concurrently as long as separate resources are available for each stage
  - Pipeline design is based on Single-Cycle CPU design

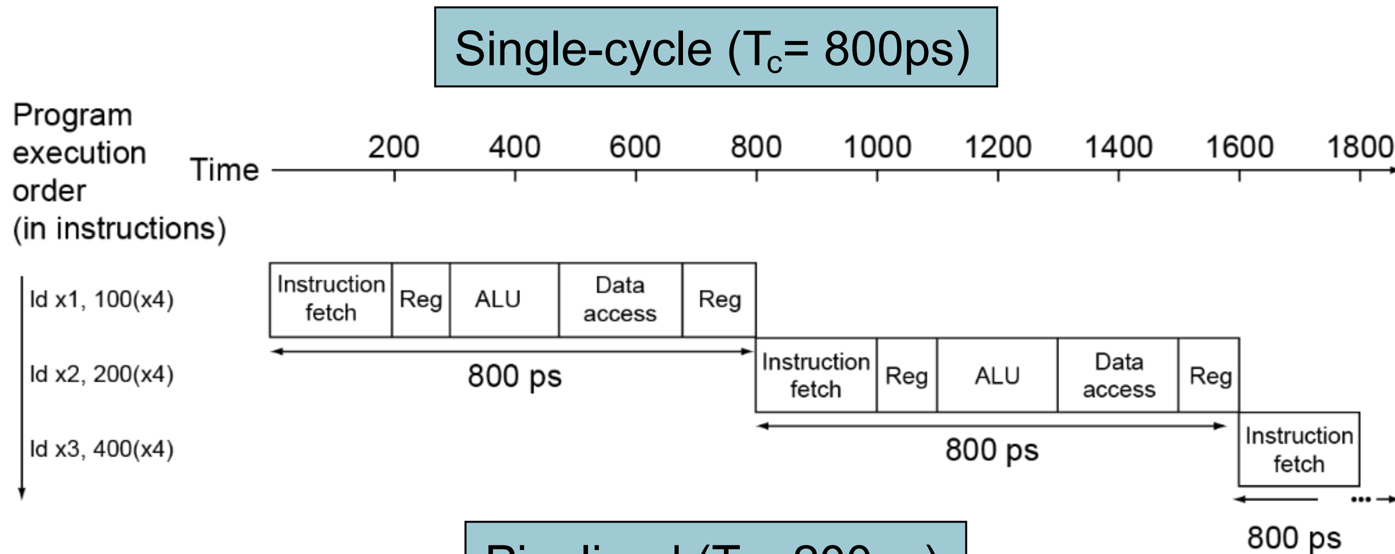
Pipeline Performance

- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages
- Compare pipelined datapath with single-cycle datapath

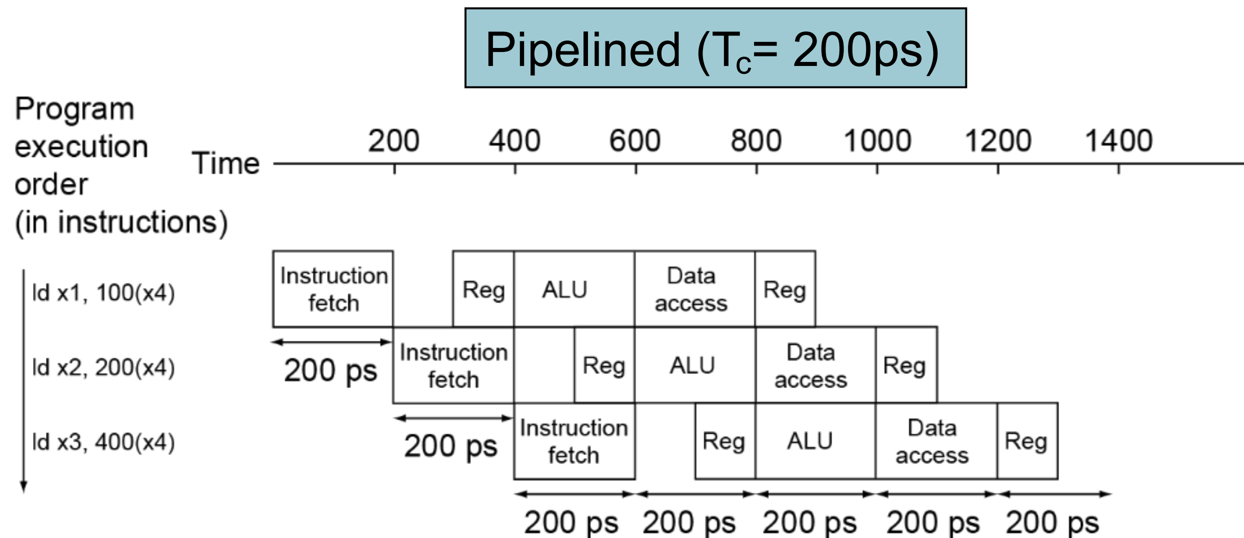
Instr	Instr fetch	Register read	ALU op	Memory access	Register write	Total time
ld	200ps	100 ps	200ps	200ps	100 ps	800ps
sd	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

Pipeline Performance

Single-cycle (T_c= 800ps)



Pipelined (T_c= 200ps)



Pipeline Speedup

- If all stages are balanced
  - i.e., all take the same time
  - $\text{Time between instructions}_{\text{pipelined}} = \dfrac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of stages}}$
- If not balanced, speedup is less
- Speedup due to increased throughput
  - Latency (time for each instruction) ***does not decrease***

MK MORGAN KAUFMANN

Pipelining and ISA Design

- RISC-V ISA designed for pipelining
  - All instructions are 32-bits
    - Easier to fetch and decode in one cycle
    - c.f. x86: 1- to 17-byte instructions
  - Few and regular instruction formats
    - Can decode and read registers in one step
  - Load/store addressing
    - Can calculate address in 3rd stage, access memory in 4th stage

Hazards

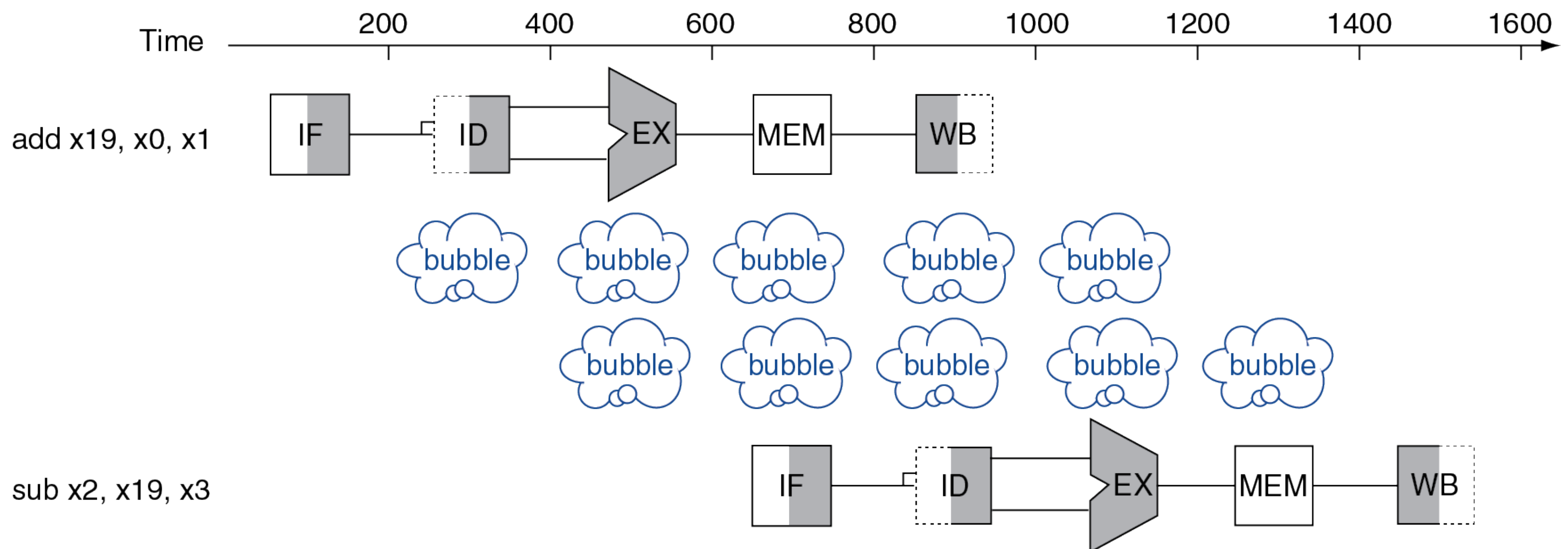
- Situations that prevent starting the next instruction in the next cycle
- Structure hazards
  - A required resource is busy
- Data hazard
  - Need to wait for previous instruction to complete its data read/write
- Control hazard
  - Deciding on control action depends on previous instruction

Structure Hazards

- Conflict for use of a resource
- In RISC-V pipeline with a single memory
  - Load/store requires data access
  - Instruction fetch would have to *stall* for that cycle
    - Would cause a pipeline "bubble"
- Hence, pipelined datapaths require separate instruction/data memories
  - Or separate instruction/data caches
- ***RISC-V pipeline is designed carefully such that there are no structure hazards***

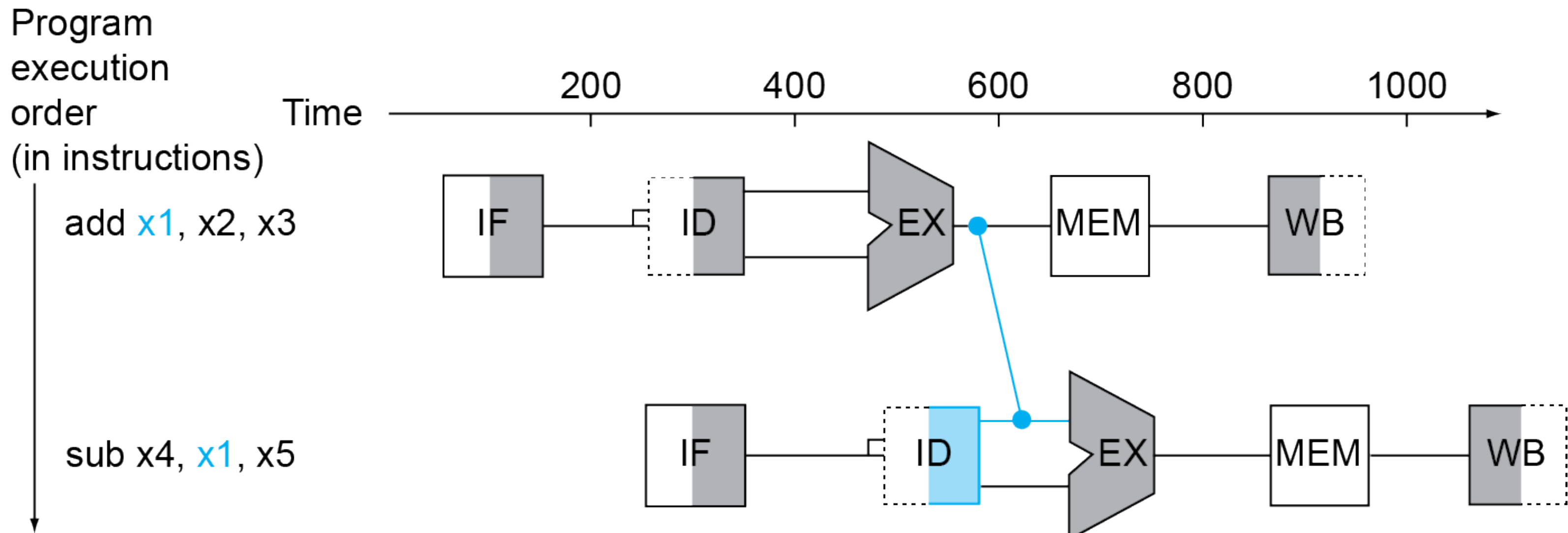
Data Hazards

- An instruction depends on completion of data access by a previous instruction
  - ```
    add    x19, x0, x1
    sub    x2, x19, x3
    ```
- Pipeline must be stalled (i.e. stopped)
  - Pipeline stall ≡ bubble


    ```

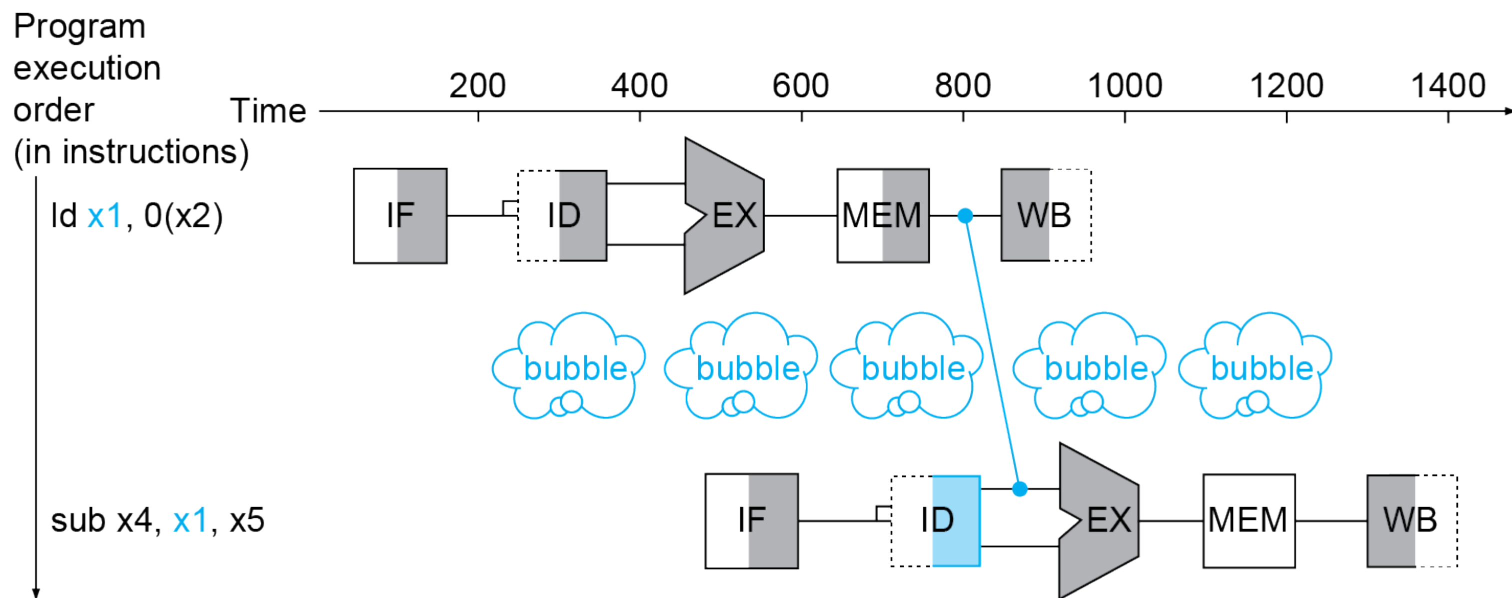
Forwarding (aka Bypassing)

- Use result when it is computed
  - Don't wait for it to be stored in a register
  - Requires extra connections in the datapath
- Forwarding paths are valid only if the destination stage is later in time than the source stage
  - Ex: There can't be a path from the output of the memory stage to the input of the execution stage



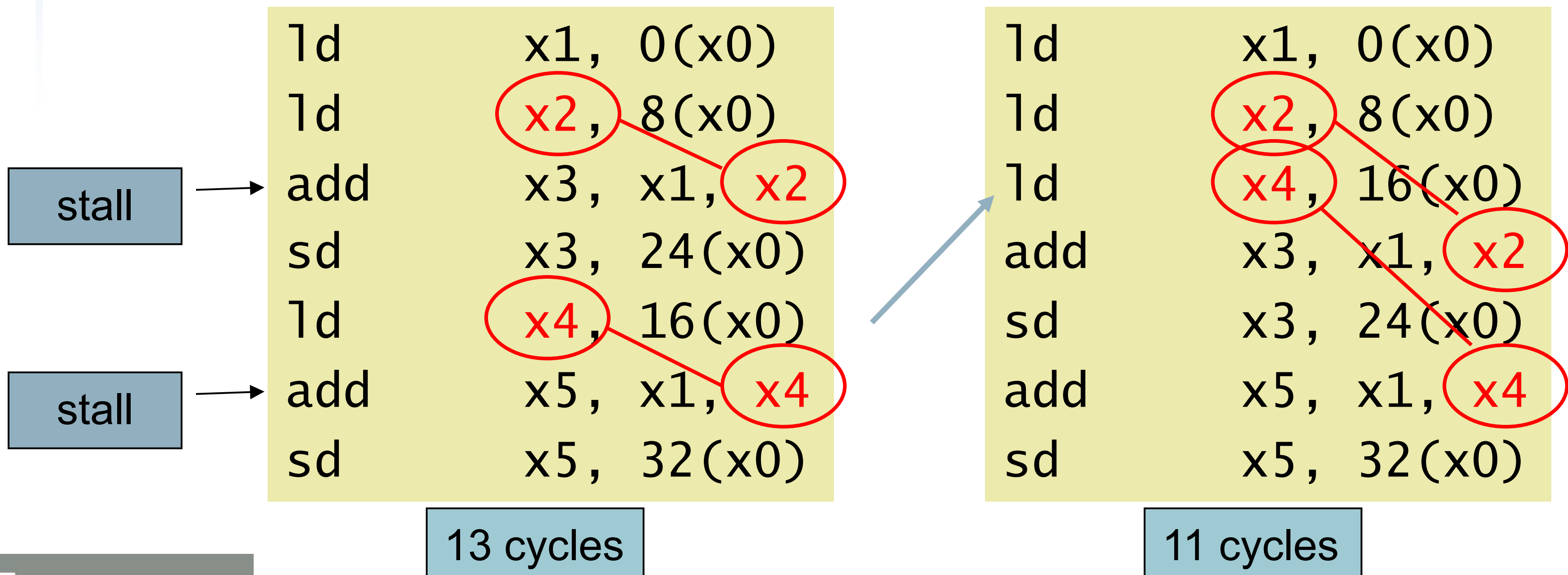
Load-Use Data Hazard

- Can't always avoid stalls by forwarding
  - If value not computed when needed
  - Can't forward backward in time!



Code Scheduling to Avoid Stalls

- Reorder code to avoid use of load result in the next instruction
- C code for a `a = b + e; c = b + f;`
- `Assume that a, b, c, e, f are in memory with offsets (24, 0, 32, 8, 16) from x0`

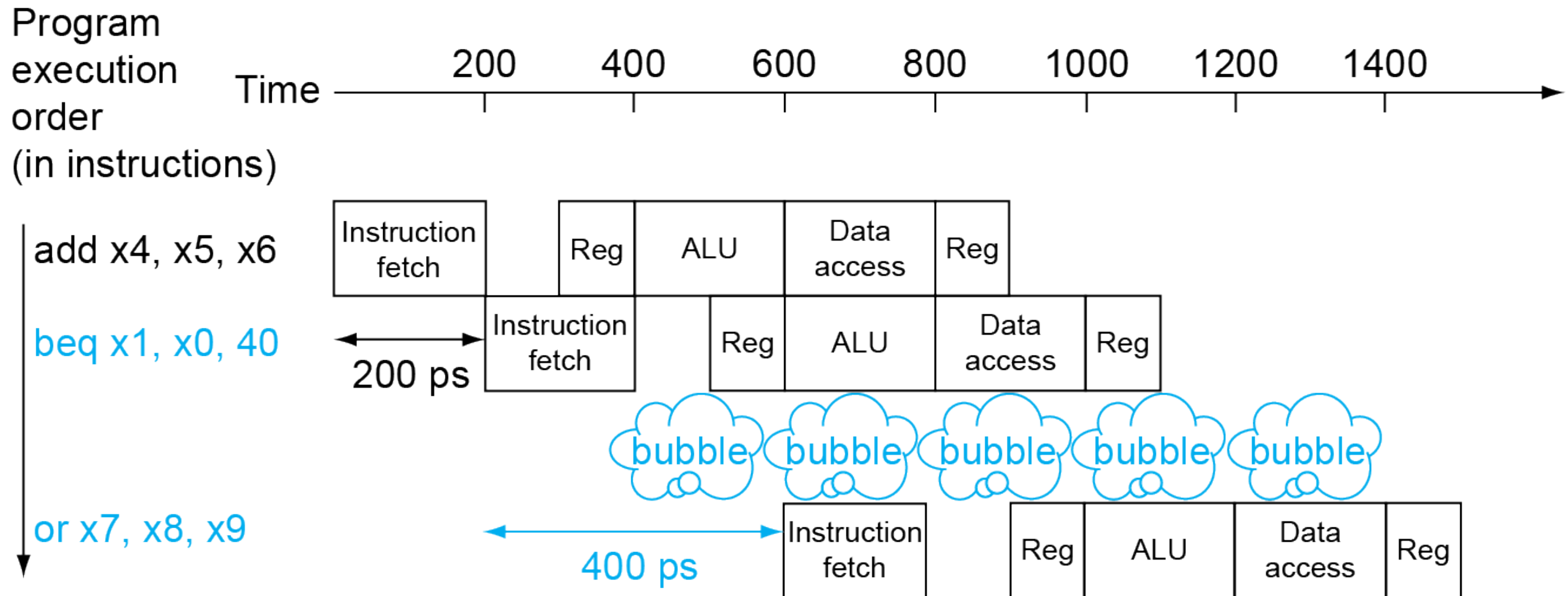


Control Hazards

- Branch determines flow of control
  - Fetching next instruction depends on branch outcome
  - Pipeline can’t always fetch correct instruction
    - Still working on ID stage of branch
- In RISC-V pipeline
  - Need to compare registers and compute target early in the pipeline
  - Add hardware to do it in ID stage

Stall on Branch

- Wait until branch outcome determined before fetching next instruction



Branch Prediction

- Longer pipelines can't readily determine branch outcome early
  - Stall penalty becomes unacceptable
- Predict outcome of branch
  - Only stall if prediction is wrong
- In RISC-V pipeline
  - Can predict branches not taken
  - Fetch instruction after branch, with no delay

More-Realistic Branch Prediction

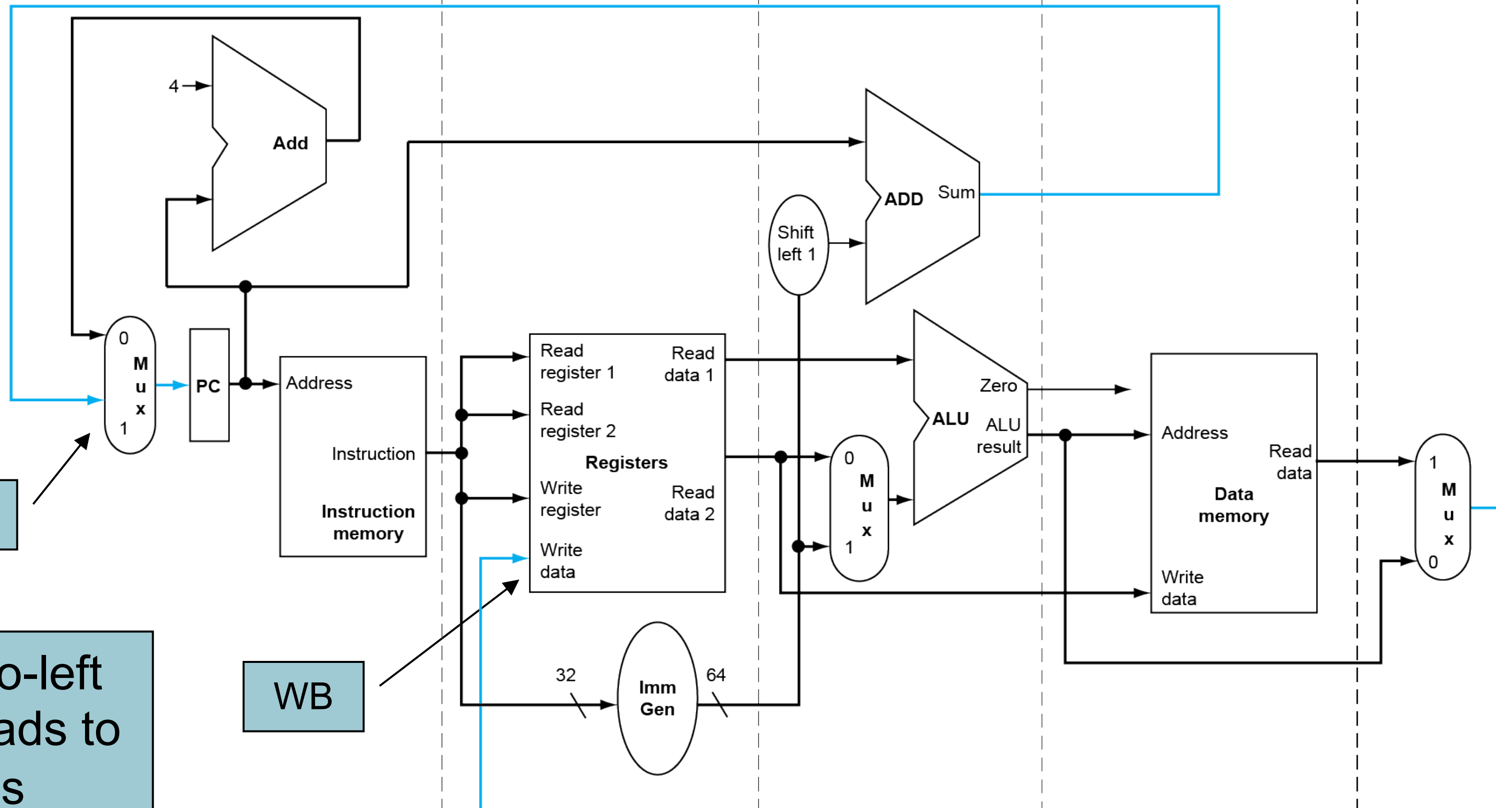
- Static branch prediction
  - Based on typical branch behavior
  - Example: loop and if-statement branches
    - Predict backward branches taken
    - Predict forward branches not taken
- Dynamic branch prediction
  - Hardware measures actual branch behavior
    - e.g., record recent history of each branch
  - Assume future behavior will continue the trend
    - When wrong, stall while re-fetching, and update history

Pipeline Summary

The BIG Picture

- Pipelining improves performance by increasing instruction throughput
  - Executes multiple instructions in parallel
  - Each instruction has the same latency
- Subject to hazards
  - Structure, data, control
- Instruction set design affects complexity of pipeline implementation

RISC-V Pipelined Datapath



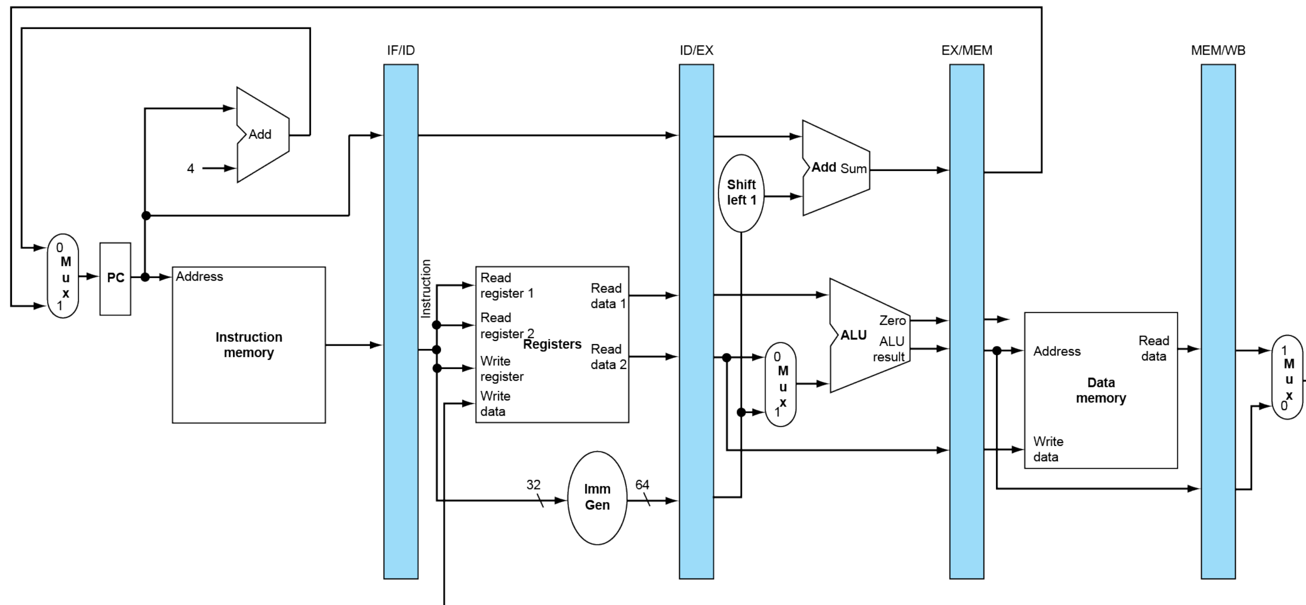
MEM

WB

Right-to-left flow leads to hazards

Pipeline registers

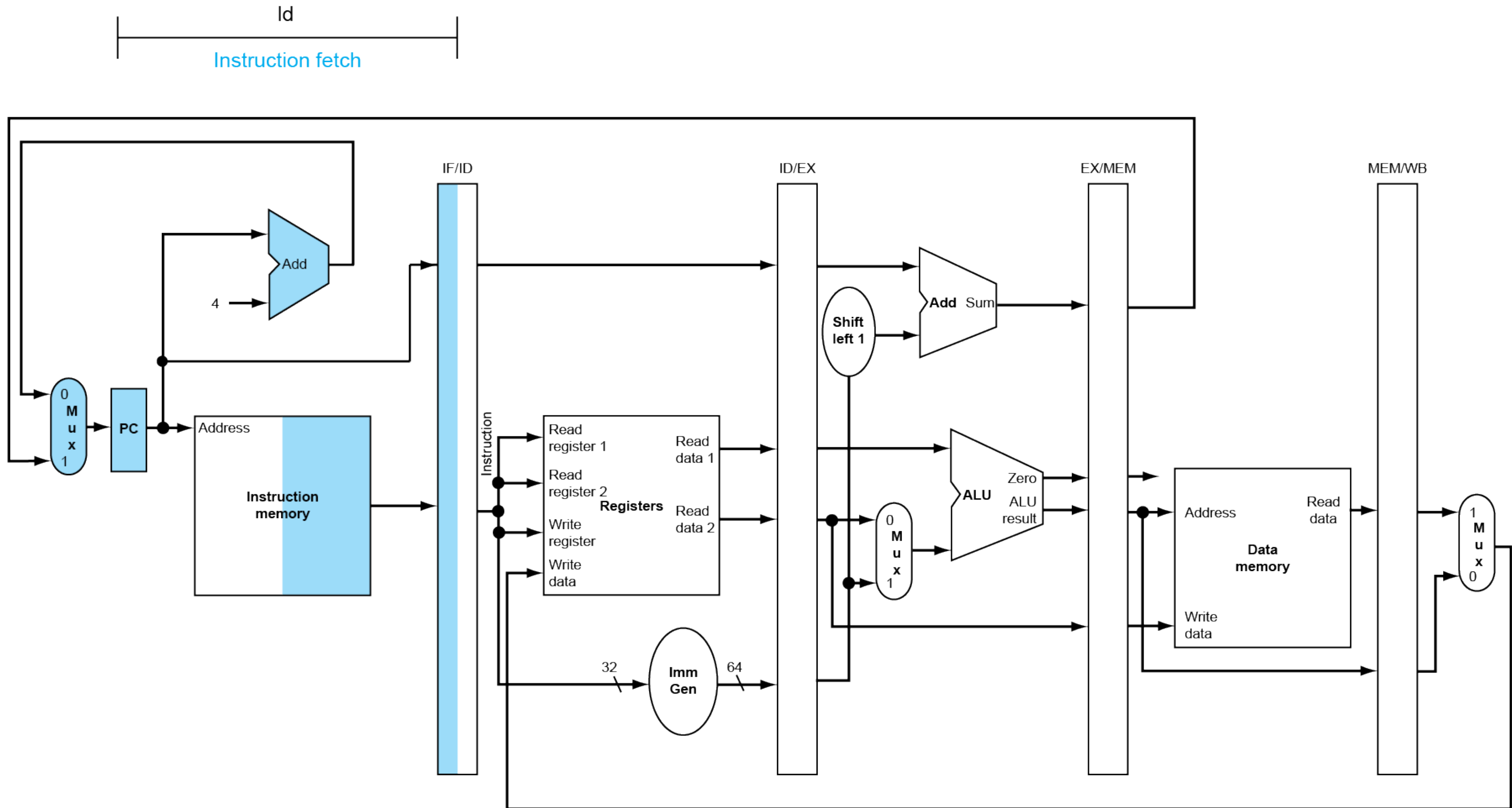
- Need registers between stages
  - To hold information produced in previous cycle
- Stage registers must be wide enough to store all the data corresponding to the lines that go through them
  - IF/ID is 96-bit : 32-bit instruction and 64-bit for PC
  - ID/EX is 256-bit: (rs1), (rs2), Sign-Extended value, and PC
  - EX/MEM is 193-bit and MEM/WB is 128-bit



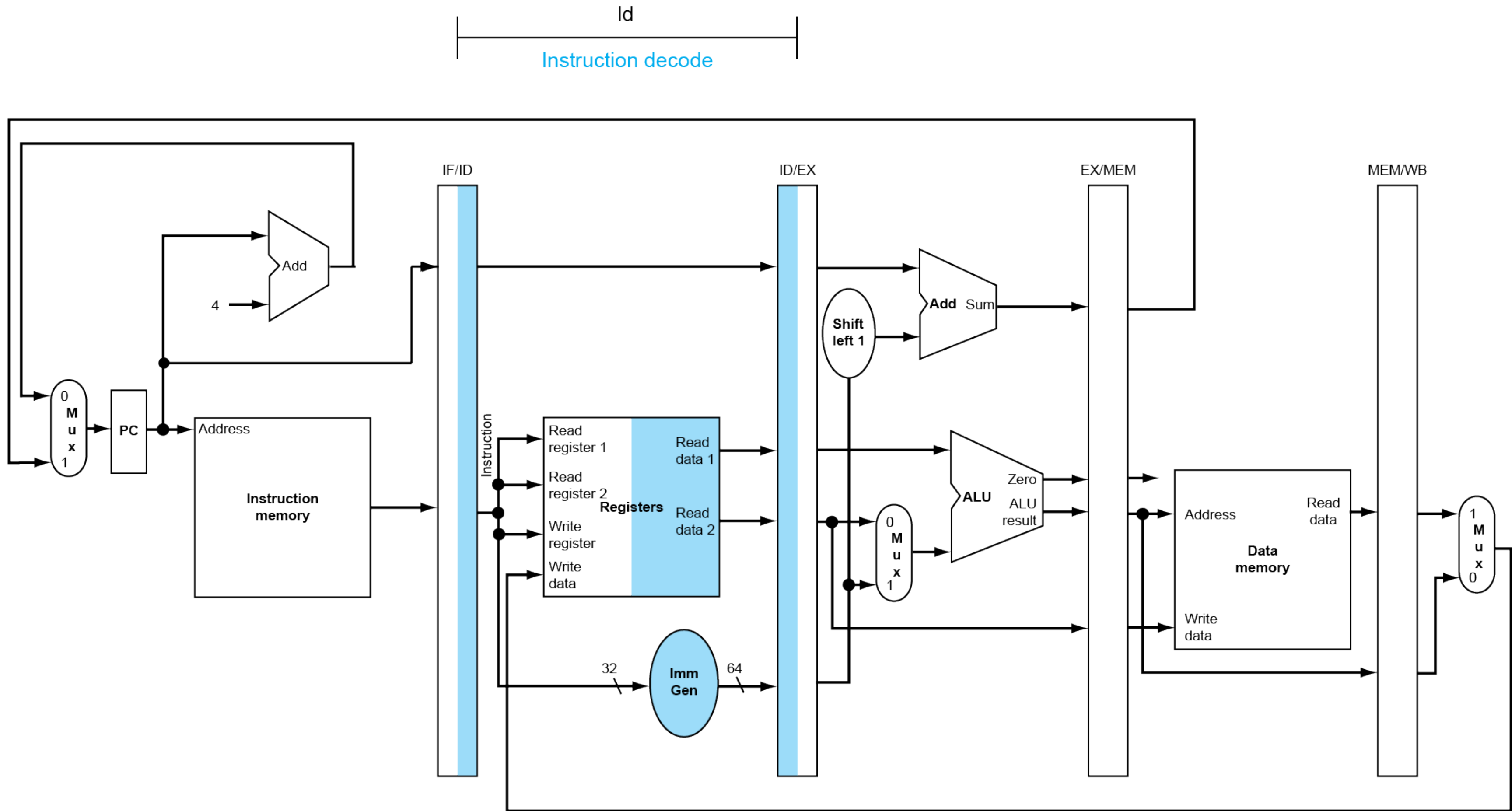
Pipeline Operation

- Cycle-by-cycle flow of instructions through the pipelined datapath
  - "Single-clock-cycle" pipeline diagram
    - Shows pipeline usage in a single cycle
    - Highlight resources used
  - c.f. "multi-clock-cycle" diagram
    - Graph of operation over time
- We'll look at "single-clock-cycle" diagrams for load & store

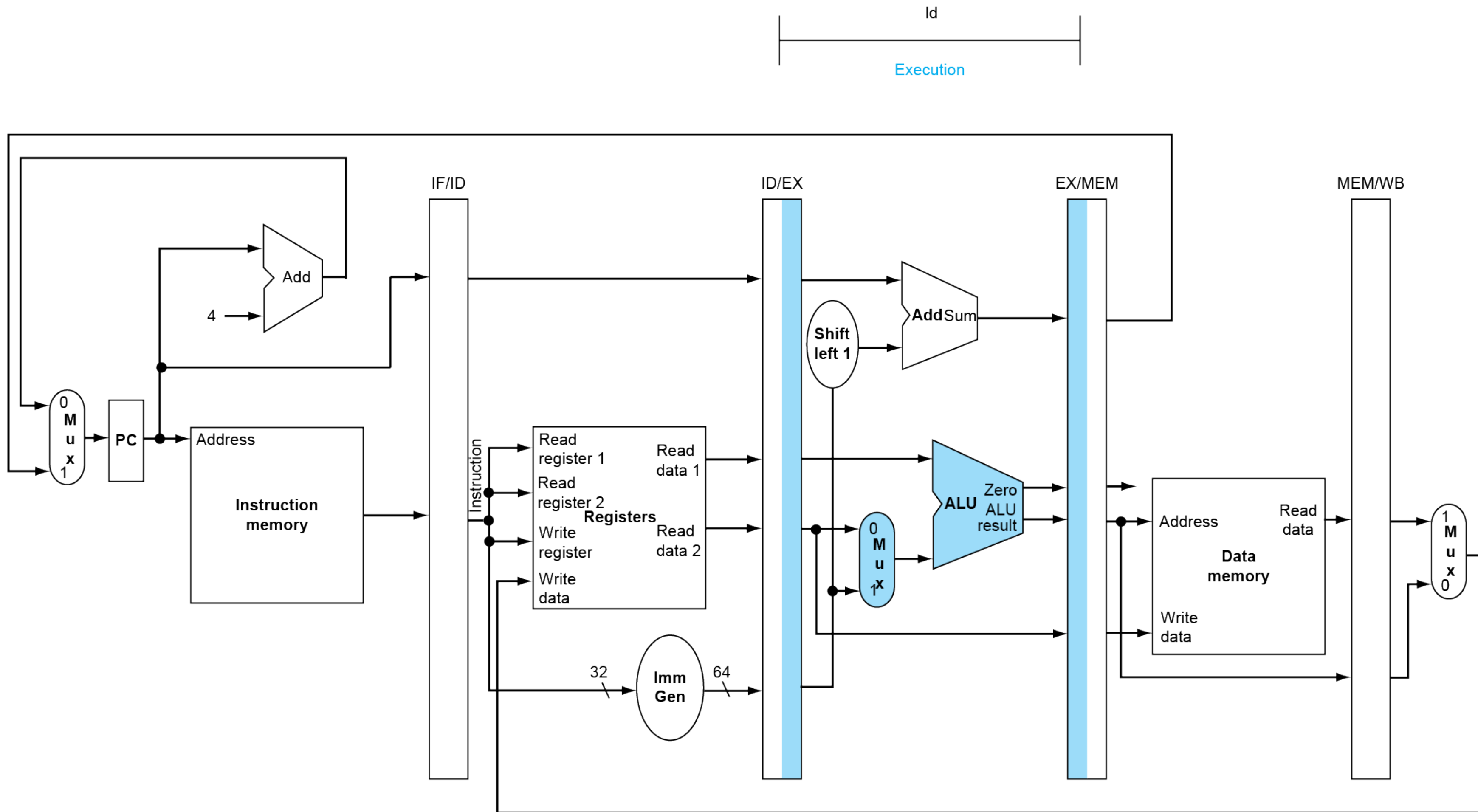
IF for Load, Store, …



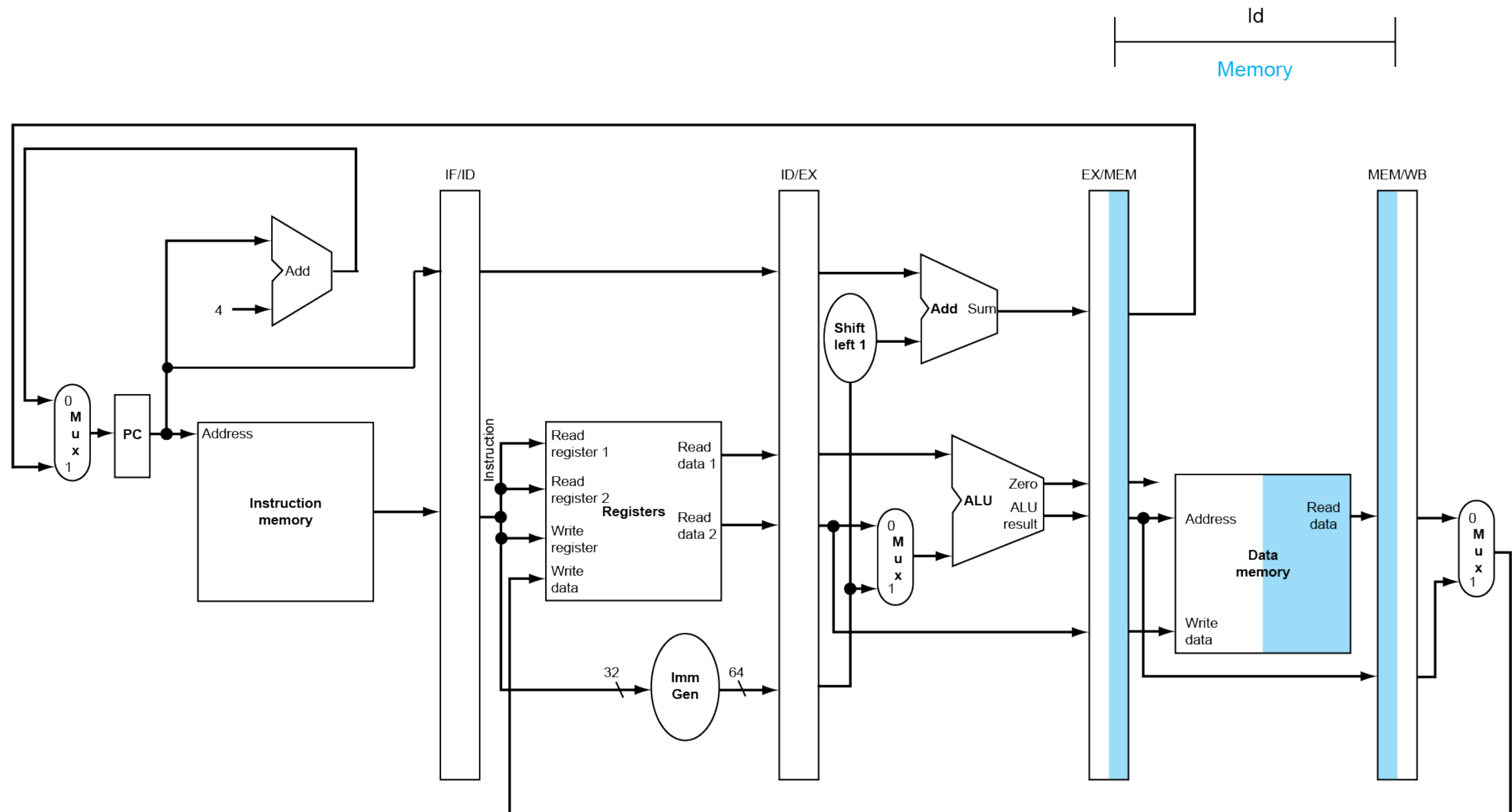
ID for Load, Store, …



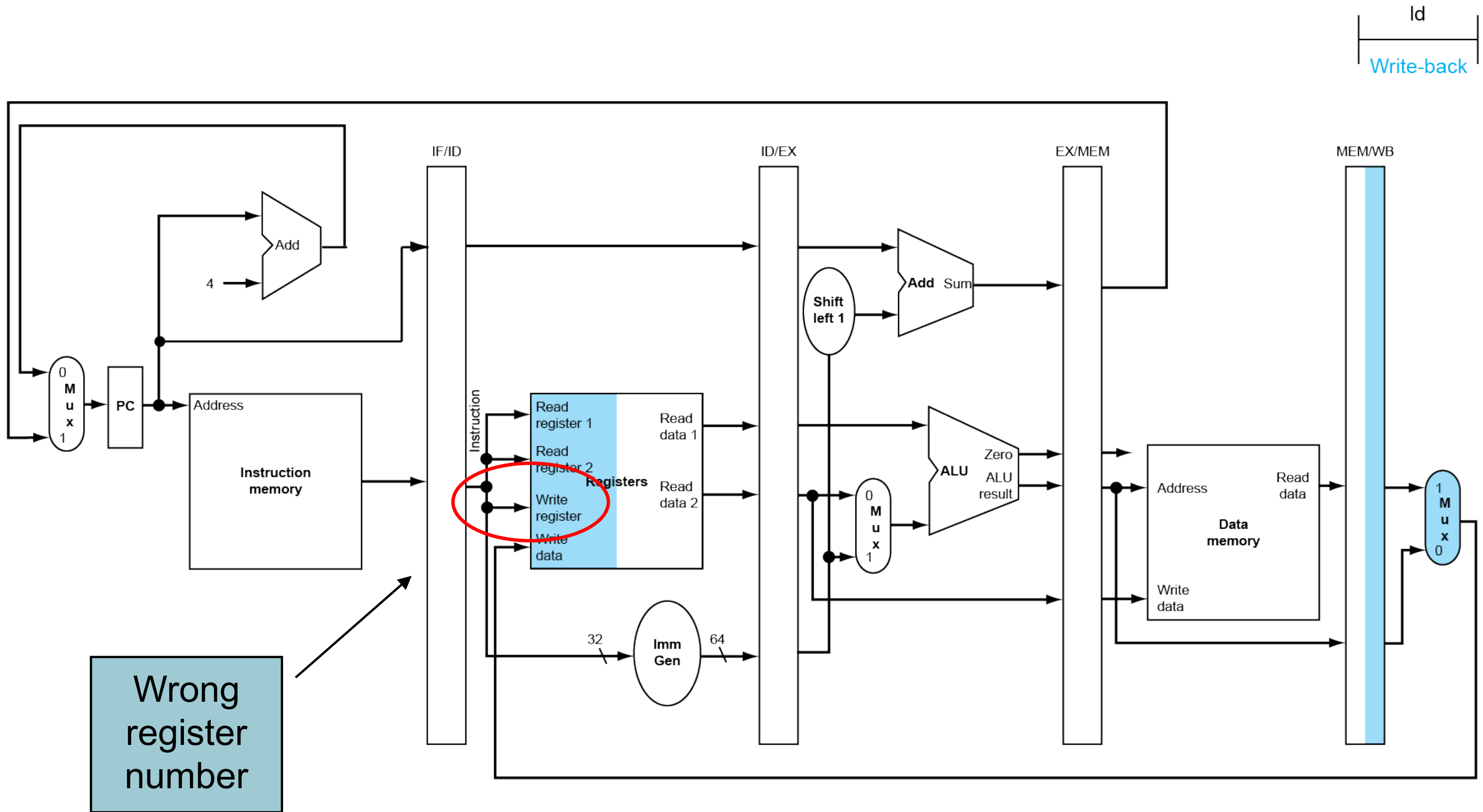
EX for Load



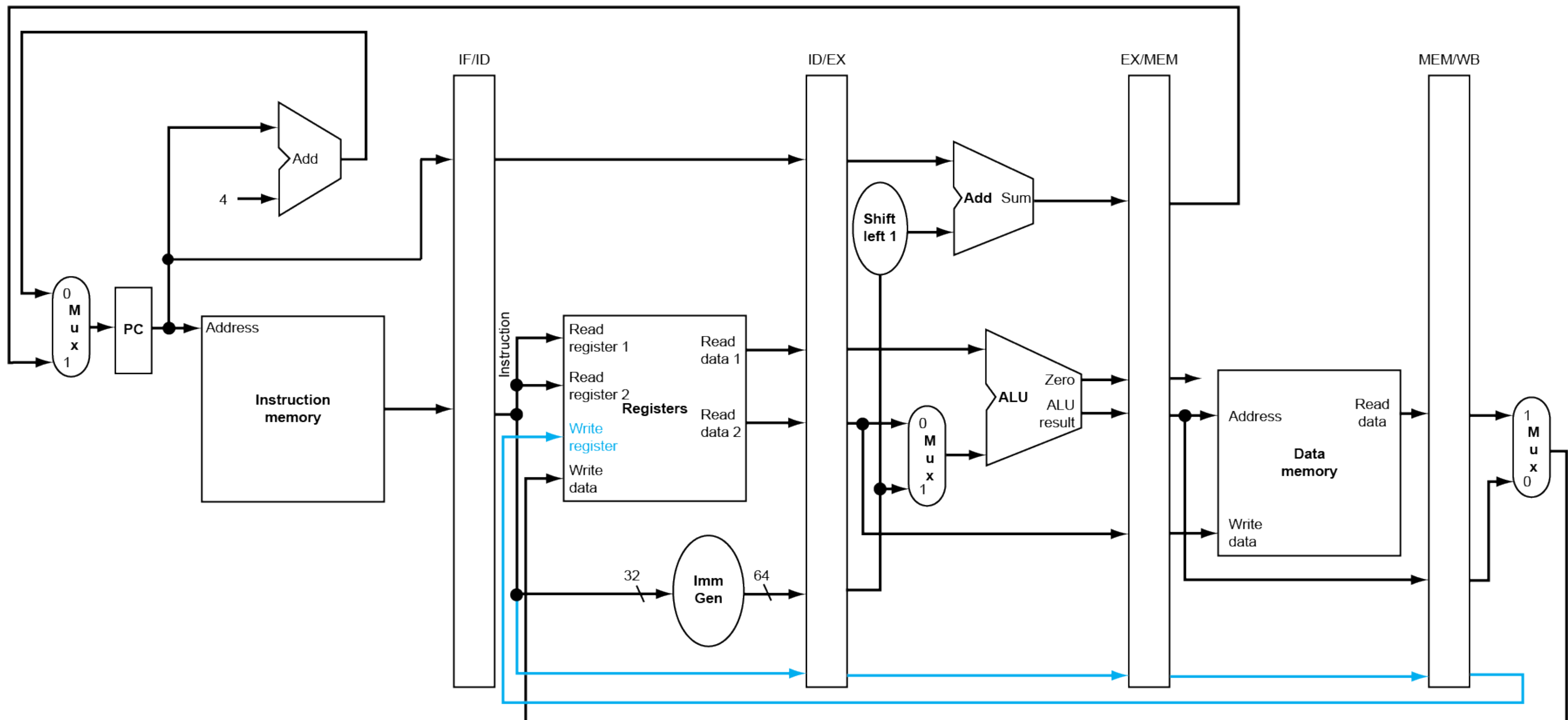
MEM for Load



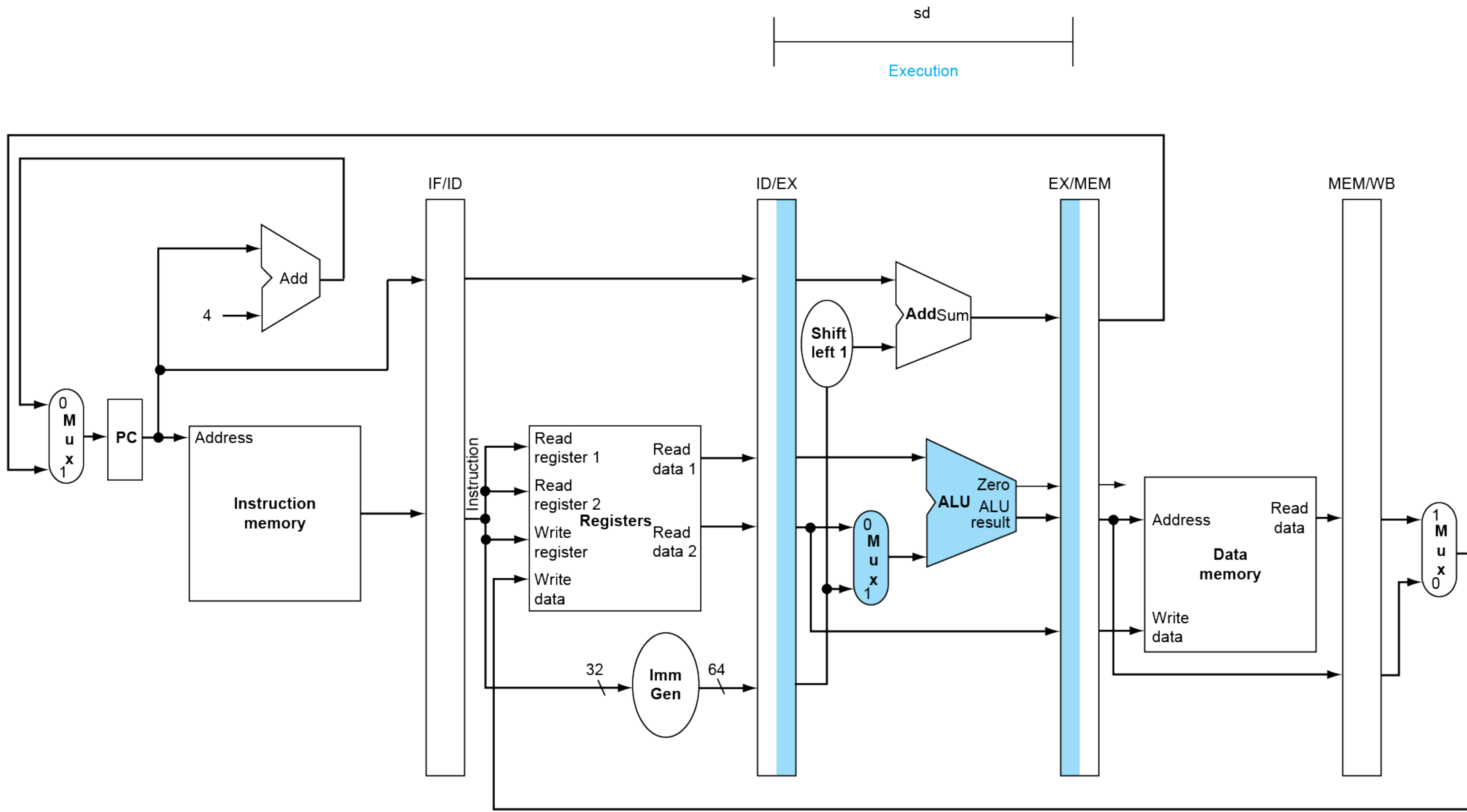
WB for Load



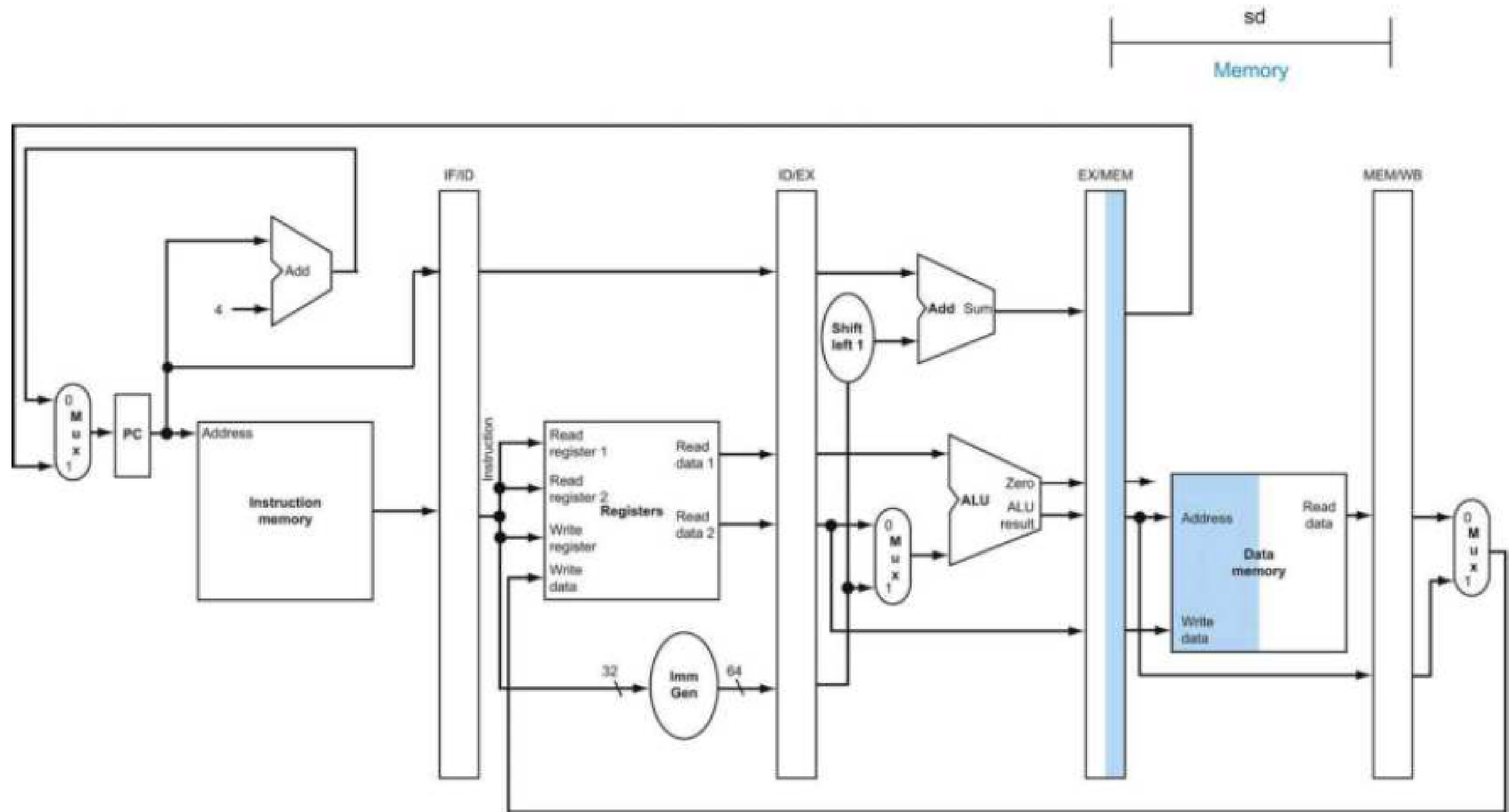
Corrected Datapath for Load



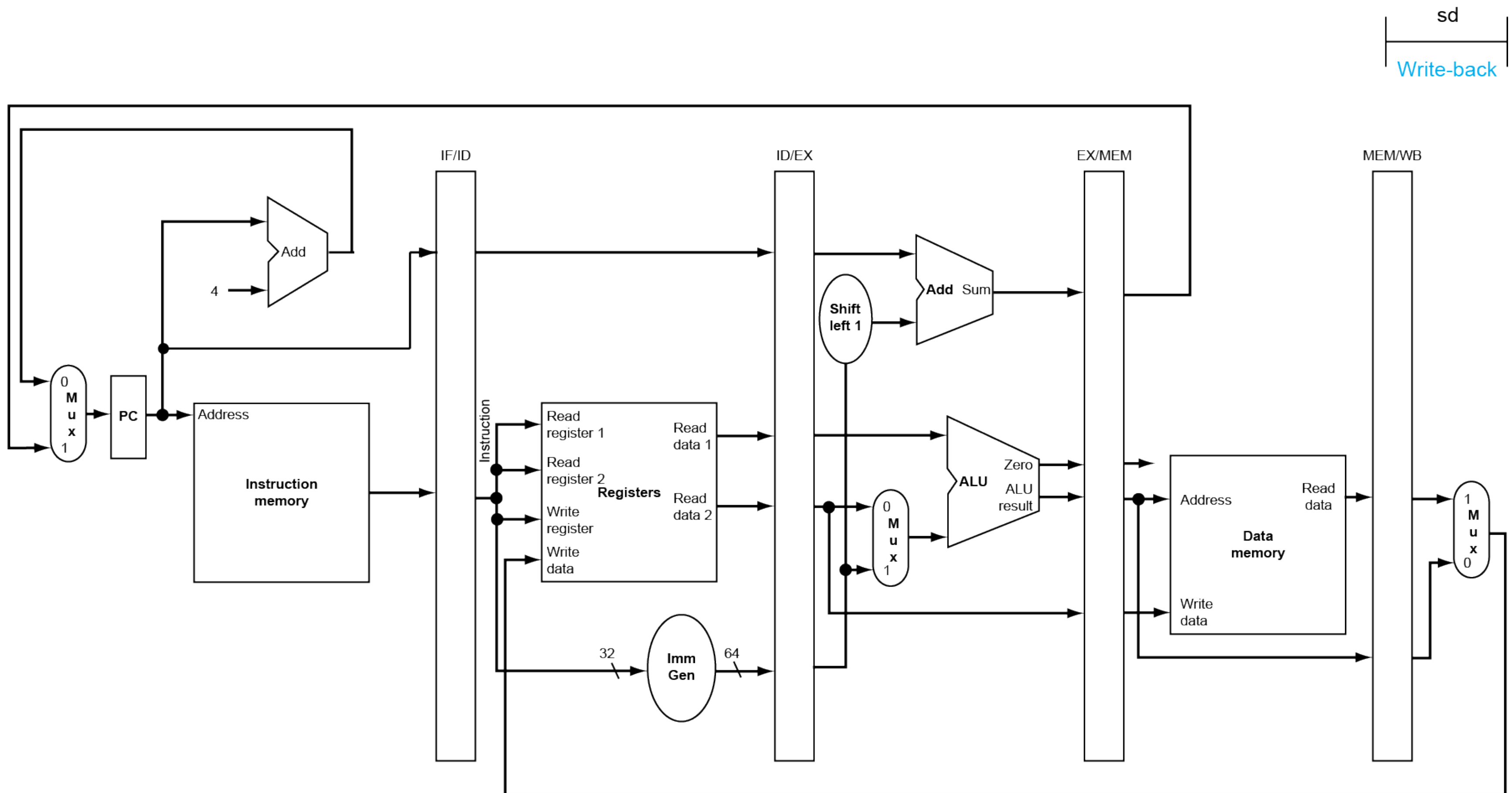
EX for Store



MEM for Store

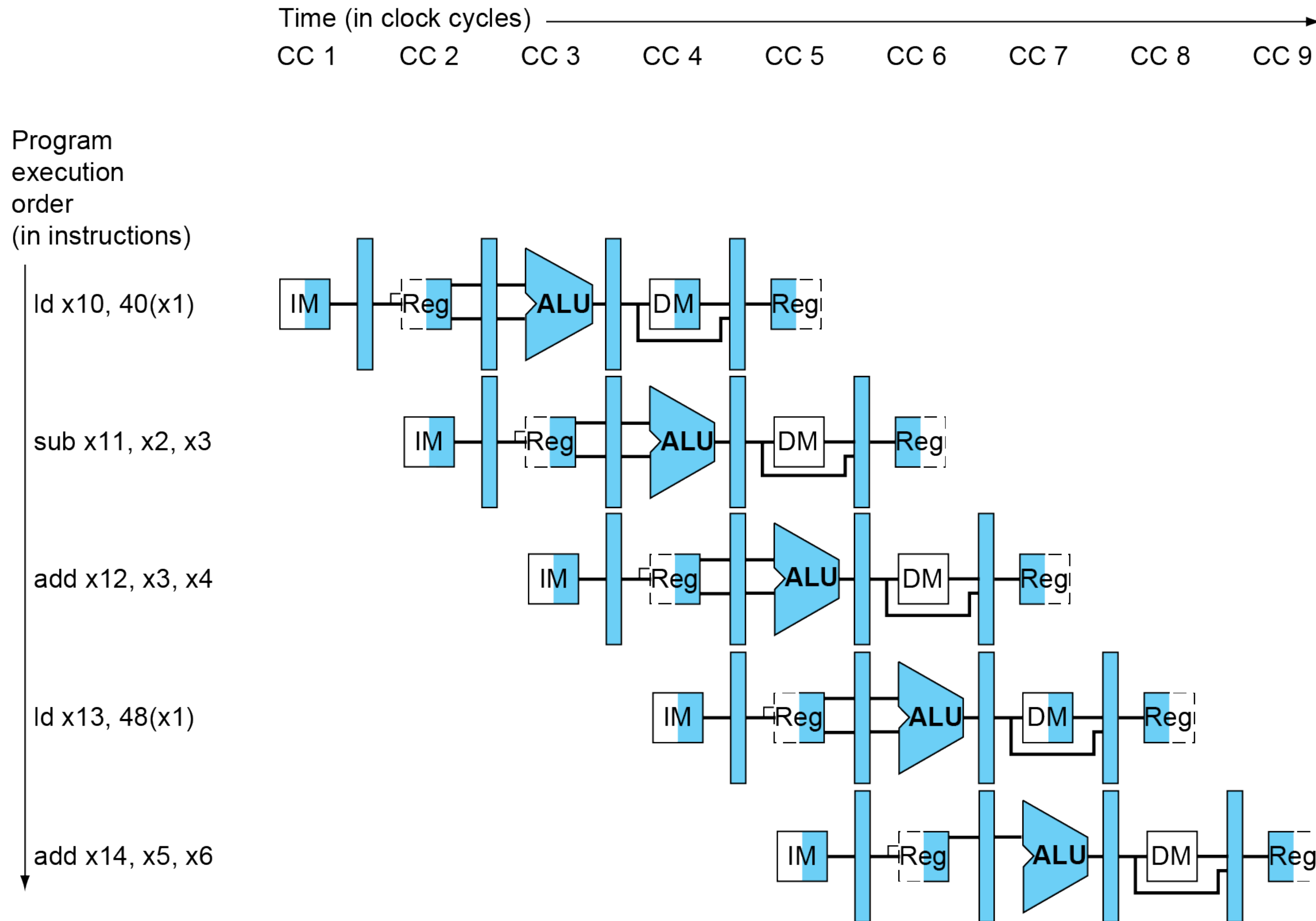


WB for Store



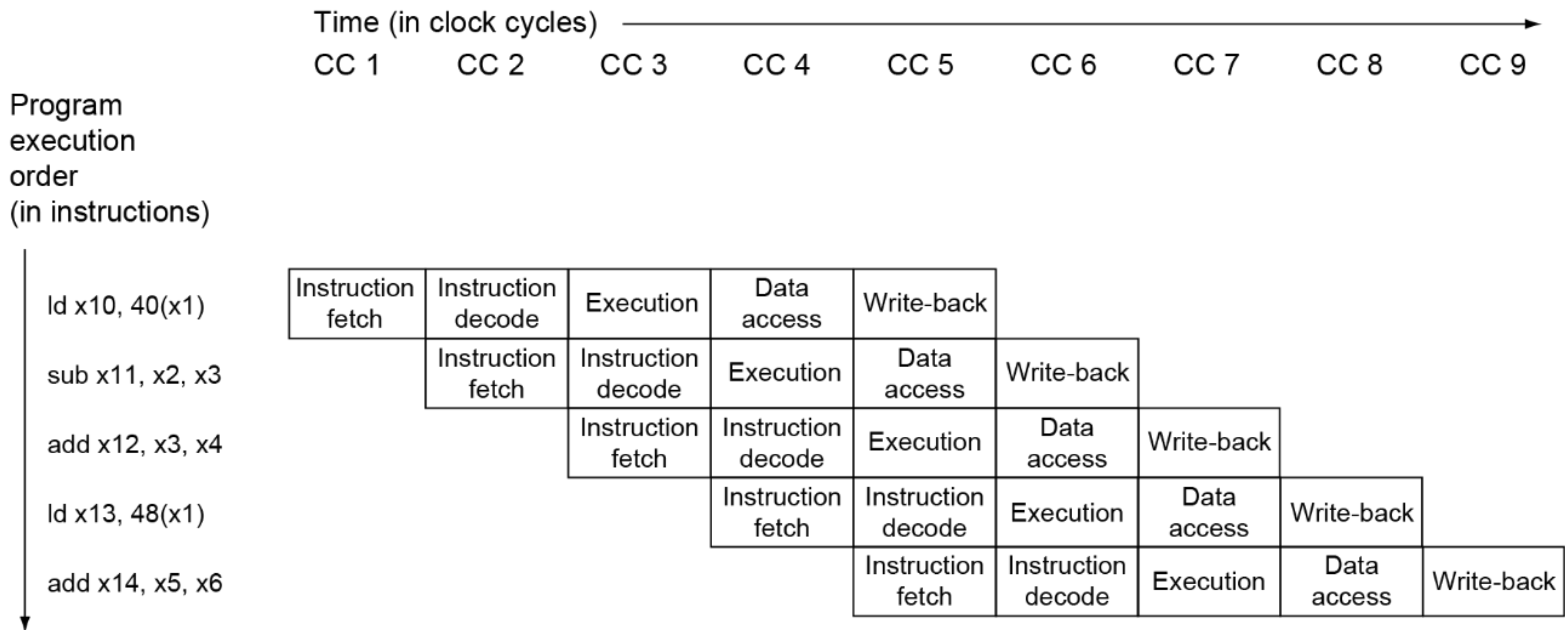
Multi-Cycle Pipeline Diagram

- Form showing resource usage



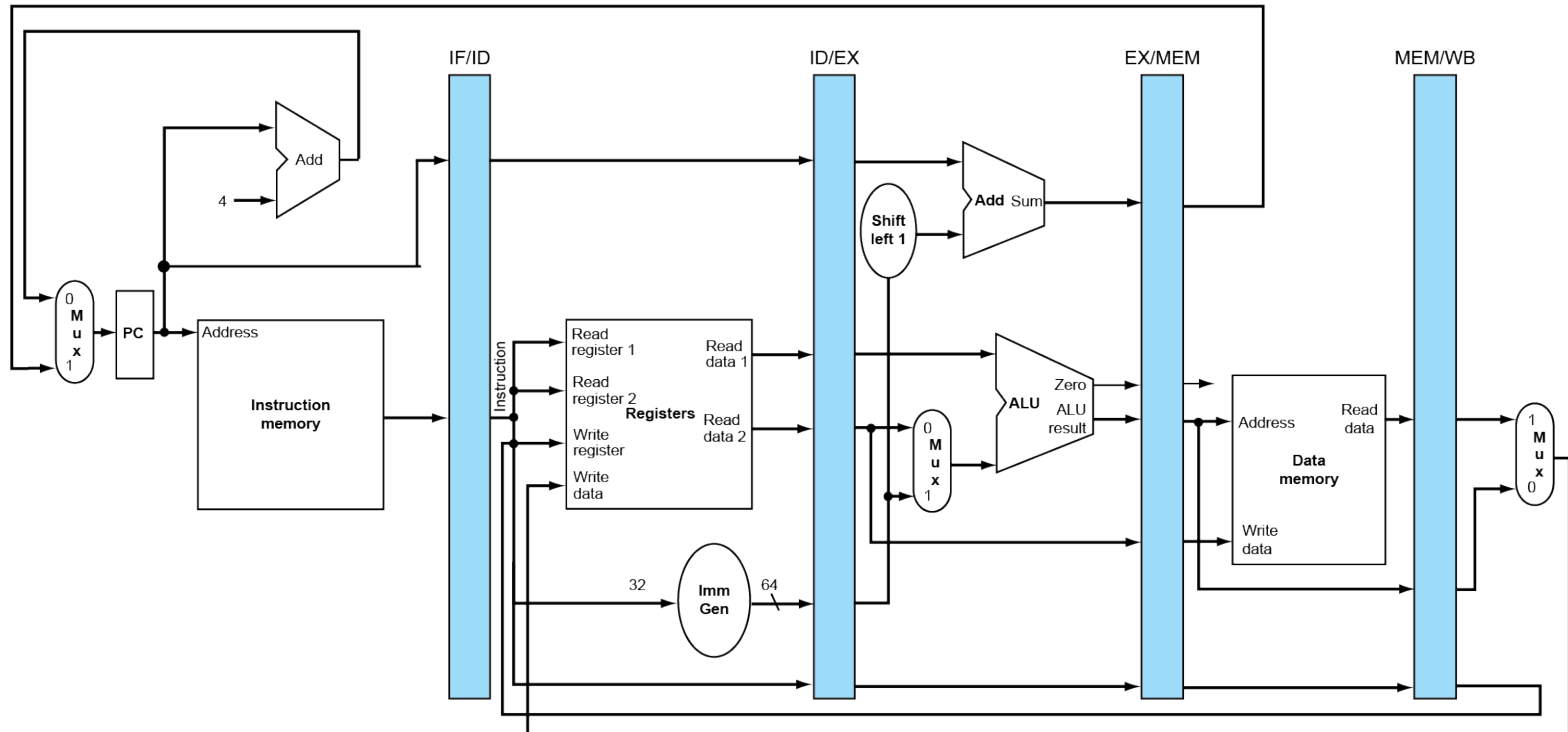
Multi-Cycle Pipeline Diagram

- Traditional form



Single-Cycle Pipeline Diagram

- State of pipeline in a given cycle

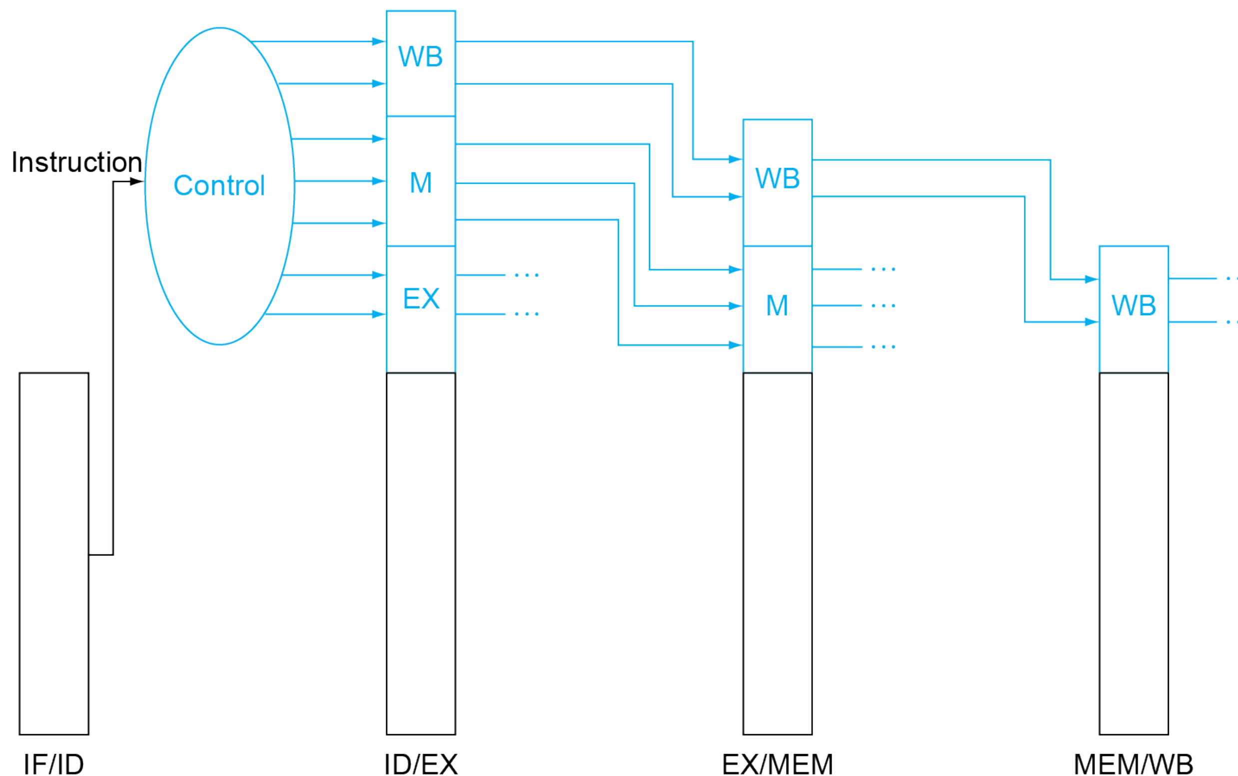


Pipelined Control (Simplified)

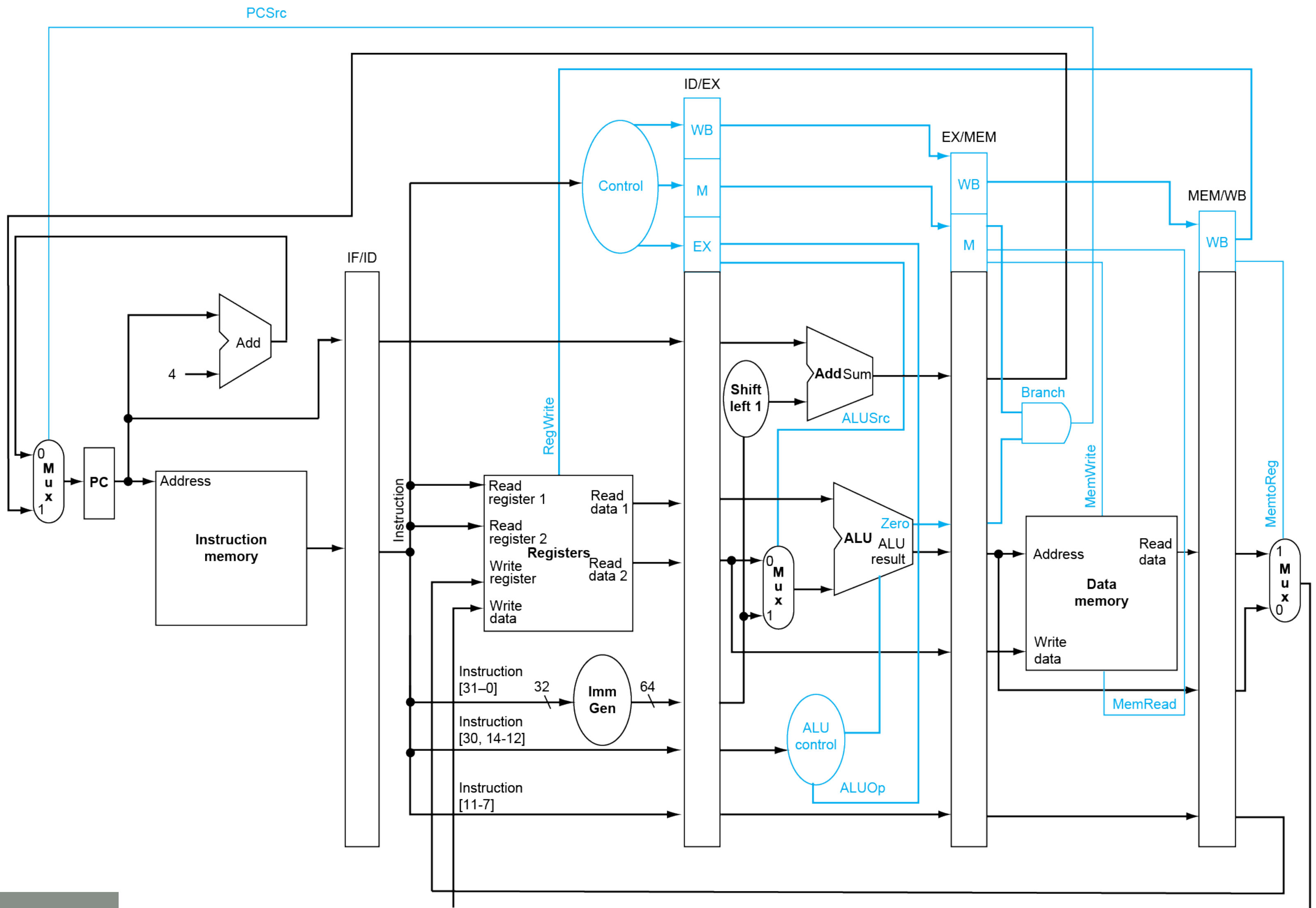
PCSrc
IF/ID
ID/EX
EX/MEM
MEM/WB
Add
4
Add Sum
Shift left 1
Branch
RegWrite
0 Mux 1
PC
Address
Instruction memory
Instruction
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
ALUSrc
0 Mux 1
Zero
Add ALU result
MemWrite
Address
Read data
Data memory
Write data
MemtoReg
1 Mux 0
Instruction [31–0]
32
Imm Gen
64
Instruction [30, 14-12]
ALU control
MemRead
Instruction [11-7]
ALUOp

Pipelined Control

- Control signals derived from instruction
  - As in single-cycle implementation



Pipelined Control



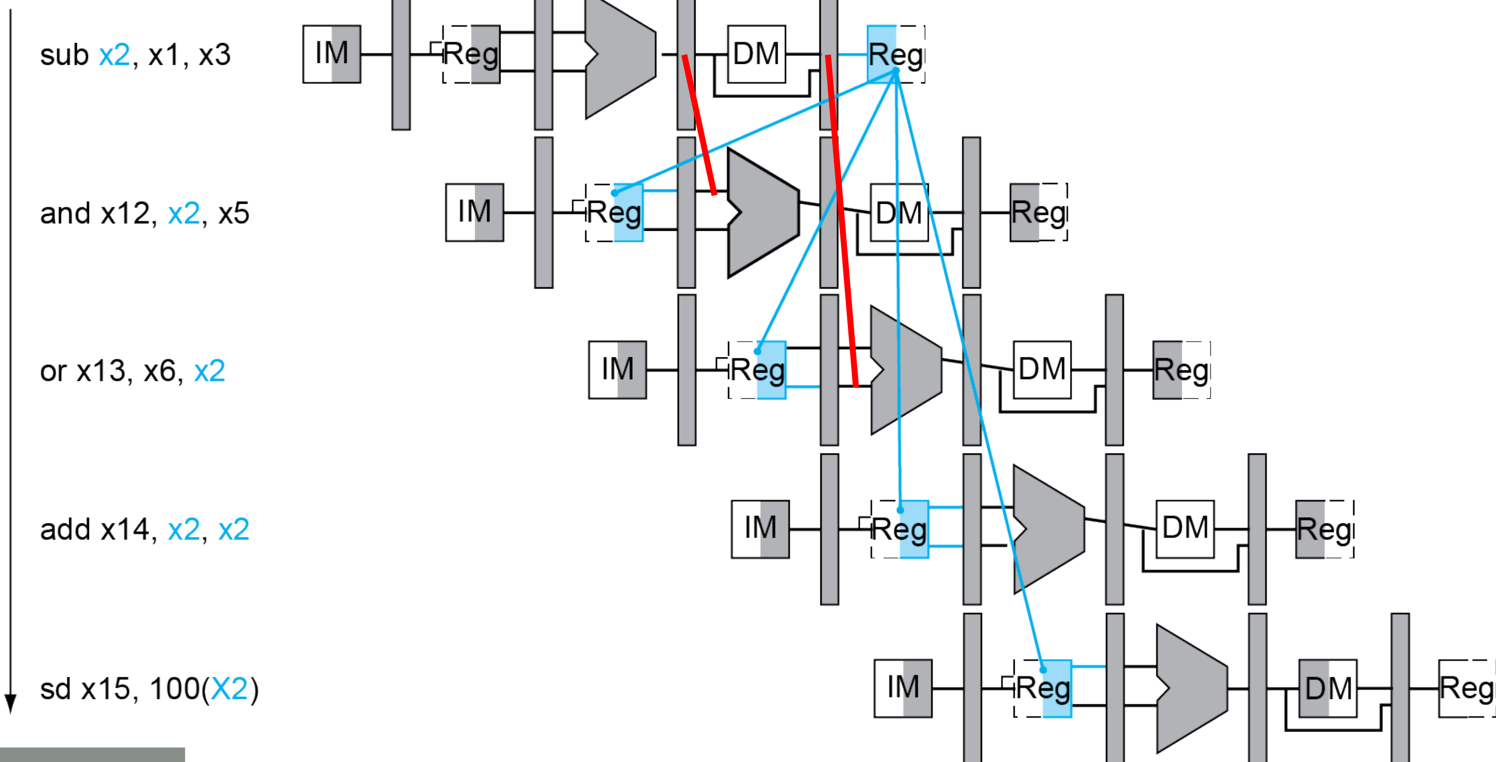
Data Hazards in ALU Instructions

- Consider this sequence:

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

- We can resolve hazards with forwarding
  - How do we detect when to forward?

Dependencies & Forwarding



Detecting the Need to Forward

- Pass register numbers along pipeline
  - e.g., ID/EX.RegisterRs1 = register number for Rs1 sitting in ID/EX pipeline register
- ALU operand register numbers in EX stage are given by
  - ID/EX.RegisterRs1, ID/EX.RegisterRs2
- Data hazards when

  1a. EX/MEM.RegisterRd = ID/EX.RegisterRs1
  1b. EX/MEM.RegisterRd = ID/EX.RegisterRs2

  Fwd from EX/MEM pipeline reg

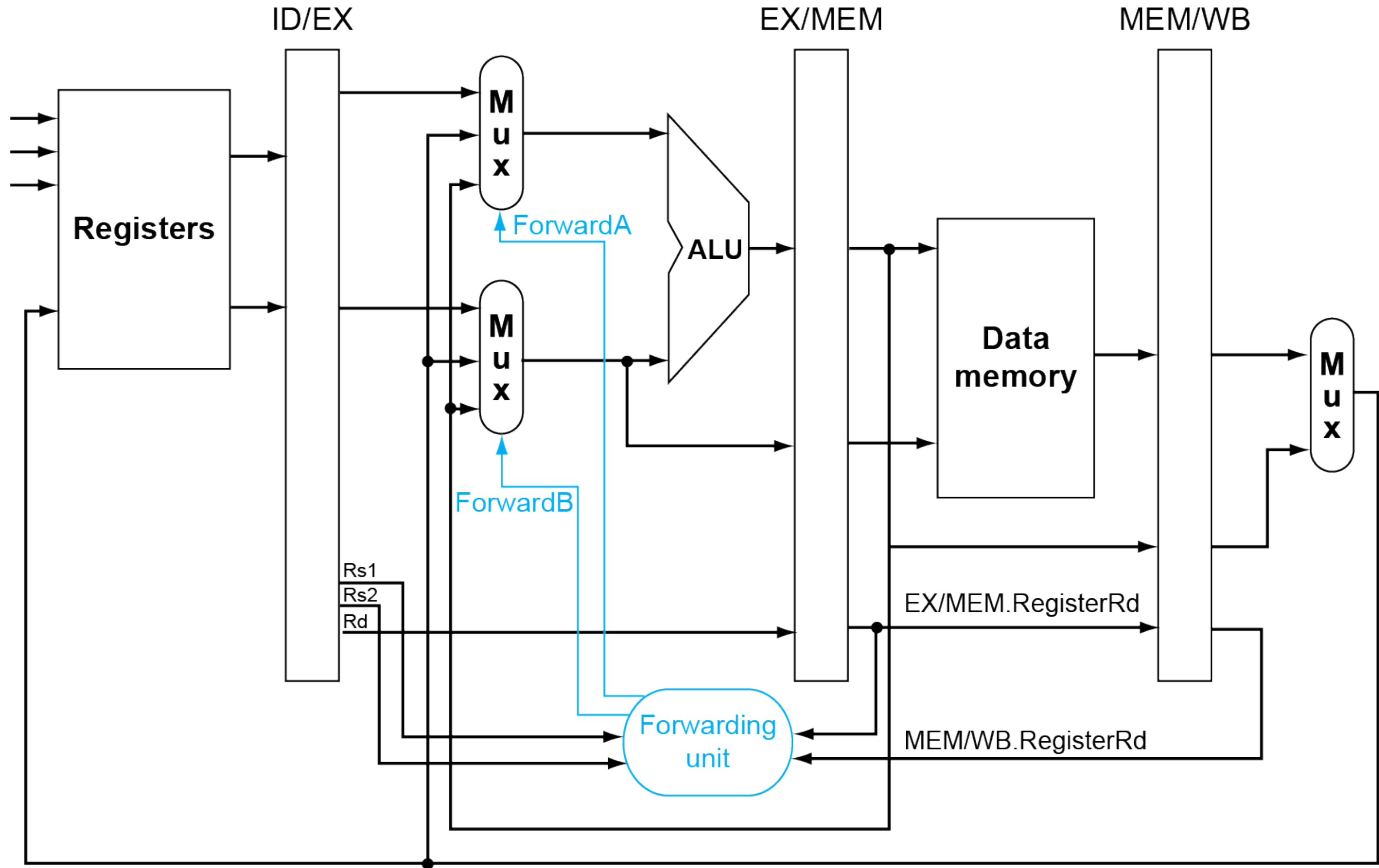
  2a. MEM/WB.RegisterRd = ID/EX.RegisterRs1
  2b. MEM/WB.RegisterRd = ID/EX.RegisterRs2

  Fwd from MEM/WB pipeline reg

Detecting the Need to Forward

- But only if forwarding instruction will write to a register!
  - EX/MEM.RegWrite, MEM/WB.RegWrite
- And only if Rd for that instruction is not x0
  - EX/MEM.RegisterRd ≠ 0, MEM/WB.RegisterRd ≠ 0

Forwarding Paths



Forwarding Conditions

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

Forwarding Conditions

- EX hazard
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs1))
    ForwardA = 10
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs2))
    ForwardB = 10
- MEM hazard
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs1))
    ForwardA = 01
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs2))
    ForwardB = 01

Double Data Hazard

- Consider the sequence:

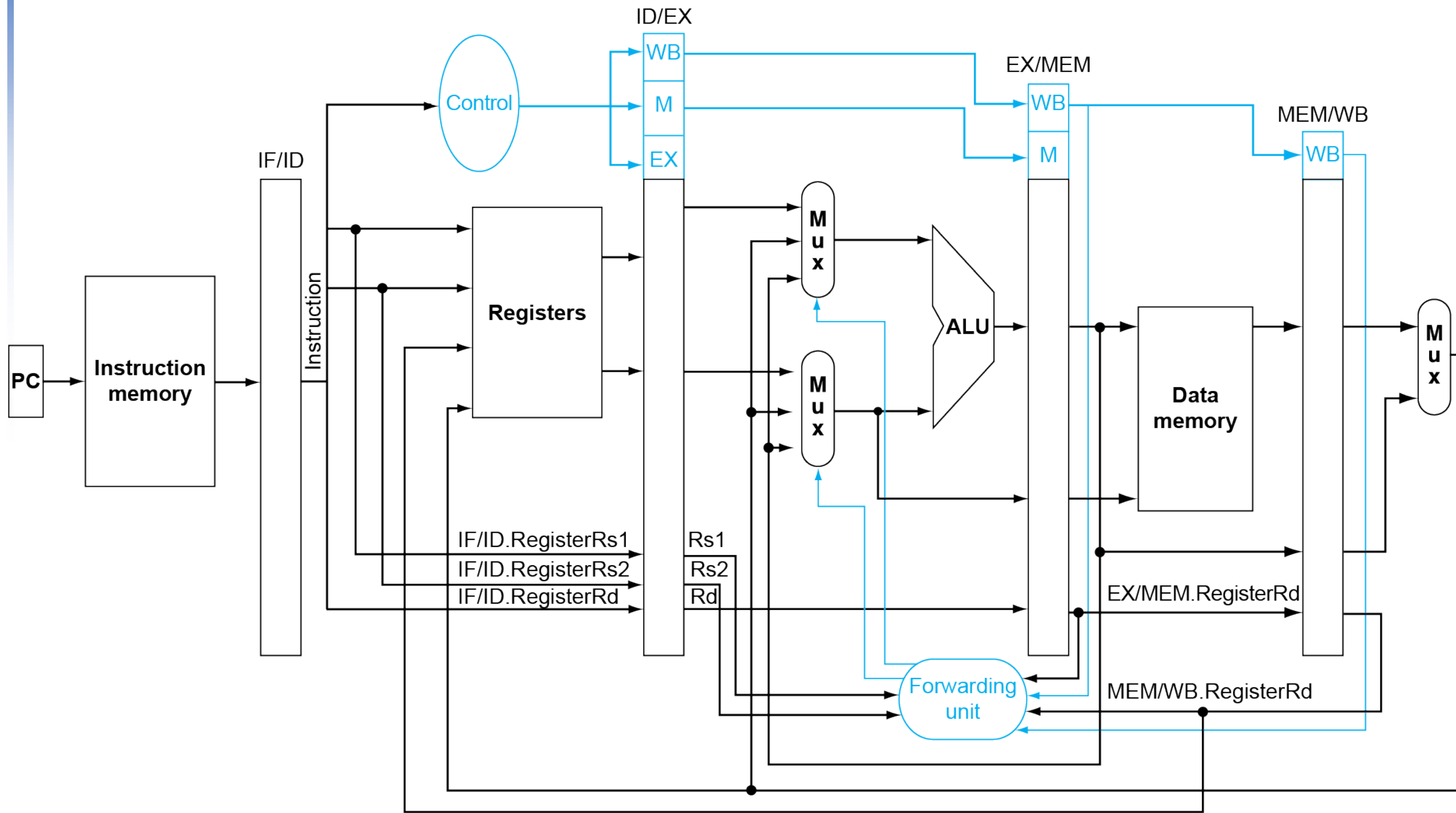
```
add x1,x1,x2
add x1,x1,x3
add x1,x1,x4
```

- Both hazards occur
  - Want to use the most recent
- Revise MEM hazard condition
  - Only fwd if EX hazard condition isn't true

Revised Forwarding Condition

- MEM hazard
  - if (MEM/WB.RegWrite
    and (MEM/WB.RegisterRd ≠ 0)
    and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs1))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs1)) ForwardA = 01
  - if (MEM/WB.RegWrite
    and (MEM/WB.RegisterRd ≠ 0)
    and not(EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs2))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs2)) ForwardB = 01

Datapath with Forwarding



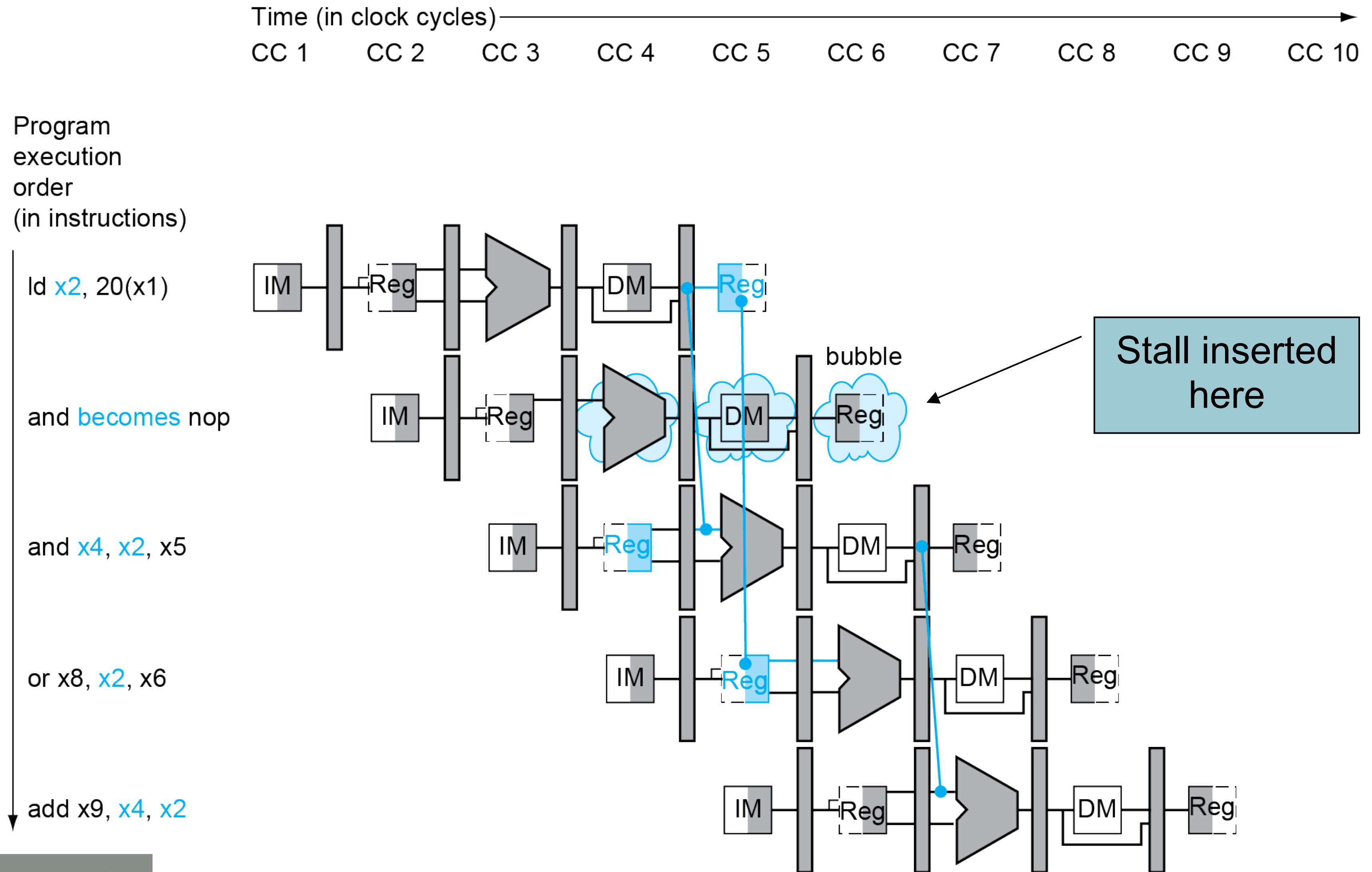
Load-Use Hazard Detection

- Check when using instruction is decoded in ID stage
- ALU operand register numbers in ID stage are given by
  - IF/ID.RegisterRs1, IF/ID.RegisterRs2
- Load-use hazard when
  - ID/EX.MemRead and (ID/EX.RegisterRd ≠ 0) and ((ID/EX.RegisterRd = IF/ID.RegisterRs1) or (ID/EX.RegisterRd = IF/ID.RegisterRs2))
- If detected, stall and insert bubble

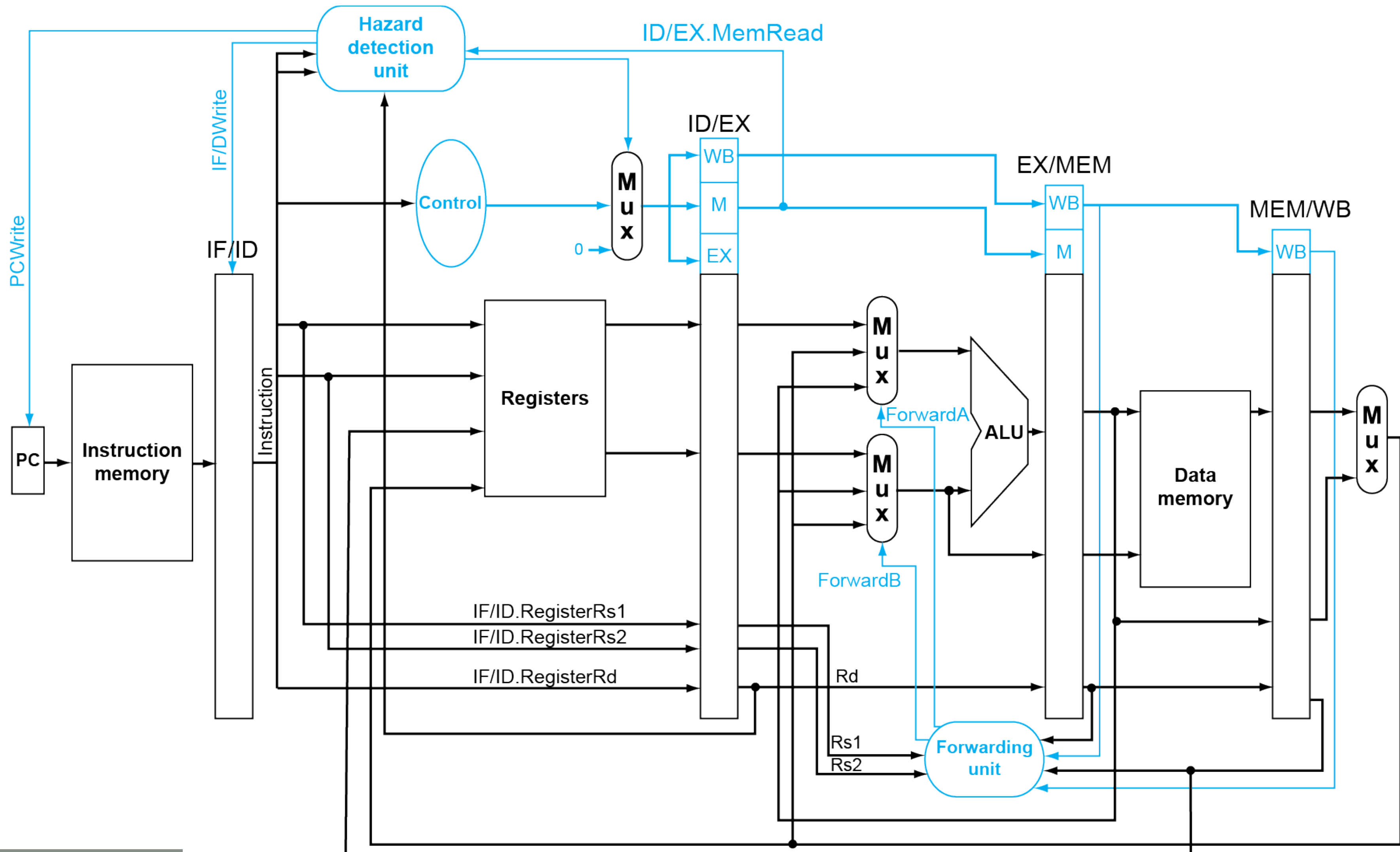
How to Stall the Pipeline

- Force control values in ID/EX register to 0
  - EX, MEM and WB do nop (no-operation)
- Prevent update of PC and IF/ID register
  - Using instruction is decoded again
  - Following instruction is fetched again
  - 1-cycle stall allows MEM to read data for `ld`
    - Can subsequently forward to EX stage

Load-Use Data Hazard



Datapath with Hazard Detection



Stalls and Performance

The BIG Picture

- Stalls reduce performance
  - But are required to get correct results
- Compiler can arrange code to avoid hazards and stalls
  - Requires knowledge of the pipeline structure

Appendix A.5:

Constructing a Basic Arithmetic Logic Unit (ALU)

Dr. Gheith Abandah

[Adapted from the slides of Professor Mary Irwin (www.cse.psu.edu/~mji) which in turn Adapted from *Computer Organization and Design*,

Patterson & Hennessy]

MIPS Number Representations

- 64-bit signed numbers (2's complement):

```
0000 0000 0000 0000 0000 0000 0000 0000two = 0ten
0000 0000 0000 0000 0000 0000 0000 0001two = + 1ten
...

0111 1111 1111 1111 1111 1111 1111 1110two = + 9,223,372,036,854,775,806ten
0111 1111 1111 1111 1111 1111 1111 1111two = + 9,223,372,036,854,775,807ten
1000 0000 0000 0000 0000 0000 0000 0000two = – 9,223,372,036,854,775,808ten
1000 0000 0000 0000 0000 0000 0000 0001two = – 9,223,372,036,854,775,807ten
...

1111 1111 1111 1111 1111 1111 1111 1110two = – 2ten
1111 1111 1111 1111 1111 1111 1111 1111two = – 1ten
```

maxint

minint

MSB

LSB

- Converting <64-bit values into 64-bit values
  - copy the most significant bit (the sign bit) into the "empty" bits

```
0010 -> 0000 0010
1010 -> 1111 1010
```

  - sign extend versus zero extend (`lb` vs. `lbu`)

RISC-V Arithmetic Logic Unit (ALU)

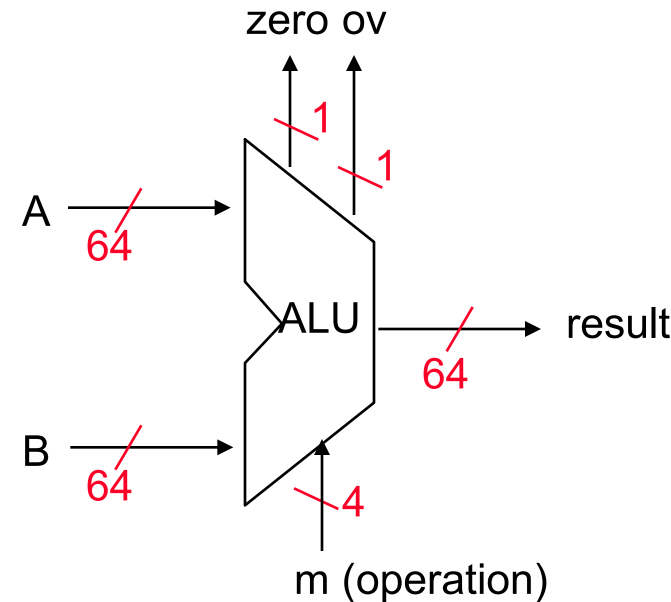
- Must support the Arithmetic/Logic operations of the ISA

  `add, addi`

  `sub`

  `and, andi, or, ori, xor, xori`

  `beq, bne, slt, slti, sltiu, sltu`



- With special handling for
  - sign extend – `addi, andi, ori, xori`
  - zero extend – `lbu`
- RISC-V world is 64-bit wide → 64-bit wide ALU
- First, generate 1-bit ALU ***slice*** then replicate 64 times
- ALU is constructed from: AND, OR, inverters, MUXes

Review: 2's Complement Binary Representation

- Negate

1011

and add a 1

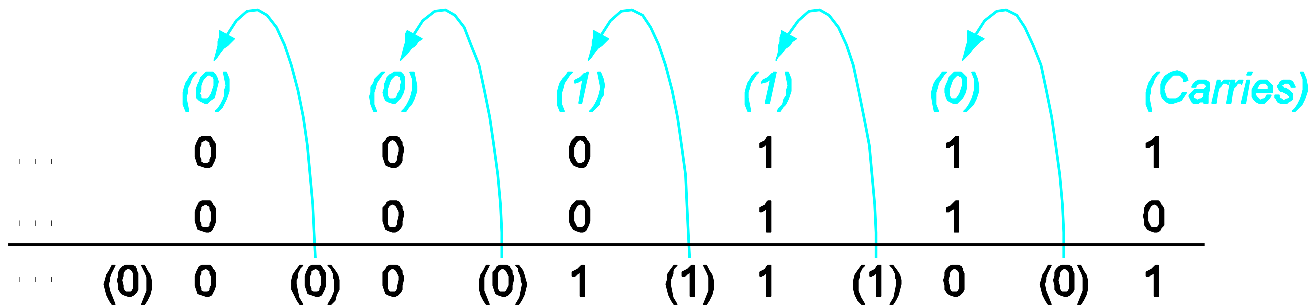
1010

complement all the bits

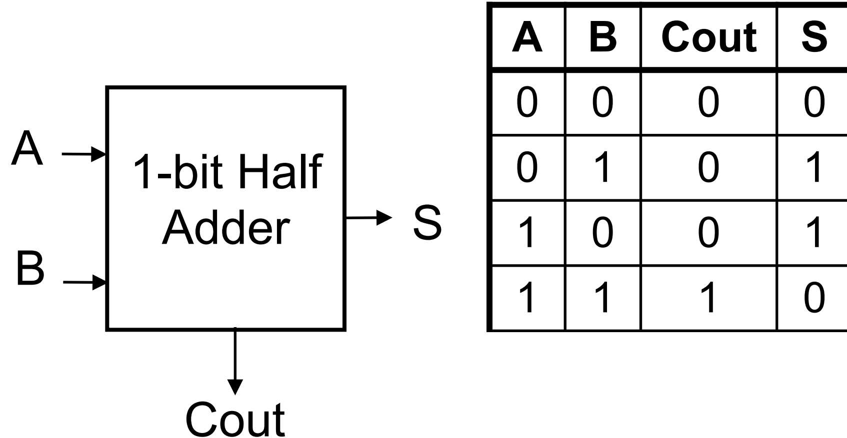
- Note: negate and invert are different!

	2'sc binary	decimal
-2^3 =	1000	-8
$-(2^3 - 1)$ =	1001	-7
	1010	-6
	1011	-5
	1100	-4
	1101	-3
	1110	-2
	1111	-1
	0000	0
	0001	1
	0010	2
	0011	3
	0100	4
	0101	5
	0110	6
$2^3 - 1$ =	0111	7

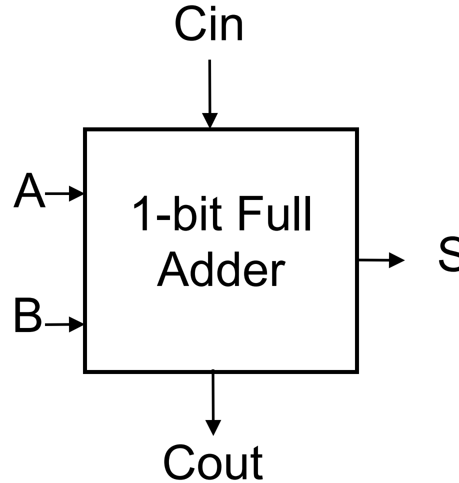
Binary Addition



Review: Half (2,2) Adder and Full (3,2) Adder



A	B	Cout	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



A	B	Cin	Cout	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$S = A \oplus B$ (odd parity function)

Cout = A&B (majority function)

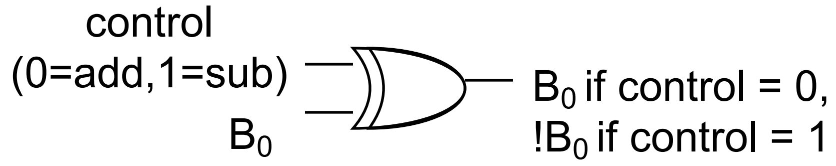
$S = A \oplus B \oplus Cin$ (odd parity function)

Cout = A&B | A&Cin | B&Cin (majority function)

- How can we use it to build a 64-bit adder?
- How can we modify it easily to build an adder/subtractor?

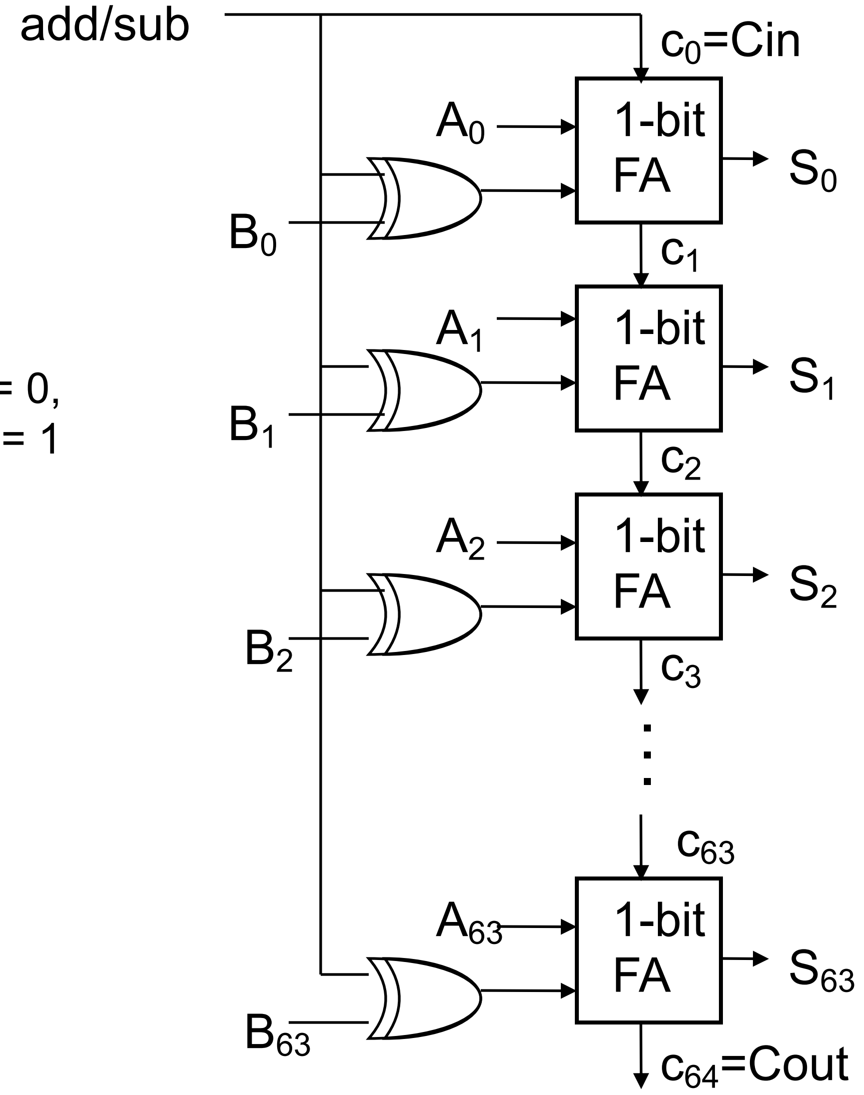
A 64-bit Ripple Carry Adder/Subtractor

- Remember 2's complement is just
  - complement all the bits



  - add a 1 in the least significant bit

```
A     0111   →     0111
B   - 0110   →   + 1001
    ------             1
      0001         -----
                 1 0001
```

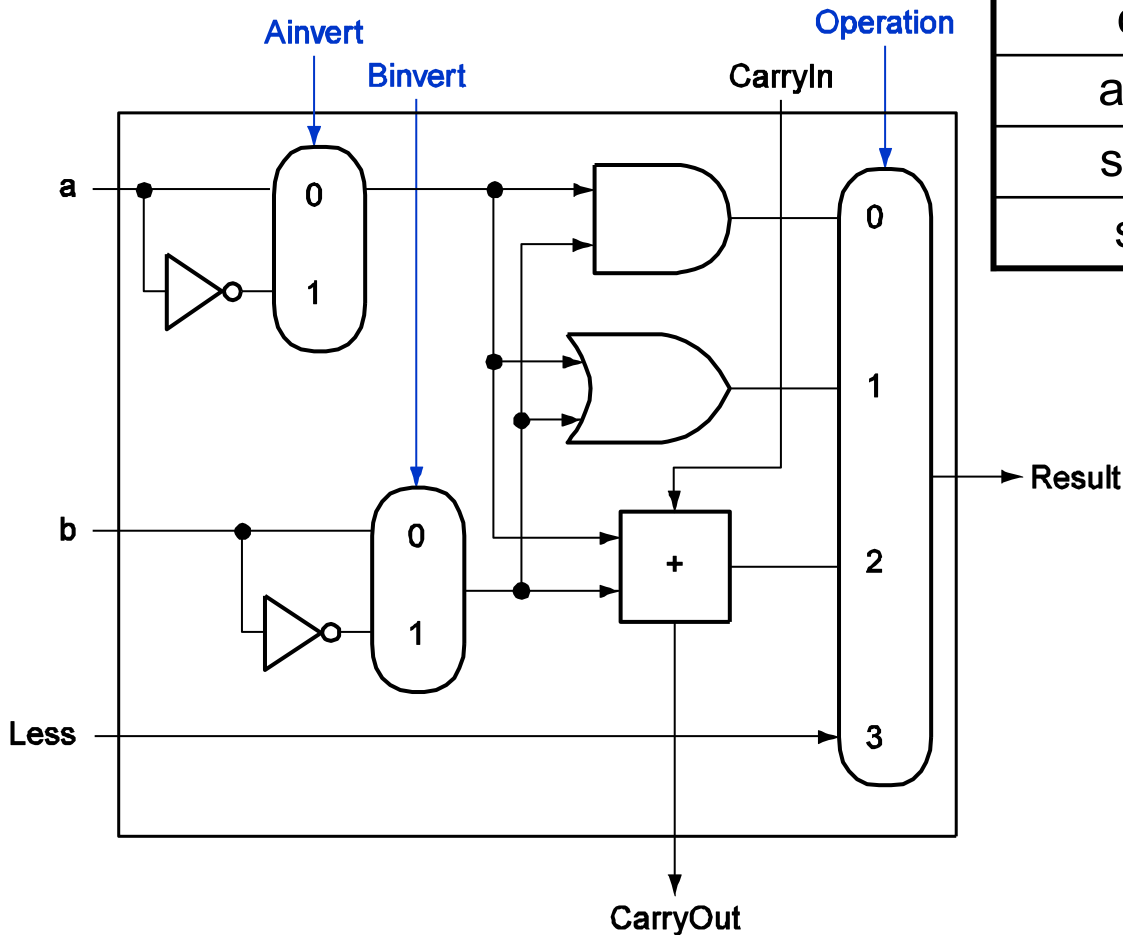


Tailoring the ALU to the RISC-V ISA

- Need to support the logic operations (`and`, `or`, `xor`)
  - Bit wise operations (no carry operation involved)
  - Need a logic gate for each function, mux to choose the output
- Need to support the set-on-less-than instruction (`slt`)
  - Use subtraction to determine if (a – b) < 0 (implies a < b)
  - Copy the sign bit into the low order bit of the result, set remaining result bits to 0
- Need to support test for equality (`bne`, `beq`)
  - Again use subtraction: (a - b) = 0 implies a = b
  - Additional logic to “nor” all result bits together
- Immediates are sign extended outside the ALU with wiring (i.e., no logic needed)

RISC-V ALU

- Least-significant bits



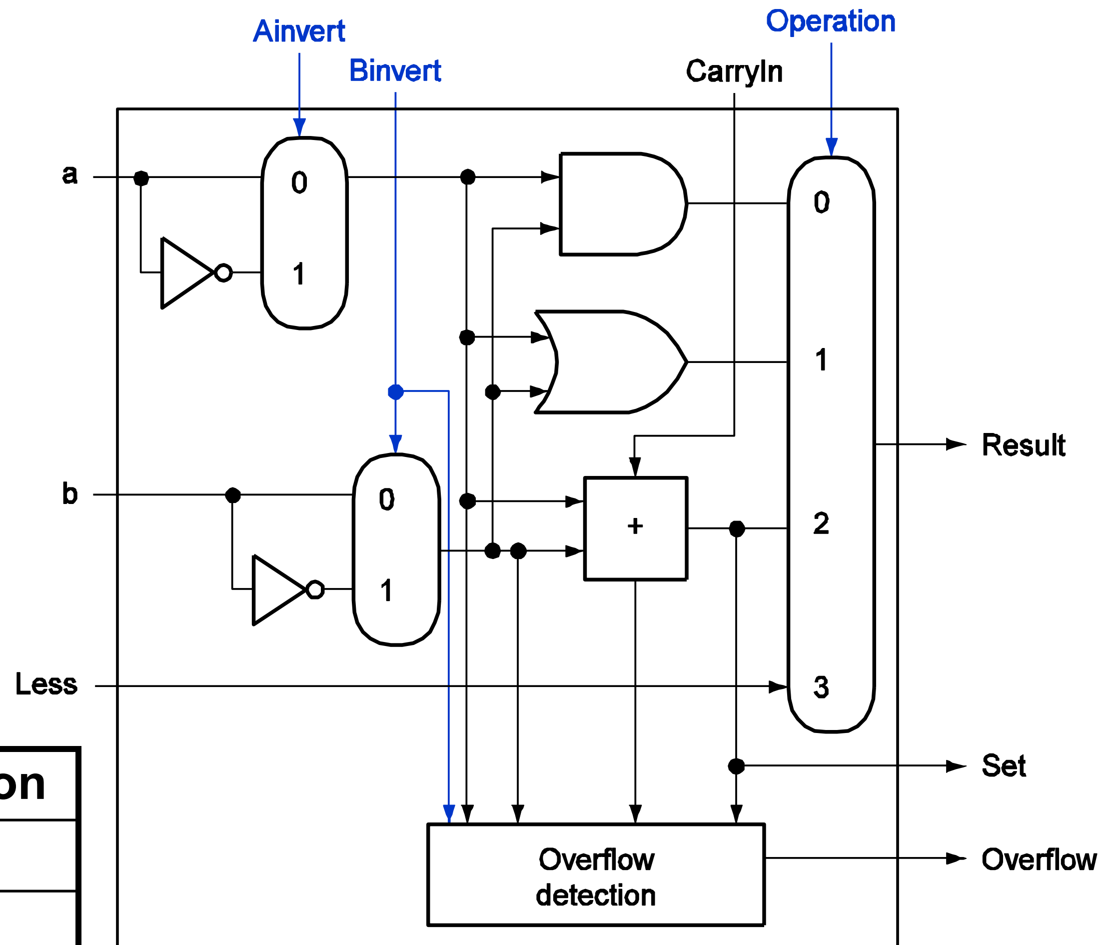
Function	Binvert	Operation
and	0	00
or	0	01
add	0	10
sub	1	10
slt	1	11

Set if less than (slt)

- slt rd, rs1, rs2
  - If rs1 < rs2 → rd = (0000000……………….0001)$_{two}$
  - If rs1 >= rs2 → rd = (0000000……………….0000)$_{two}$
- For ALU_1 to ALU_{63}, connect the ***Less*** input to ground or zero
- Use subtraction to determine if rs1 is less than rs2
  - rs1 –rs2 is negative → ***adder/subtractor output in ALU_{63} is 1***
  - The adder/subtractor output of ALU_{63} is called ***Set*** and is connected to the ***Less*** input of ALU_0
- Since we need a special ALU slice for the most significant digit to generate the ***Set*** output, we add the functionality needed to generate the overflow detection logic since it is associated with that bit too

RISC-V ALU

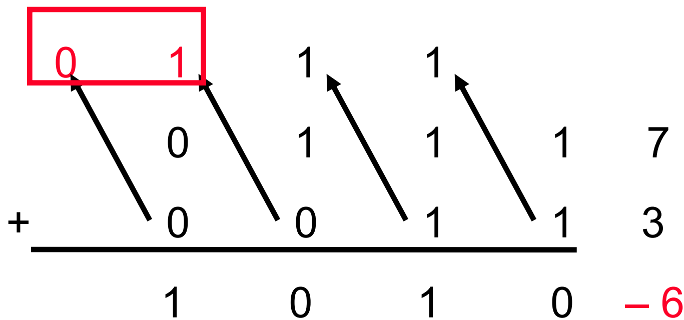
- Most-significant bit

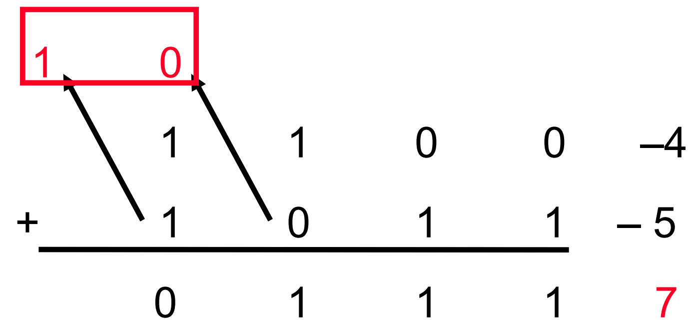


Function	Binvert	Operation
and	0	00
or	0	01
add	0	10
sub	1	10
slt	1	11

Overflow Detection

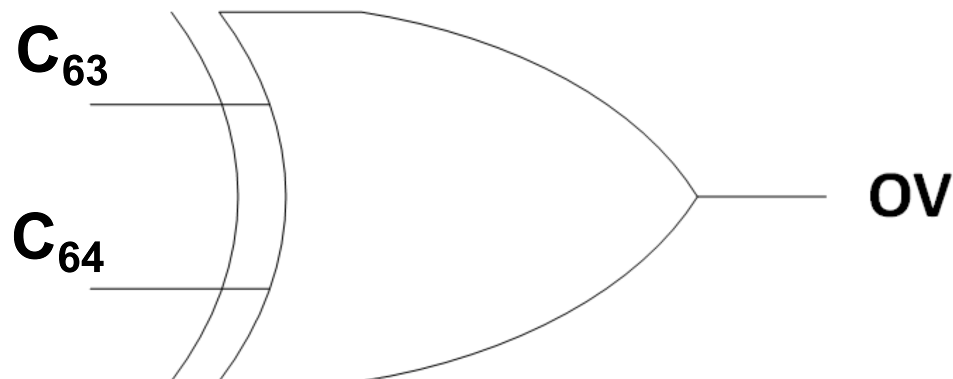
- Overflow: the result is too large to represent in 64 bits
- Overflow occurs when
  - adding two positives yields a negative
  - or, adding two negatives gives a positive
  - or, subtract a negative from a positive gives a negative
  - or, subtract a positive from a negative gives a positive
- On your own: Prove you can detect overflow by:
  - Carry *into* MSB ***XOR*** Carry *out of* MSB, ex for 4 bit signed numbers





Overflow Detection

A_{63}	B_{63}	C_{63} (Cin)	C_{64} (Cout)	S_{63}	Overflow (OV)
0	0	0	0	0	0
0	0	1	0	1	1
0	1	0	0	1	0
0	1	1	1	0	0
1	0	0	0	1	0
1	0	1	1	0	0
1	1	0	1	0	1
1	1	1	1	1	0

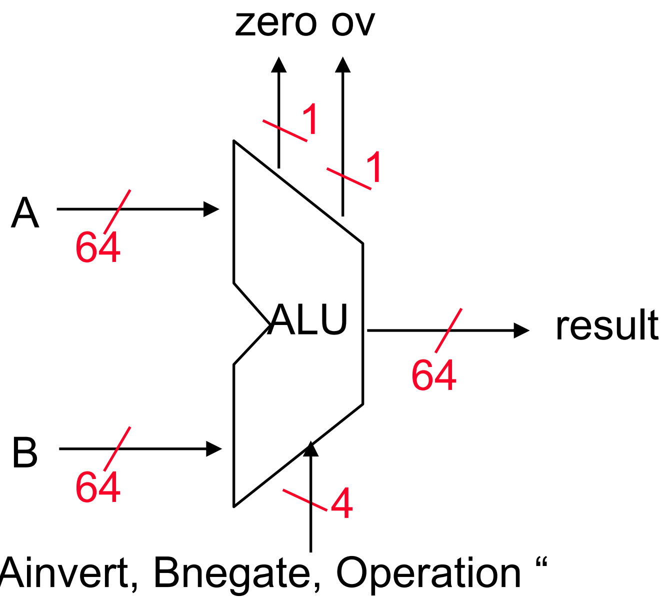


RISC-V ALU

- For sub, slt, and branch: Binvert = 1 and C_0 = 1
- For add: Binvert = 0 and C_0 = 0
- For logical except NOR: Binvert = 0 and C_0 = X
- For NOR: Binvert = 1 and C_0 = X
- **Based on above: Binvert and C_0 are merged together into once signal called Bnegate**

Ainvert	Bnegate	Operation	
0	0	00	AND
0	0	01	OR
0	0	10	ADD
0	1	10	SUB/Branch
0	1	11	SLT
1	1	00	NOR

RISC-V ALU



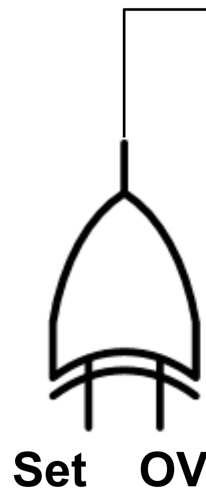
Bnegate
Operation
Ainvert
a0
b0
CarryIn
ALU0
Less
CarryOut
Result0
a1
b1
0
CarryIn
ALU1
Less
CarryOut
Result1
Zero
a2
b2
0
CarryIn
ALU2
Less
CarryOut
Result2
CarryIn
a63
b63
0
CarryIn
ALU63
Less
Result63
Set
Overflow

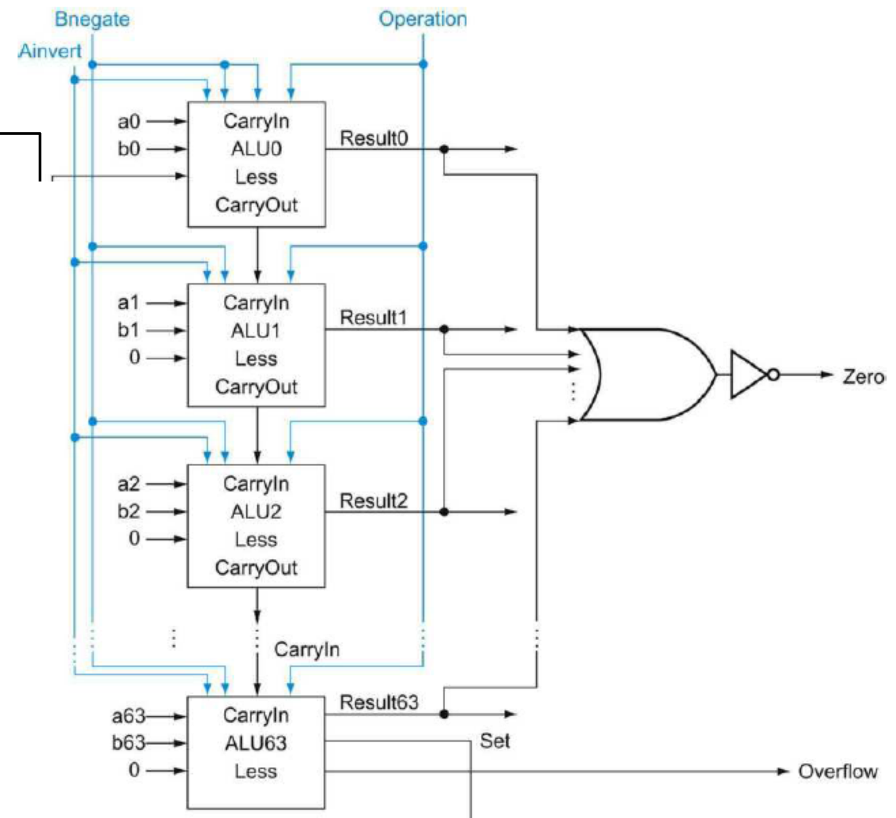
Set if less than correction

- Previously we connected ***$Less_0$*** to ***Set*** which is only correct if there is no **overflow**
- From the truth table below, we deduce that:

$$Less_0 = Set \oplus OV$$

Set	OV	$Less_0$
0	0	0
0	1	1
1	0	1
1	1	0



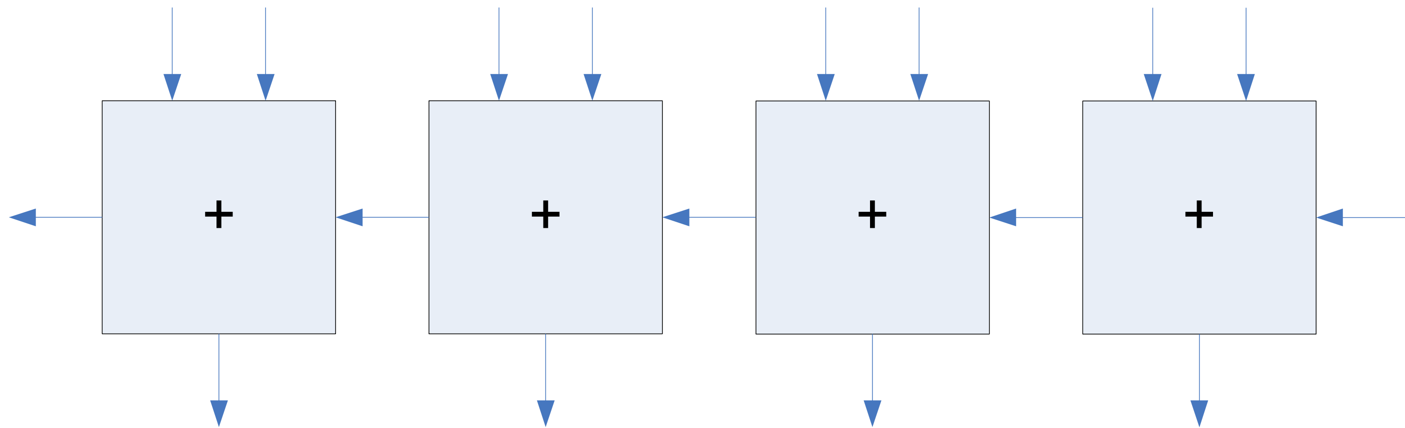


Appendix A.6:

Faster Addition: Carry Lookahead

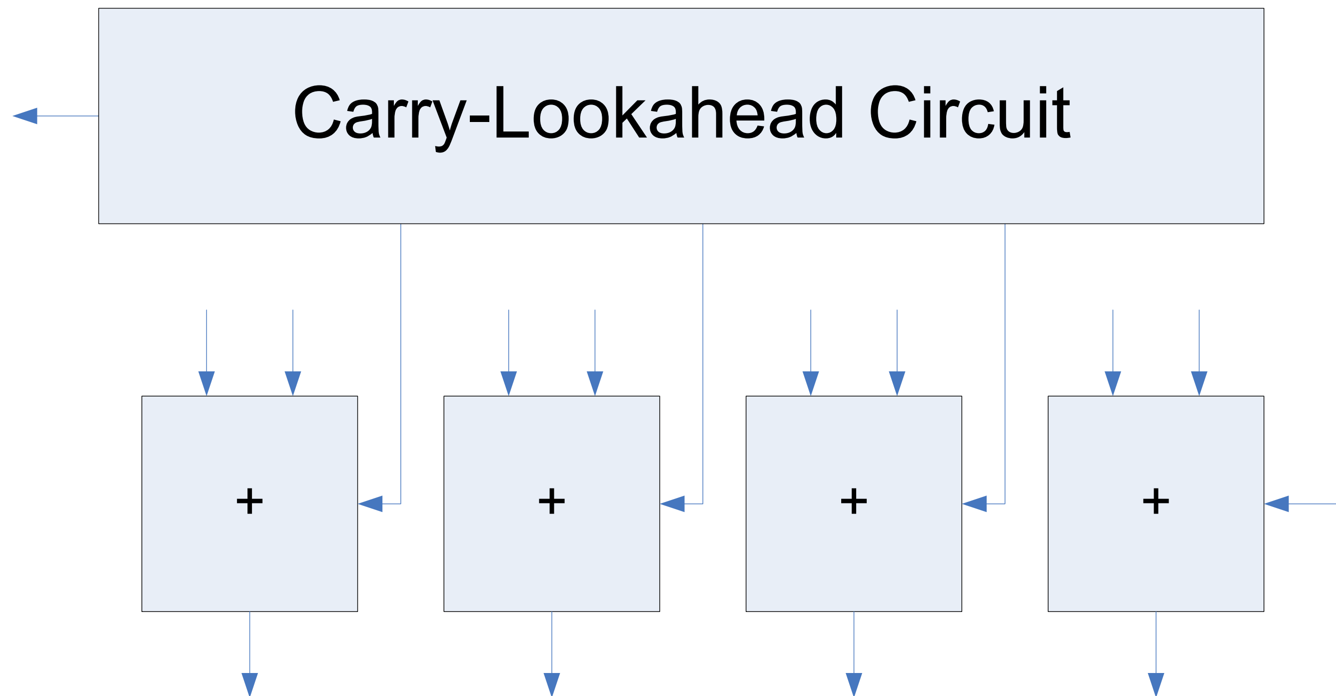
Improving Addition Performance

- The ripple-carry adder is slow



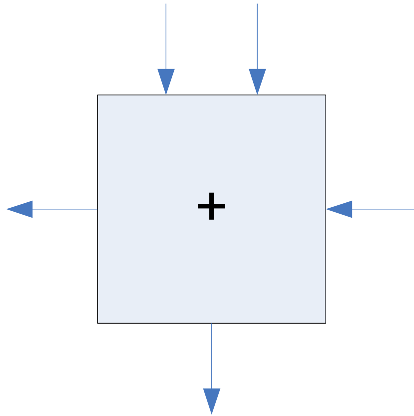
Carry-Lookahead Adder

- Need fast way to find the carry



Carry-Lookahead Adder

- Carry generate and carry propagate



a_i	b_i	g_i	p_i
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

- $g_i = a_i \,.\, b_i$
- $p_i = a_i + b_i$

Carry-Lookahead Adder

Carry Equations:

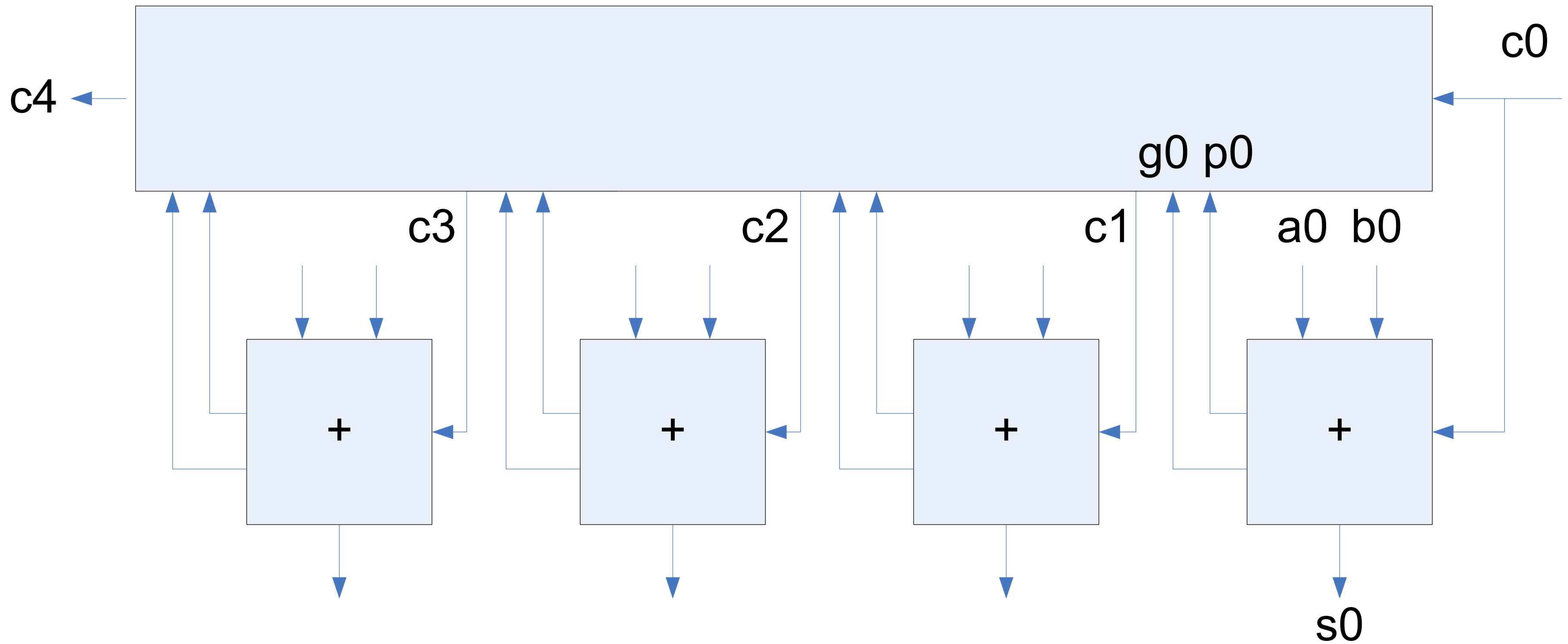
```
c1 = g0 + p0c0

c2 = g1 + p1c1
   = g1 + p1g0 + p1p0c0

c3 = g2 + p2c2
   = g2 + p2g1 + p2p1g0 + p2p1p0c0

c4 = g3 + p3c3
   = g3 + p3g2 + p3p2g1 + p3p2p1g0 + p3p2p1p0c0
```

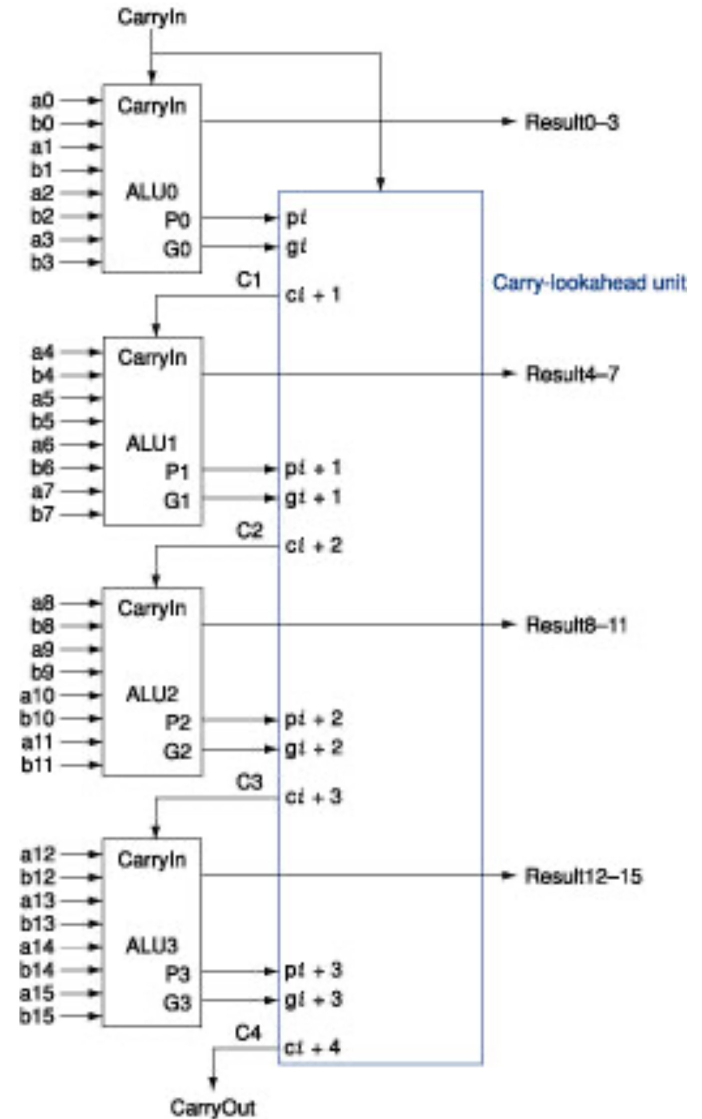
4-bit Carry-Lookahead Adder



Larger Carry-Lookahead Adders

$$P = p0p1p2p3$$

$$G = g3 + g2p3 + g1p2p3 + g0p1p2p3$$



Appendix C: Multi-Cycle Datapath and Control

Dr. Iyad F. Jafar*

Adapted from Dr. Gheith Abandah slides
http://www.abandah.com/gheith/Courses/CPE335_S08/index.html

*Slightly updated by Dr. Waleed Dweik

Outline

- Introduction
- Multi-cycle Datapath
- Multi-cycle Control
- Performance Evaluation

Introduction

- The single-cycle datapath is straightforward, but…
  - **Hardware duplication**
    - It has to use one ALU and two 64-bit adders
    - It has separate Instruction and Data memories
  - **Cycle time** is determined by worst-case path! Time is wasted for instructions that finish earlier!!
- **Can we do any better?**
  - Break the instruction execution into steps
  - Each step finishes in one shorter cycle
  - Since instructions differ in number of steps, so will the number of cycles! Thus, time is different!
  - **Multi-Cycle implementation!**

Multi-Cycle Datapath

- Instruction execution is done over multiple steps such that
  - Each step takes one cycle
  - The amount of work done per cycle is balanced
  - Restrict each cycle to use one major functional unit
- **Expected benefits**
  - Time to execute different instructions will be different (Better Performance!)
  - The cycle time is smaller (faster clock rate!)
  - Allows functional units to be used more than once per instruction as long as they are used in different cycles
    - One memory is needed!
    - One ALU is needed!

Multi-Cycle Datapath

- **Requirements**
  - Keep in mind that we have one ALU, Memory, and PC
  - **Thus,**
    - Add/expand multiplexors at the inputs of major units that are used differently across instructions
    - Add intermediate registers to hold values between cycles !!
    - Define additional control signals and redesign the control unit

Multi-Cycle Datapath

- **Requirements - ALU**
  - Operations
    - Compute PC+4
    - Compute the Branch Address
    - Compare two registers
    - Perform ALU operations
    - Compute memory address
  - Thus, the first ALU input could be
    - R[rs] (R-type)
    - PC (PC = PC + 4)

    ➔ Add a MUX and define the ALUScrA signal
  - The second ALU input could be
    - R[rt] (R-type)
    - A constant value of 4 (to compute PC + 4)
    - Sign-extended immediate (to compute address of LW and SW)
    - Sign-extended immediate x 2 (compute branch address for BEQ)

    ➔ Expand the MUX at the second ALU input and make the ALUSrcB signal two bits
  - The values read from register file will be used in the next cycle

    ➔ Add the A and B registers
  - The ALU result (R-type result or memory address) will be used in the following cycle

    ➔ Add the ALUOut register

Multi-Cycle Datapath

- **Requirements - PC**
  - PC input could be
    - PC + 4 (sequential execution)
    - Branch address
  - The PC is not written on every cycle

    ➔ Define the PCWrite singal

Multi-Cycle Datapath

- **Requirements – Memory**
  - Memory input could be
    - Memory address from PC
    - Memory address from ALU

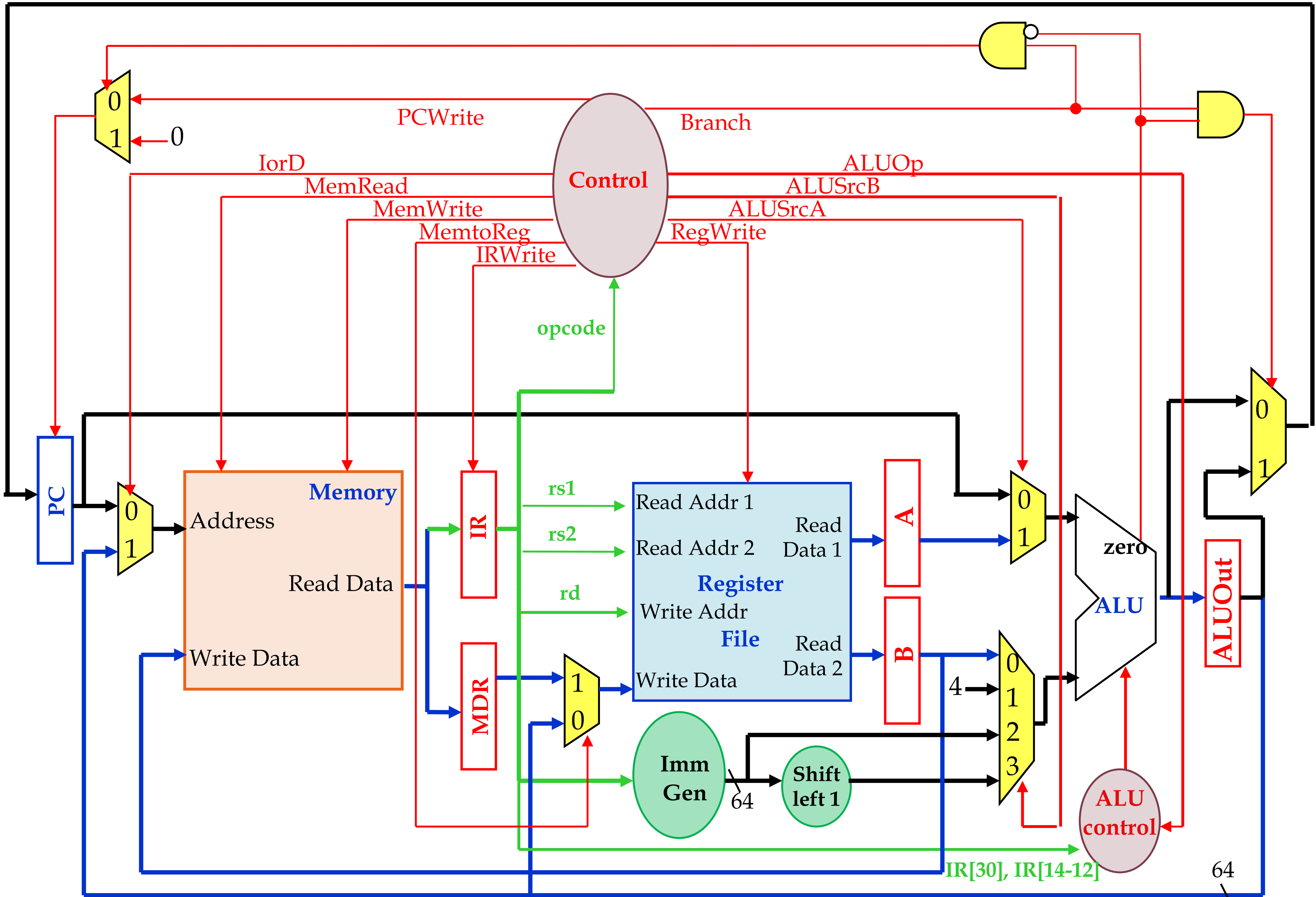
    ➔ Add MUX at the address port of the memory and define the IorD signal
  - Memory output could be
    - Instruction
    - Data

    ➔ Add the IR register to hold the instruction

    ➔ Add the MDR register to hold the data loaded from memory (Load)
  - The IR is not written on every cycle

    ➔ Define the IRWrite signal

Multi-Cycle Datapath



Multi-Cycle Control Signals

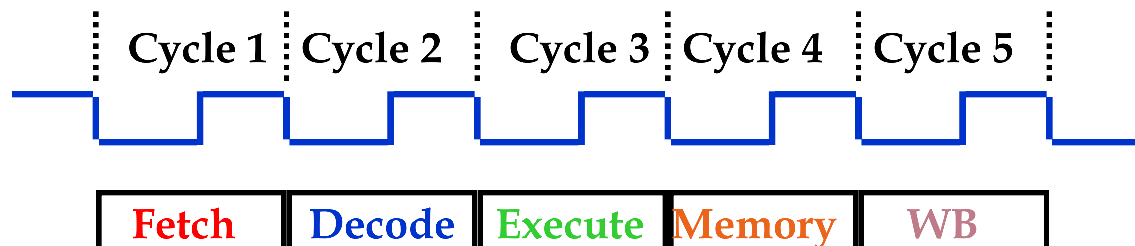
Signal Name	Effect when Deasserted (0)	Effect when Asserted (1)
RegWrite	None	Write is enabled to selected destination register
ALUSrcA	The first ALU operand is the PC	The first ALU operand is register A
MemRead	None	Content of memory address is placed on Memory data out
MemWrtite	None	Memory location specified by the address is replaced by the value on Write data input
MemtoReg	The value fed to register file is from ALUOut	The value fed to register file is from memory
IorD	PC is used as an address to memory unit	ALUOut is used to supply the address to the memory unit
IRWrite	None	The output of memory is written into IR
PCWrite	None	PC is written
Branch	Output of ALU (PC +4) is sent to the PC for writing	if Zero output from ALU is also active, The content of ALUOut are sent to the PC for writing

Multi-Cycle Control Signals

Signal	Value	Effect
ALUOp	**00**	ALU performs add operation
	01	ALU performs subtract operation
	10	The funct field of the instruction determines the ALU operation
ALUSrcB	**00**	The second input to the ALU comes from register B
	01	The second input to the ALU is 4 (to increment PC)
	10	The second input to the ALU is the sign extended immediate
	11	The second input to the ALU is the sign extended immediate shifted left by one bit

Instruction Execution

- The execution of instructions is broken into multiple cycles
- In each cycle, only one major unit is allowed to be used
- The major units are
  - The ALU
  - The Memory
  - The Register File
- Keep in mind that not all instructions use all the major functional units
- In general we may need up to five cycles



Instruction Execution

- **Cycle 1 – Fetch**
  - **Same for all instructions**
  - Operations
    - Send the PC to fetch instruction from memory and store in IR

IR ← Mem[PC]

  - Control Signals
    - IorD = **0** **(Select the PC as an address)**
    - MemRead = **1** **(Reading from memory)**
    - IRWrite = **1** **(Update IR)**

Instruction Execution

- **Cycle 2 – Decode**
  - Operations
    - Read two registers based on the rs1 and rs2 fields and store them in the A and B registers

      A 🡐 Reg[IR[19:15]]

      B 🡐 Reg[IR[24:20]]
    - Use the ALU to compute the branch address

      ALUOut 🡐 PC + (sign-extend(Imm) << 1)
    - Is it always a branch instruction???
  - Control Signals
    - ALUSrcA = **0** **(Select PC)**
    - ALUSrcB = **11** **(Select the sign-extended offsetx2)**
    - ALUOp = **00** **(Add operation)**

Instruction Execution

- **Cycle 3 – Execute & Taken Branch Completion**
  - The instruction is known!
  - Different operations depending on the instruction
  - Operations
    - **Memory Access Instructions (Load or Store)**
      - Use the ALU to compute the memory address

ALUOut 🡐 A + sign-extend(Imm)

- Control Signals
  - ALUSrcA = **1** **(Select A register)**
  - ALUSrcB = **10** **(Select the sign-extended offset)**
  - ALUOp = **00** **(Addition operation)**

Instruction Execution

- **Cycle 3 – Execute & Taken Branch Completion**
  - Operations
    - **ALU instructions**
      - Perform the ALU operation according to the ALUop and Func fields between registers A and B

ALUOut ← A op B

- Control Signals
  - ALUSrcA = **1** **(Select A register)**
  - ALUSrcB = **00** **(Select B register)**
  - ALUOp = **10** **(ALUoperation)**

Instruction Execution

- **Cycle 3 – Execute & Taken Branch Completion**
  - Operations
    - **Branch Equal Instruction**
      - Compare the two registers

if (A == B) then PC ← ALUOut

- Control Signals
  - ALUSrcA = **1** **(Select A register)**
  - ALUSrcB = **00** **(Select B register)**
  - ALUOp = **01** **(Subtract)**
  - Branch **= 1** **(Branch instruction)**
  - PCWrite **= 1** **(Write the PC)**

Instruction Execution

- **Cycle 4 – Memory Read or (R-type, Store, Not Taken Branch) Completion**
- Different operations depending on the instruction
  - Operations
    - **Load instruction**
      - Use the computed address (found in ALUOut) , read from memory and store value in MDR

        MDR 🡐 Memory[ALUOut]
      - Control Signals
        - IorD = **1** **(Address is for data)**
        - MemRead = **1** **(Read from memory)**
    - **Store instruction**
      - Use the computed address to store the value in register B into memory

        Memory[ALUOut] 🡐 B
      - Update the PC

        PC 🡐 PC + 4
      - Control Signals
        - IorD = **1** **(Address is for data)**
        - MemWrite = **1** **(Write to memory)**
        - ALUSrcA = 0 (Select PC as first input to ALU)
        - ALUSrcB = 01 (Select 4 as second input to ALU)
        - ALUOp = 00 (Addition)
        - PCWrite = 1 (Update PC)
        - Branch = 0 (Select PC+4)

Instruction Execution

- **Cycle 4 – Memory Read or (R-type, Store, Not Taken Branch) Completion**
  - Operations
    - **ALU instructions**
      - Write the results (ALUOut) into the register filer

        Reg[IR[11:7]] ← ALUOut
      - Update the PC

        PC ← PC + 4
      - Control Signals
        - MemToReg = **0** **(Data is from ALUOut)**
        - RegWrite = **1** **(Write to register)**
        - ALUSrcA = 0 (Select PC as first input to ALU)
        - ALUSrcB = 01 (Select 4 as second input to ALU)
        - ALUOp = 00 (Addition)
        - PCWrite = 1 (Update PC)
        - Branch = 0 (Select PC+4)

Instruction Execution

- **Cycle 4 – Memory Read or (R-type, Store, Not Taken Branch) Completion**
  - Operations
    - **Not Taken Branch Equal instruction**
      - Update the PC

        PC ← PC + 4

      - Control Signals
        - ALUSrcA = 0 (Select PC as first input to ALU)
        - ALUSrcB = 01 (Select 4 as second input to ALU)
        - ALUOp = 00 (Addition)
        - PCWrite = 1 (Update PC)
        - Branch = 0 (Select PC+4)

Instruction Execution

- **Cycle 5 – Memory Read Completion**
  - Needed for Load instructions only
  - Operations
    - **Load instruction**
      - Store the value loaded from memory and found in the MDR register in the register file based on the rd field of the instruction

        Reg[IR[11:7]] 🡐 MDR
      - Update the PC

        PC 🡐 PC + 4
      - Control Signals
        - MemToReg = **1** **(Data is from MDR)**
        - RegWrite = **1** **(Write to register)**
        - ALUSrcA = 0 (Select PC as first input to ALU)
        - ALUSrcB = 01 (Select 4 as second input to ALU)
        - ALUOp = 00 (Addition)
        - PCWrite = 1 (Update PC)
        - Branch = 0 (Select PC+4)

Instruction Execution

- **In the proposed multi-cycle implementation, we may need up to five cycles to execute the supported instructions**

Instruction Class	Clock Cycles Required
Load	5
Store	4
Branch	3 or 4
Arithmetic-logical	4

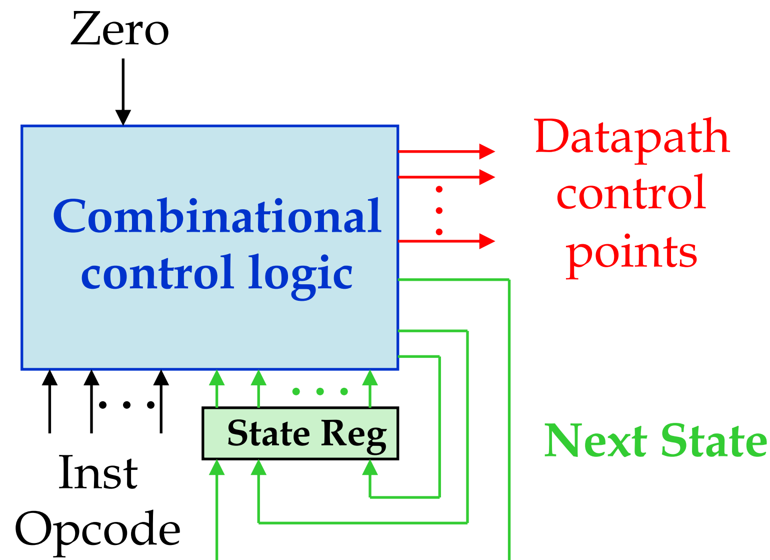
Multi-Cycle Control

(1) FSM Implementation

- The control of single-cycle is simple! All control signals are generated in the same cycle!
- **However, this is not true for the multi-cycle approach:**
  - The instruction execution is broken to multiple cycles
  - Generating control signals is not determined by the opcode only! It depends on the current cycle as well!
  - In order to determine what to do in the next cycle, we need to know what was done in the previous cycle!
  - Memorize ! Finite state machine (Sequential circuit)!

FSM

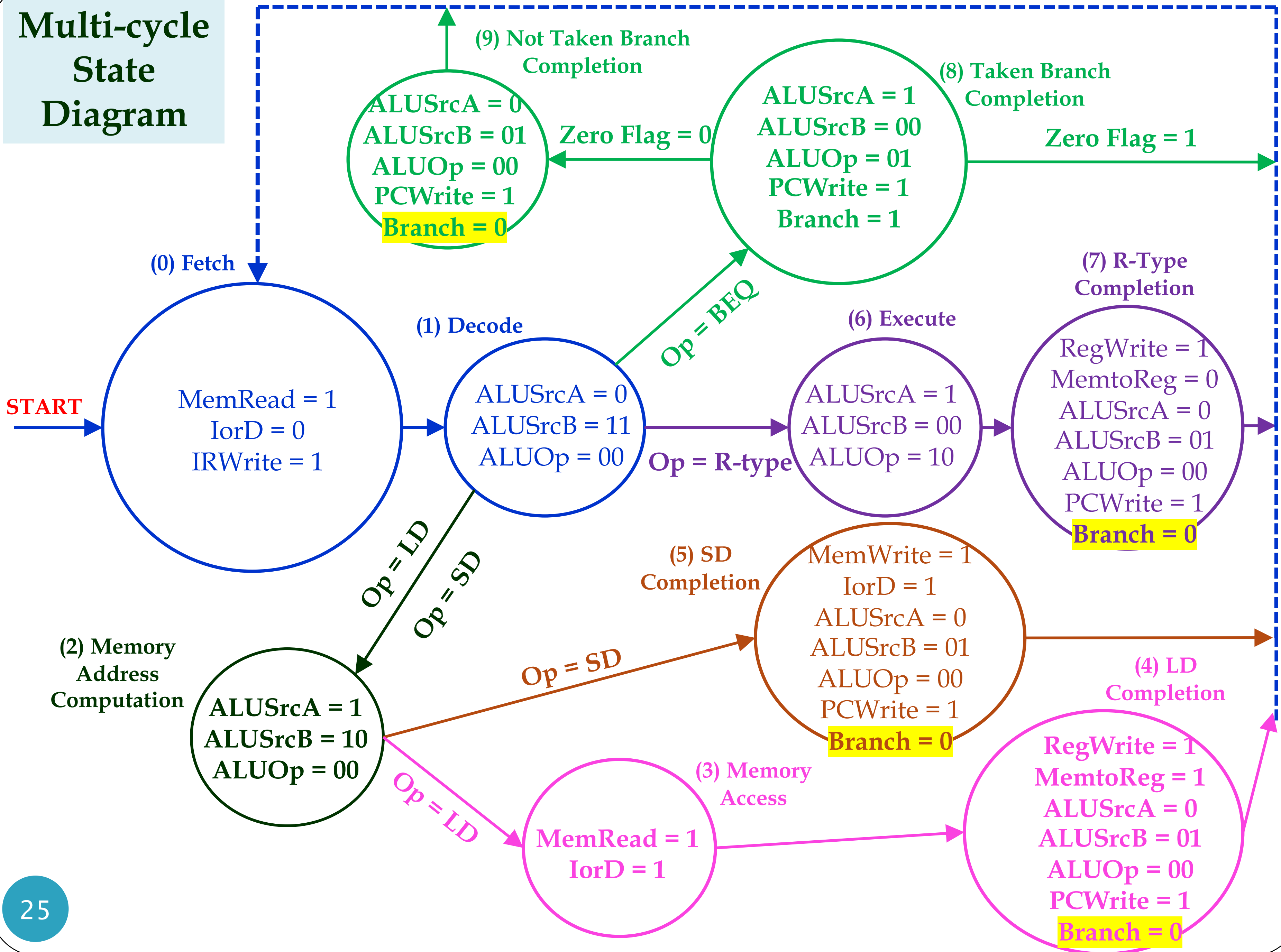
- A set of states (current state stored in State Register)
- Next state function (determined by current state and the input)
- Output function (determined by current state and the input)



Multi-Cycle Control

- Need to build the **state diagram**
  - Add a state whenever different operations are to be performed
  - For the supported instructions, we need 10 different states (next slide)
  - The first two states are the same for all instructions

- Once the state diagram is obtained, build the state table, derive combinational logic responsible for computing next state and outputs



Multi-cycle State Diagram
(0) Fetch
START
MemRead = 1
IorD = 0
IRWrite = 1
(1) Decode
ALUSrcA = 0
ALUSrcB = 11
ALUOp = 00
Op = BEQ
Op = R-type
Op = LD
Op = SD
(9) Not Taken Branch Completion
ALUSrcA = 0
ALUSrcB = 01
ALUOp = 00
PCWrite = 1
Branch = 0
Zero Flag = 0
(8) Taken Branch Completion
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 01
PCWrite = 1
Branch = 1
Zero Flag = 1
(6) Execute
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10
(7) R-Type Completion
RegWrite = 1
MemtoReg = 0
ALUSrcA = 0
ALUSrcB = 01
ALUOp = 00
PCWrite = 1
Branch = 0
(2) Memory Address Computation
ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00
Op = SD
(5) SD Completion
MemWrite = 1
IorD = 1
ALUSrcA = 0
ALUSrcB = 01
ALUOp = 00
PCWrite = 1
Branch = 0
Op = LD
(3) Memory Access
MemRead = 1
IorD = 1
(4) LD Completion
RegWrite = 1
MemtoReg = 1
ALUSrcA = 0
ALUSrcB = 01
ALUOp = 00
PCWrite = 1
Branch = 0

Finite State Machine (FSM)

- Control Unit Design as Moore Machine
  - State memory (SM): 4 bits that represent the current state ($S_3S_2S_1S_0$)
  - Next State Logic (NSL): ***Combinational logic*** responsible for determining the next state as function of current state and input ($NS_3NS_2NS_1NS_0$)
  - Output Function Logic (OFL): ***Combinational logic*** responsible for determining the control signals as function of current state

- The values for the signals that are not mentioned in a state are either:
  - Don't Care for MUX select signals and ALUOp OR
  - Deasserted (0) for all other control signals

NSL Truth Table

Current State ($S_3S_2S_1S_0$)	Input (Opcode + Zero Flag)	Next State ($NS_3NS_2NS_1NS_0$)
0000 (State0)	Op[6:0] = xxxxxxx	0001
0001 (State1)	Op[6:0] = 0000011 (LD)	0010
0001 (State1)	Op[6:0] = 0100011 (SD)	0010
0001 (State1)	Op[6:0] = 0110011 (R-format)	0110
0001 (State1)	Op[6:0] = 1100011 (BEQ)	1000
0010 (State2)	Op[6:0] = 0000011 (LD)	0011
0010 (State2)	Op[6:0] = 0100011 (SD)	0101
0011 (State3)	Op[6:0] = xxxxxxx	0100
0100 (State4)	Op[6:0] = xxxxxxx	0000
0101 (State5)	Op[6:0] = xxxxxxx	0000
0110 (State6)	Op[6:0] = xxxxxxx	0111
0111 (State7)	Op[6:0] = xxxxxxx	0000
1000 (State8)	(Op[6:0] = xxxxxxx) && (Zero Flag = 1)	0000
1000 (State8)	(Op[6:0] = xxxxxxx) && (Zero Flag = 0)	1001
1001 (State9)	Op[6:0] = xxxxxxx	0000

- $NS_0 = State0 + State2.\big((Op = LD) + (Op = SD)\big) + State6 + State8.(Zero_Flag = 0)$
- $NS_0 = \overline{S_3}\,\overline{S_2}\,\overline{S_1}\,\overline{S_0} + \overline{S_3}\,\overline{S_2}S_1\overline{S_0}.(\overline{Op_6}\,\overline{Op_5}\,\overline{Op_4}\,\overline{Op_3}\,\overline{Op_2}Op_1Op_0 + \overline{Op_6}Op_5\overline{Op_4}\,\overline{Op_3}\,\overline{Op_2}Op_1Op_0)$ $+ \overline{S_3}\,S_2S_1\overline{S_0} + S_3\,\overline{S_2}\,\overline{S_1}\,\overline{S_0}\bar{Z}$
- Using the same approach write the equations for $NS_0, NS_1, and\ NS_2$.

OFL Truth Table

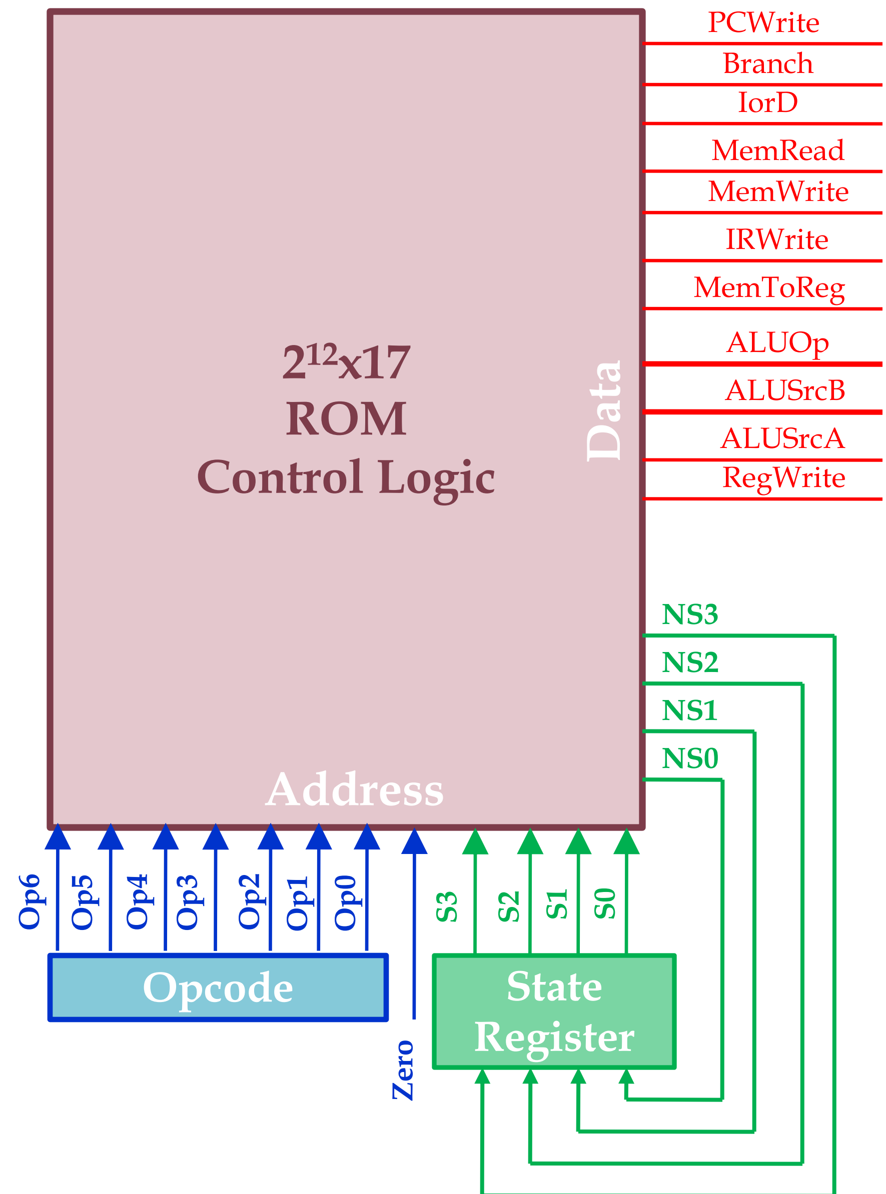
Current State ($S_3S_2S_1S_0$)	PCWrite	Branch	IorD	MemRead	MemWrite	IRWrite	MemToReg	ALUOp<1>	ALUOp<0>	ALUSrcB<1>	ALUSrcB<0>	ALUSrcA	RegWrite
0000 (State0)	0	X	0	1	0	1	X	X	X	X	X	X	0
0001 (State1)	0	X	X	0	0	0	X	0	0	1	1	0	0
0010 (State2)	0	X	X	0	0	0	X	0	0	1	0	1	0
0011 (State3)	0	X	1	1	0	0	X	X	X	X	X	X	0
0100 (State4)	1	0	X	0	0	0	1	0	0	0	1	0	1
0101 (State5)	1	0	1	0	1	0	X	0	0	0	1	0	0
0110 (State6)	0	X	X	0	0	0	X	1	0	0	0	1	0
0111 (State7)	1	0	X	0	0	0	0	0	0	0	1	0	1
1000 (State8)	1	1	X	0	0	0	X	0	1	0	0	1	0
1001 (State9)	1	0	X	0	0	0	X	0	0	0	1	0	0

- $PCWrite = State_4 + State_5 + State_7 + State_8 + State_9 = \overline{S_3}S_2\overline{S_1}\,\overline{S_0} + \overline{S_3}S_2\overline{S_1}S_0 + \overline{S_3}S_2S_1S_0 + S_3\overline{S_2}\,\overline{S_1}\,\overline{S_0} + S_3\overline{S_2}\,\overline{S_1}S_0$
- $IorD = State_3 + State_5 = \overline{S_3}\ \overline{S_2}S_1S_0 + \overline{S_3}S_2\overline{S_1}S_0$
- Using the same approach write the equations for the remaining control signals.

Multi-Cycle Control

(2) ROM Implementation

- **FSM design**
  - 12 inputs
  - 17 outputs
  - TT size = 2^{12}x17
- **ROM**
  - Can be used to implement the truth table above
  - ROM Size = 69632 bits
  - Each location stores the control signals values and the next state
  - Each location is addressable by the opcode and current state value
  - For which ROM addresses will the RegWrite be 1?
    - xxxxxxxx0111
    - xxxxxxxx0100
    - Total of 512 addresses
  - For our design, Only 15 locations are needed → ***Huge waste of space***



Multi-Cycle Control

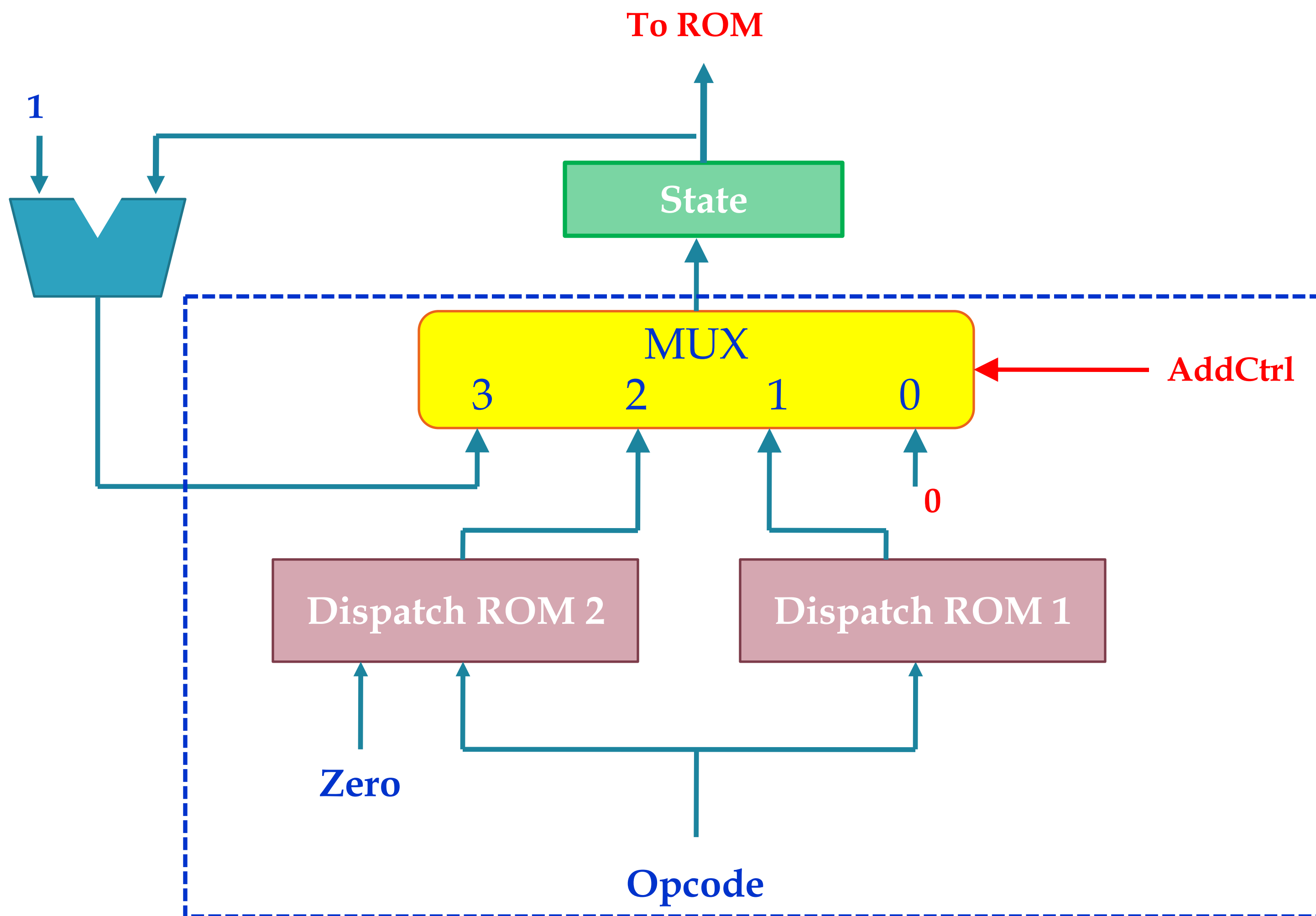
- ROM implementation is vulnerable to bugs and expensive especially for complex CPU
- Size increase as the number and complexity of instructions (states) increases
- If we use two ROMs, one for NSL and one for OFL:
  - ***NSL ROM*** is addressable by the opcode, zero flag and current state value:
    - NSL ROM Size = 2^{12} x 4
  - ***OFL ROM*** is addressable by the current state value:
    - OFL ROM Size = 2^{4} x 13
  - Total Size = 16592 bits (much lower than a single ROM)
  - NSL consumes approximately 99% of the ROM area

- **Solution: Implement the Next State Function with a Sequencer**

Multi-Cycle Control

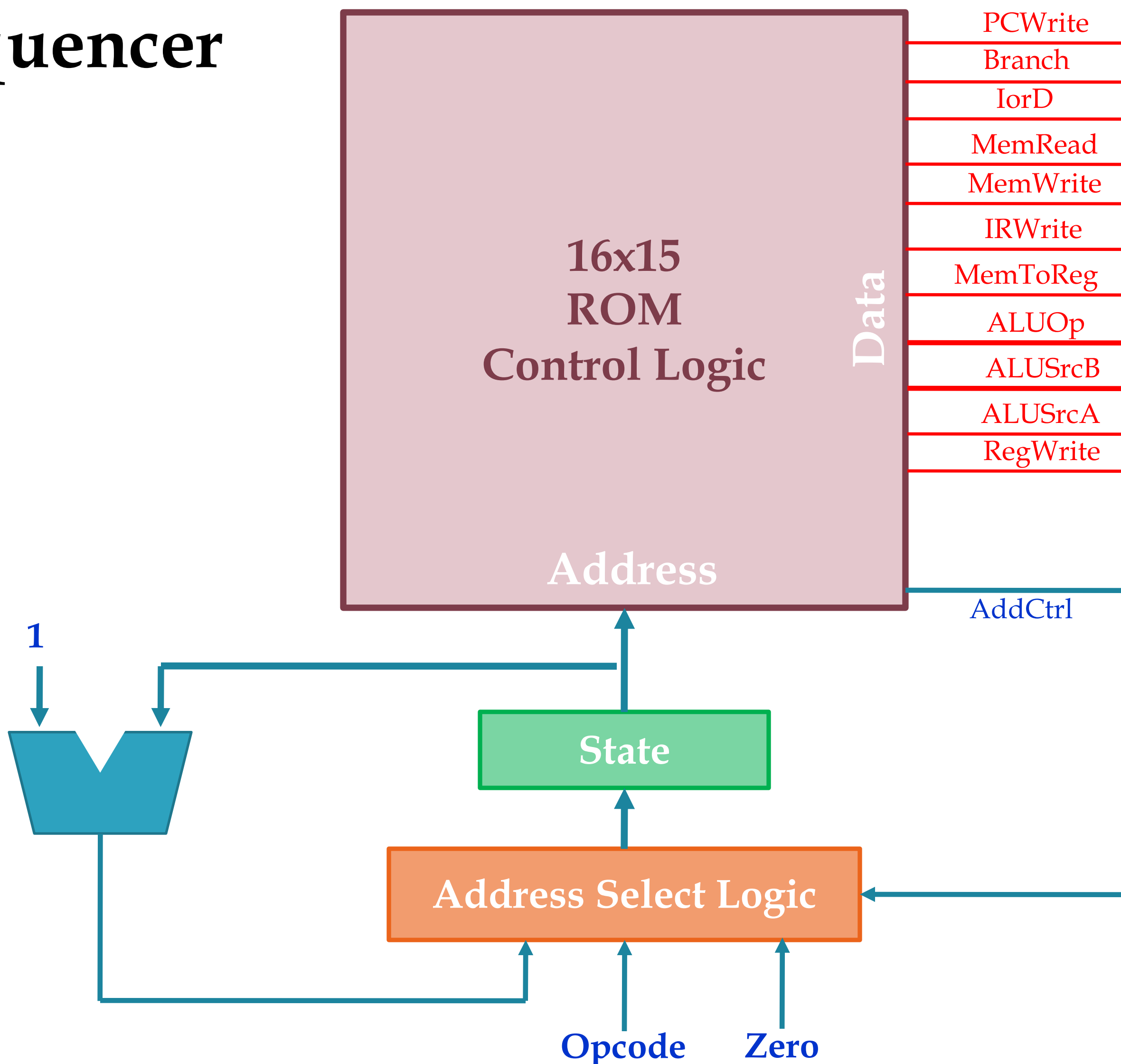
- **Implementing the Next State Function with a Sequencer**

Inside the address select logic



Multi-Cycle Control

- **Implementing the Next State Function with a Sequencer**



Multi-Cycle Control

- **Implementing the Next State Function with a Sequencer**

Dispatch ROM1		
Opcode Field	Opcode Name	Value
0110011	R-Format	0110
1100011	beq	1000
0000011	ld	0010
0100011	sd	0010

	Dispatch ROM2		
Zero	Opcode Field	Opcode Name	Value
0	1100011	beq	1001
1	1100011	beq	0000
0	0000011	ld	0011
1	0000011	ld	0011
0	0100011	sd	0101
1	0100011	sd	0101

State number	Address-control action	Value of AddrCtl
0	Use incremented state	3
1	Use dispatch ROM 1	1
2	Use dispatch ROM 2	2
3	Use incremented state	3
4	Replace state number by 0	0
5	Replace state number by 0	0
6	Use incremented state	3
7	Replace state number by 0	0
8	Use dispatch ROM 2	2
9	Replace state number by 0	0

AddrCtl value	Action
0	Set state to 0
1	Dispatch with ROM 1
2	Dispatch with ROM 2
3	Use the incremented state

Multi-Cycle ALU Control

- Same as Single-Cycle ALU Control

Multi-Cycle Performance

- **Example 1. Compare the performance of the multi-cycle and single-cycle implementations for the SPECINT2000 program which has the following instruction mix: 25% loads, 10% stores, 11% Taken branches, 2% Not Taken branches, 52% ALU.**

- **$Time_{SC} = IC \times CPI_{SC} \times CC_{SC}$**
  $= IC \times 1 \times CC_{SC} = IC_{SC} \times CC_{SC}$

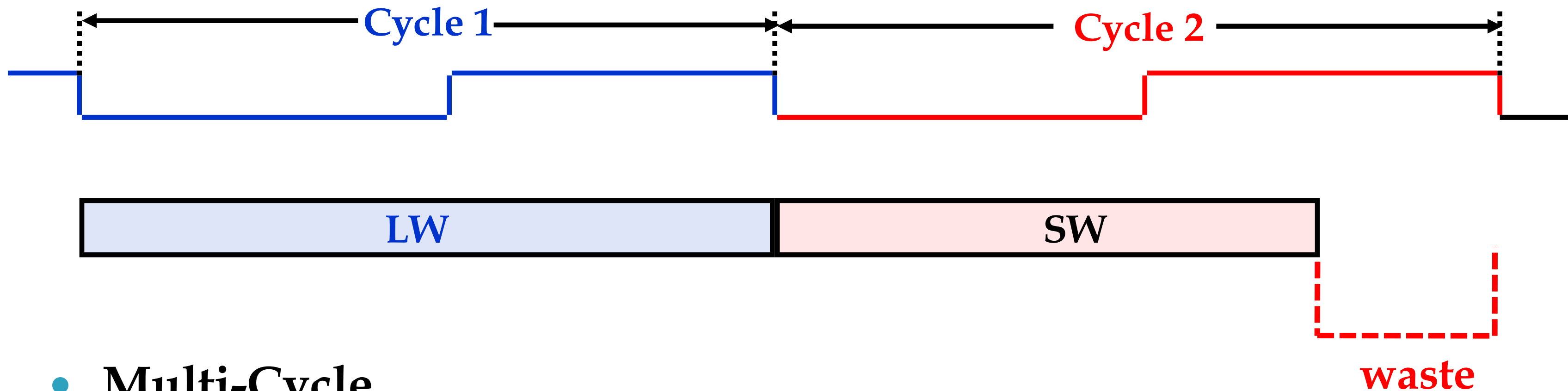
- **$Time_{MC} = IC \times CPI_{MC} \times CC_{MC}$**
  $CPI_{MC} = 0.25 \times 5 + 0.1 \times 4 + 0.11 \times 3 + 0.02 \times 4 + 0.52 \times 4 = 4.14$
  $CC_{MC} = 1/5 * CC_{SC}$ (Is that true!!)

- **Speedup = $Time_{SC} / Time_{MC} = 5 / 4.14 = 1.21$!**

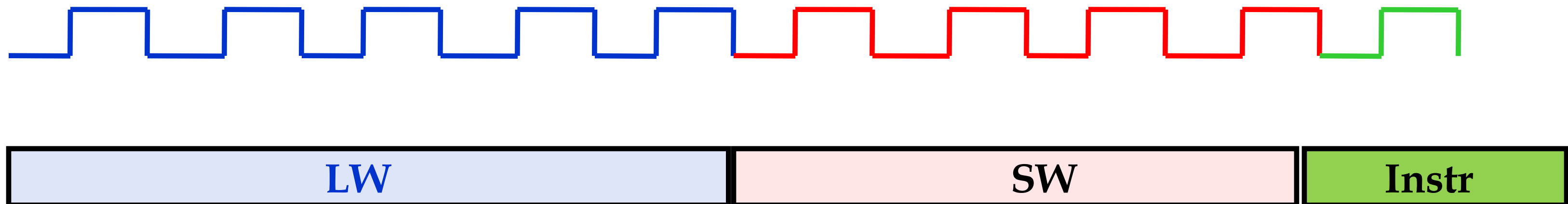
- **Multi-cycle is cost effective as well, as long as the time for different processing units are balanced!**

Multi-Cycle Performance

- **Single-Cycle**



- **Multi-Cycle**



- **This is true as long as the delay of all functional units is balanced!**

Multi-Cycle Performance

- **Example 2. Redo example 1 without assuming that the cycle time for multi-cycle is 1/5 that of single cycle. Assume the delay times of different units as given in the table.**

Unit	Time (ps)
Memory	200
ALU and adders	100
Register File	50

- $Time_{SC} = IC \times CPI_{SC} \times CC_{SC}$
  $= IC \times 1 \times 600 = 600\ IC$

- $Time_{MC} = IC \times CPI_{MC} \times CC_{MC}$
  $CPI_{MC} = 0.25x5 + 0.1x4 + 0.11x3 + 0.02 \times 4 + 0.52 \times 4 = 4.14$
  $CC_{MC} = 200$ (should match the time of the slowest functional unit)
  $Time_{MC} = IC \times 4.14 \times 200 = 828\ IC$

- **Speedup =** $Time_{SC} / Time_{MC} = 600 / 828 = 0.725$!