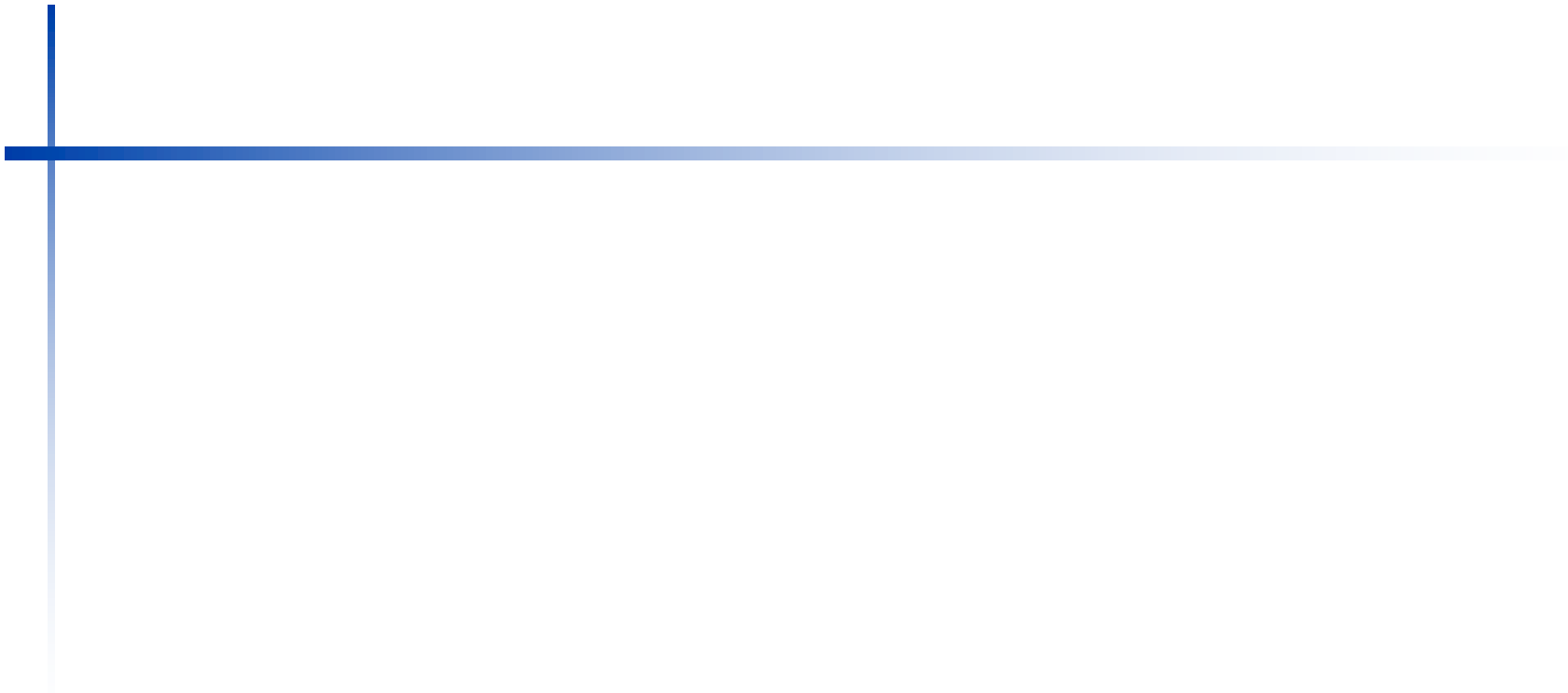


Chapter 1

Computer Abstractions and Technology

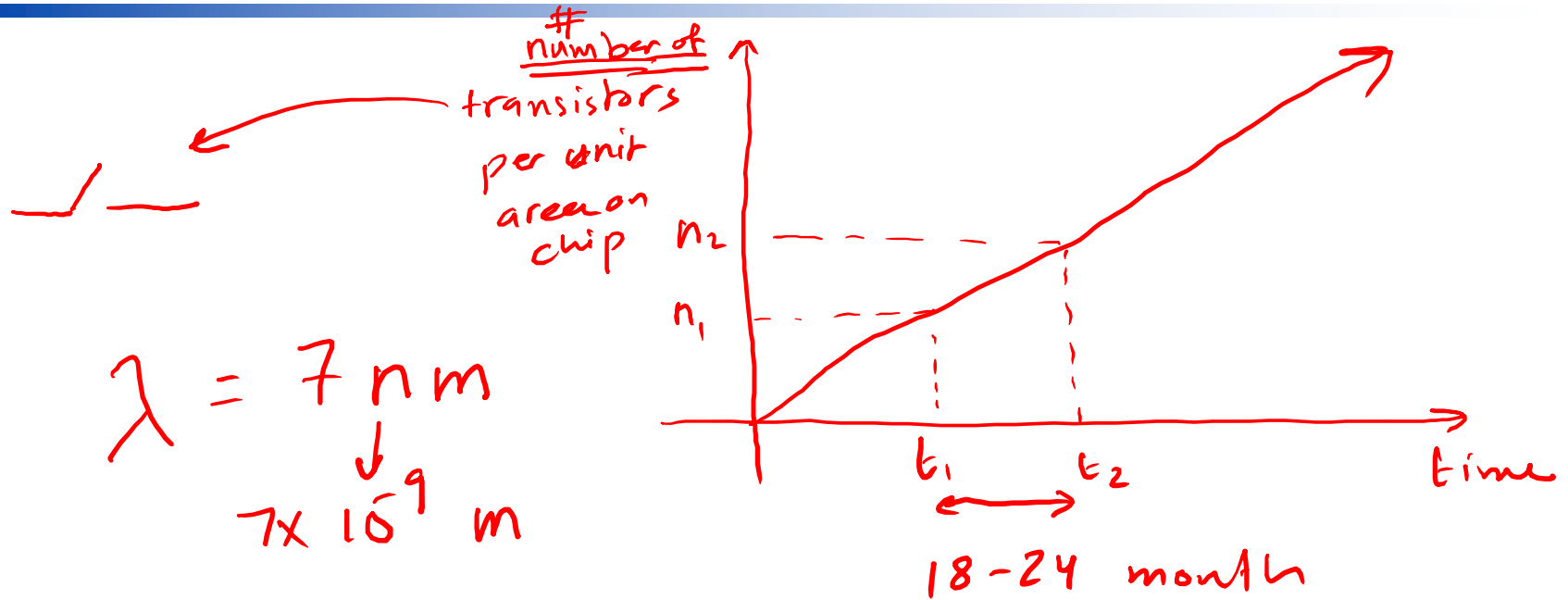


Updated by Dr. Waleed Dweik

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

Moore's Law



$$n_2 = 2 \times n_1$$

transistors per unit area doubles every
18 - to - 24 months

Classes of Computers

- Personal computers
 - General purpose, variety of software
 - Subject to cost/performance tradeoff
- Server computers
 - Network based
 - High capacity, performance, reliability

Classes of Computers

■ Supercomputers

top500.org

- 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

- ① ■ Personal Mobile Device (PMD)
 - Battery operated
 - Connects to the Internet
 - Hundreds of dollars
 - Smart phones, tablets, electronic glasses
- ② ■ 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

Eight Great Ideas (1)

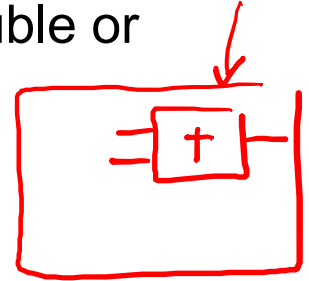
① Design for Moore's Law



- 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

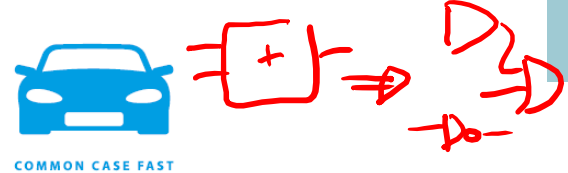
Transistors
 $(Z) \underline{2021} \rightarrow 2023 (2 * Z)$

② 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

③ Make the common case fast



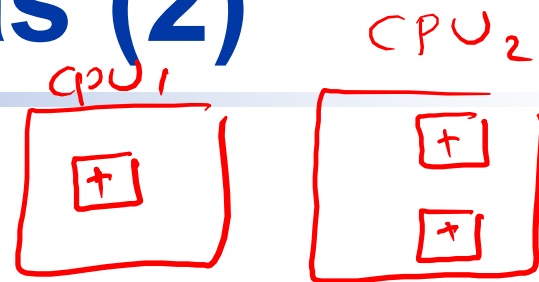
- Enhance the performance is better than optimizing the rare case
- Example?

Solution 3: addition(+) 1.5 ns
 division(-) 20 ns

$$\begin{aligned} \text{Time} &= \underline{1.5 \times 0.9 \gamma} + 20 \times 0.1 \gamma \\ &= 1.35 \gamma + 2 \gamma = 3.35 \gamma \end{aligned}$$

Eight Great Ideas (2)

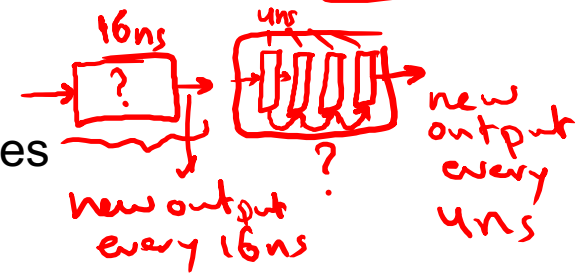
4 Performance via parallelism



PARALLELISM

5 Performance via pipelining

- Special pattern of parallelism
- Example: Human bridge to fight fires



PIPELINING

6 Performance via prediction

- Better ask for forgiveness than ask for permission

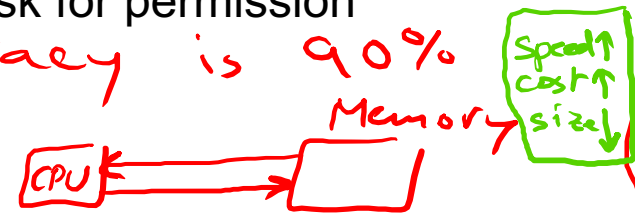


PREDICTION

7 Hierarchy of memories

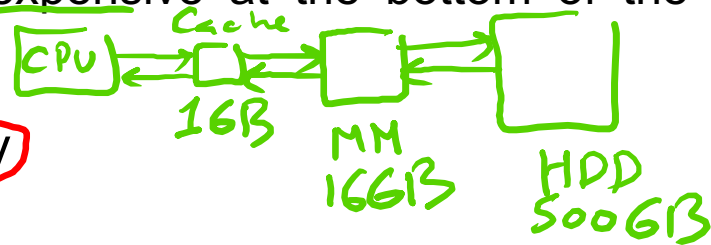
accuracy is 90%

- 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)



HIERARCHY

8 Dependability via redundancy



DEPENDABILITY

Eight Great Ideas (3)

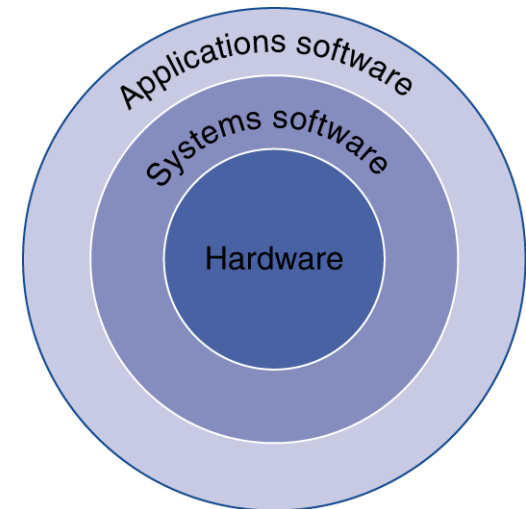
- Why these ideas are considered great?
 - Lasted long after the 1st computer that used them
 - They have been around for the last 60 years

Below Your Program

- ① 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
- $a = b + c ;$

- ② System software
- ^{2.1} Compiler: translates HLL code to machine code ✓
 - ^{2.2} Operating System: service code
 - Handling input/output
 - Managing memory and storage
 - Scheduling tasks & sharing resources
 - E.g. Windows, Linux, iOS

- ③ Hardware
- Processor, memory, I/O controllers } ✓
CPU



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)
```

shift left logical immediate

load double

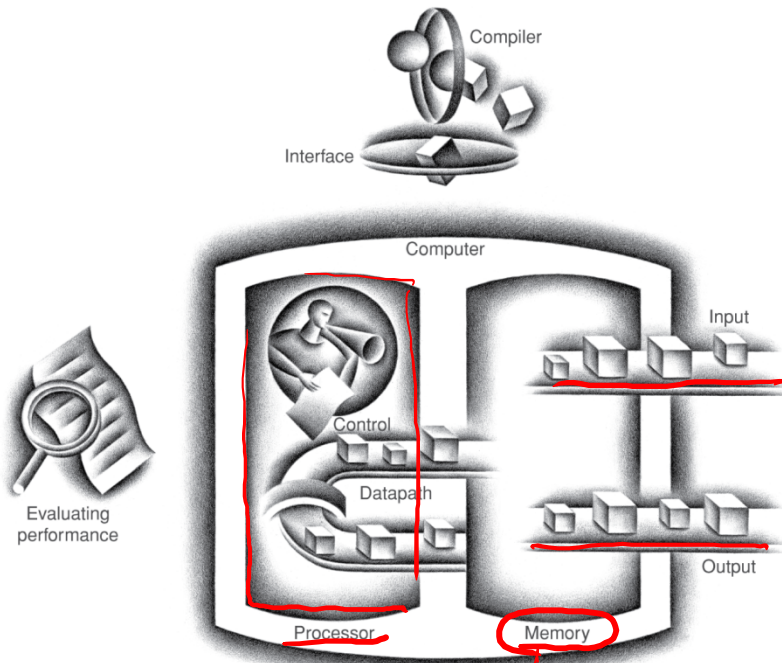
Assembler

Binary machine language program (for RISC-V)

```
00000000001101011001001100010011
000000000011001010000001100110011
000000000000000110011001010000011
000000000100000110011001110000011
0000000001110011001100000100011
00000000010100110011010000100011
0000000000000000100000001100111
```

Components of a Computer

The BIG Picture

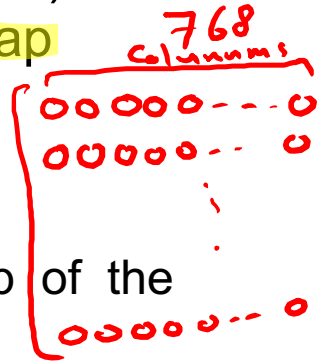


- Same components for all kinds of computer
 - Desktop, server, embedded
- Five components
 - Input, output, memory, datapath and control } ⇒ CPU processor
- Input/output includes
 - ① User-interface devices
 - Display, keyboard, mouse
 - ② Storage devices
 - Hard disk, CD/DVD, flash
 - ③ Network adapters
 - For communicating with other computers

Cache
Main
Memory

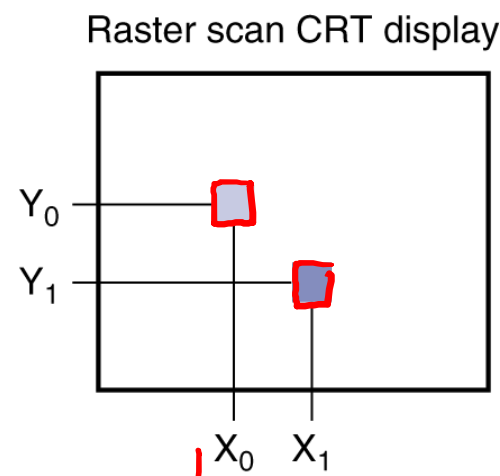
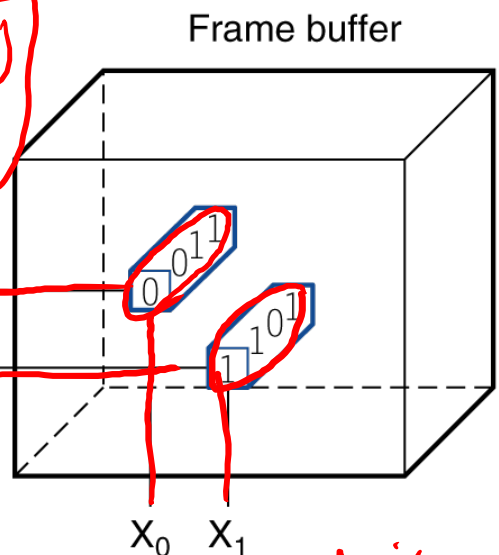
Through the Looking Glass

- 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



$\frac{1}{24}$ second

Frame size =
Bit map size \times
number of bits per
pixel

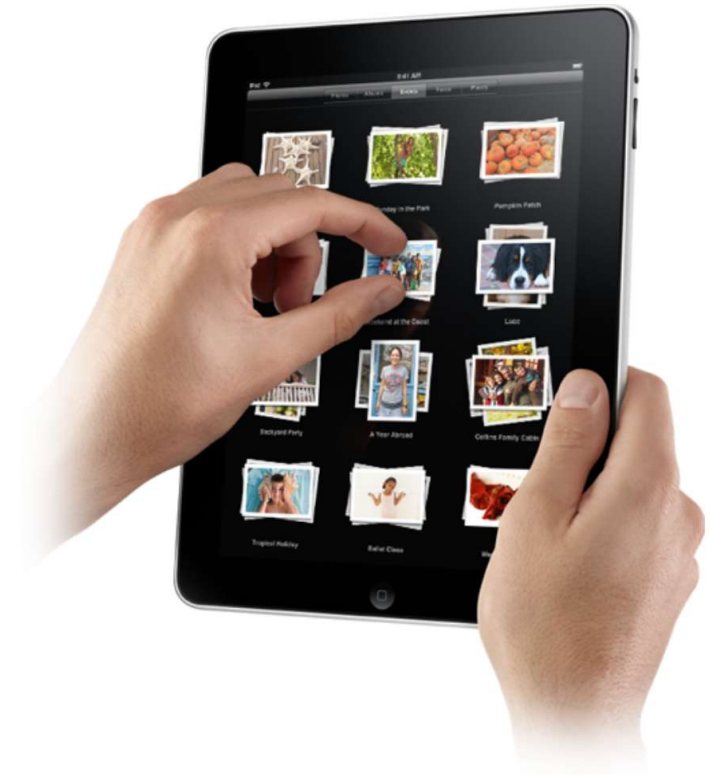


$\frac{24 \text{ fps}}{\downarrow}$
frame per
second

$= 1024 \times 768 \times 4$ bits every $\frac{1}{24}$ seconds

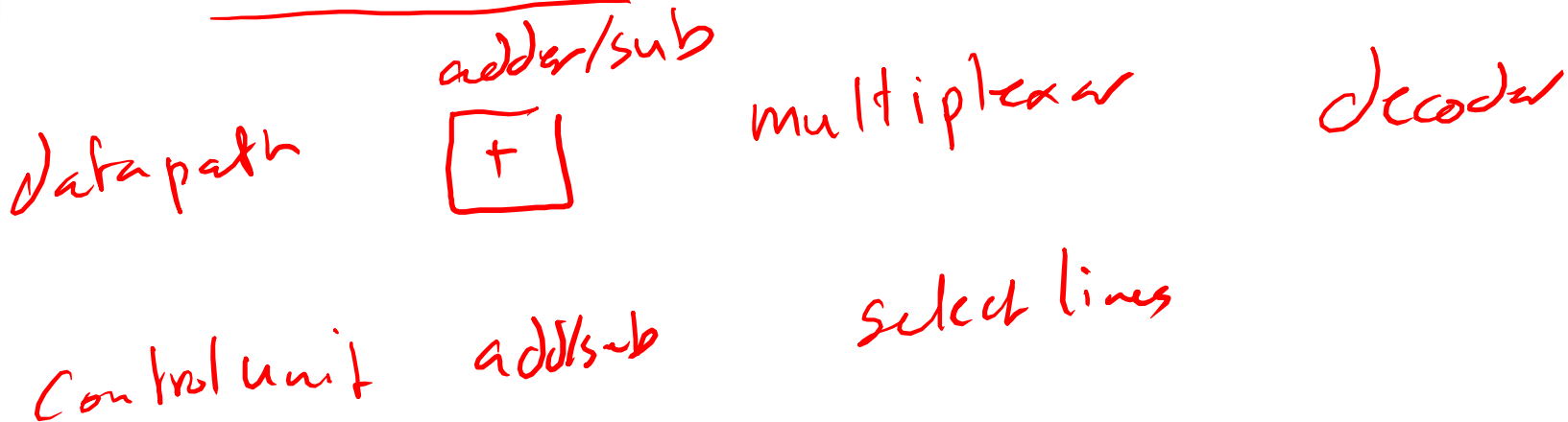
Touchscreen

- PostPC device
- Supersedes keyboard and mouse
- ① Resistive and ② Capacitive types
 - Most tablets, smart phones use capacitive
 - Capacitive allows multiple touches simultaneously



Inside the Processor (CPU)

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



A Safe Place for Data (1)

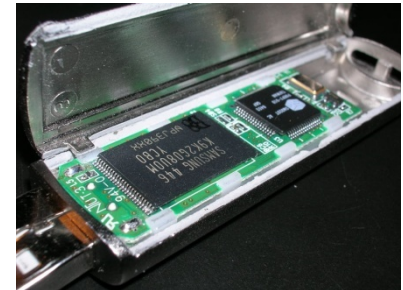
- Volatile main (primary) memory
 - DRAMs: Dynamic Random Access Memory
 - Holds data and programs while running
 - Loses instructions and data when power off
- Non-volatile secondary memory
 - Magnetic disk
 - Flash memory in PMDs
 - Optical disk (CDROM, DVD)

Main Memory

Storage devices

input/output devices

Large Capacitances



A Safe Place for Data (2)

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

10⁹ bytes

SSD

HDD

Fast (50-70)ns $\rightarrow 10^{-9}$

Slow (5-20)ms $\rightarrow 10^{-3}$

Moderate (5-50) μ s $\rightarrow 10^{-6}$

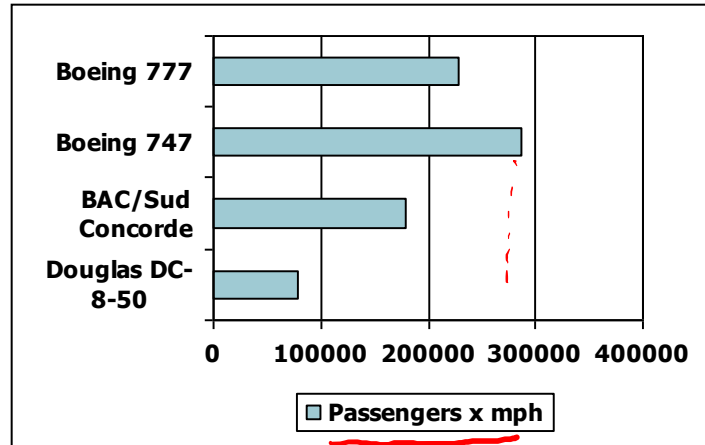
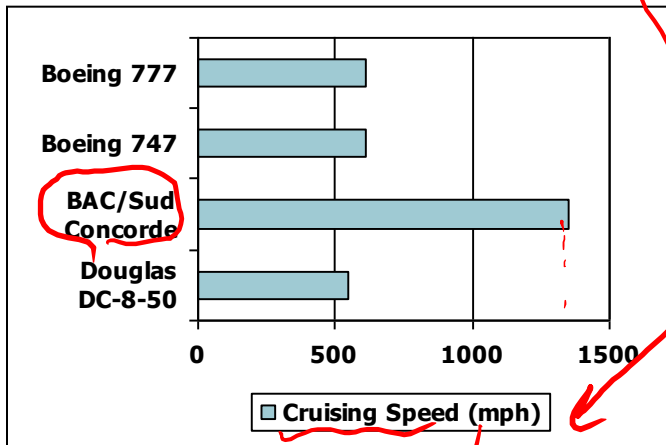
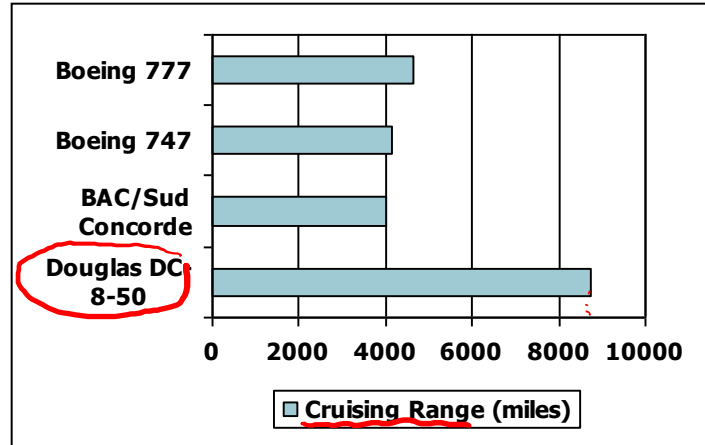
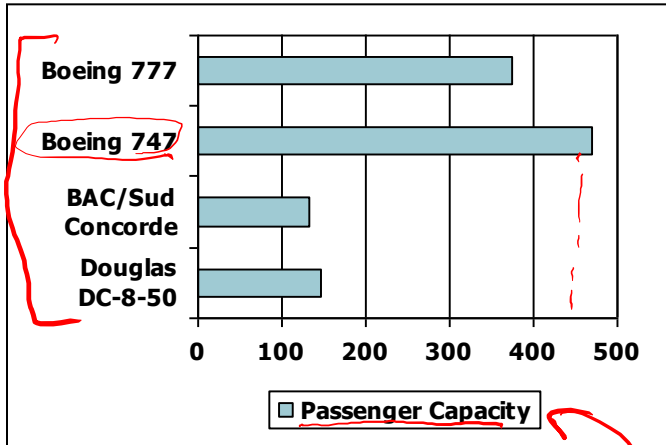
$$500 \text{ GB} \times 0.1 = 50 \$$$

$$128 \text{ GB} \times 1 = 128 \$$$

SSD

Defining Performance

- Which airplane has the **best performance?**



mile per hour

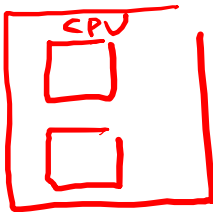
Response Time and Throughput

- 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

① ■ Replacing the processor with a faster version?

② ■ Adding more processors?

■ We'll focus on response time for now...



response time ↓
throughput ↑

→ response time (same)
throughput ↑

Relative Performance

ET

- Define Performance = 1/Execution Time

- "X is n time faster than Y"

$$\frac{\text{Performance}_X}{\text{Performance}_Y} = n$$

$$\text{Performance}_X / \text{Performance}_Y$$

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

$$\frac{1/ET_X}{1/ET_Y} = n$$

- Example: time taken to run a program

- 10s on A, 15s on B

- Execution Time_B / Execution Time_A

$$= 15s / 10s = 1.5$$

$$1.5 - 1 = 0.5 \times 100\% = 50\%$$

- So A is 1.5 times faster than B

A is 50%

" " "

$$\frac{ET_Y}{ET_X} = n$$

Measuring Execution Time

- Elapsed time (response or wall clock time) *or Execution Time*
 - Total response time, including all aspects
 - ① Processing, ② I/O, ③ OS overhead, ④ idle time
 - System performance = $1 / \text{Elapsed time}$
- CPU time = User CPU time + System CPU time
 - Time spent processing a given job
 - Discounts I/O time, other jobs' shares *idle time*
 - 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 = $1 / \text{User CPU time}$

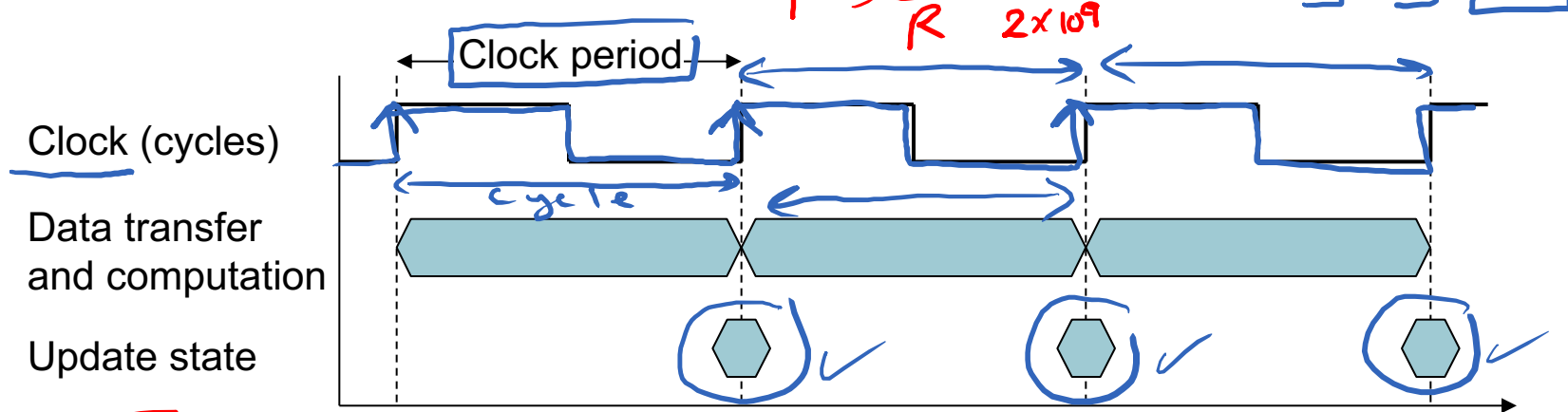
CPU Clocking

Digital Synchronous System

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

$$R = 2 \text{ GHz}$$

$$T = \frac{1}{R} = \frac{1}{2 \times 10^9} = 0.5 \text{ ns}$$



- Clock period: duration of a clock cycle

- e.g., $250 \text{ ps} = 0.25 \text{ ns} = 250 \times 10^{-12} \text{ s}$

- Clock frequency (rate): cycles per second

- e.g., $4.0 \text{ GHz} = 4000 \text{ MHz} = 4.0 \times 10^9 \text{ Hz}$

$$\frac{1}{250 \times 10^{-12}}$$

$$\frac{10^{12}}{250} = \frac{1000 \times 10^9}{250}$$

$$T = \frac{1}{R} = \frac{1}{F}$$

$$F = R = \frac{1}{T}$$

$$4 \times 10^9 \text{ Hz}$$

$$4 \text{ GHz}$$

CPU Time

$$\begin{aligned} \text{CPU Time} &= \text{CPU Clock Cycles} \times \text{Clock Cycle Time} \\ &= \frac{\text{CPU Clock Cycles}}{\text{Clock Rate}} \end{aligned}$$

T
Clock period

- 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 \Rightarrow $CPU_{time} = \frac{CPU\ clock\ cycles}{Clock\ Rate}$
- Designing Computer B
 - Aim for 6s CPU time $CPU\ clock\ cycle = 1.2 \times 2 \times 10^{10} = 24 \times 10^9\ cycles$
 - Can do faster clock, but causes 1.2 × clock cycles $10 = \frac{CPU\ clock\ cycles}{2 \times 10^9}$
- How fast must Computer B clock be?

$$Clock\ Rate_B = \frac{Clock\ Cycles_B}{CPU\ Time_B} = \frac{1.2 \times Clock\ Cycles_A}{6s}$$

$$Clock\ Cycles_A = CPU\ Time_A \times Clock\ Rate_A$$

$$= 10s \times 2GHz = 20 \times 10^9$$

$$Clock\ Rate_B = \frac{1.2 \times 20 \times 10^9}{6s} = \frac{24 \times 10^9}{6s} = \underline{4GHz}$$

* B is $\frac{10}{6}$ times faster than A

$$CPU_{time} = \frac{24 \times 10^9}{clock\ Rate_B}$$

$$clock\ Rate_B = \frac{24 \times 10^9}{6} = 4 \times 10^9\ Hz = 4\ GHz$$

Instruction Count and CPI

each program consists of multiple
CPI instruction

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

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

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

- 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

** program X consists of 100 instructions*
** each instruction require 3 cycles to complete*

$$100 \times 3 = 300 \text{ cycles}$$

$$\text{CPU time} = 300 \times T = \frac{300}{R}$$

CPI Example

on Average, each instruction requires 2 cycles to execute

■ Computer A: Cycle Time = 250ps, CPI = 2.0

■ Computer B: Cycle Time = 500ps, CPI = 1.2

→ Same ISA and Compiler

$$IC_A = IC_B$$

■ Which is faster, and by how much?

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

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

A is faster...

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

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

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

...by this much

A is 1.2 times faster than B
A is 20% " " "

"Same ISA + Same Compiler \implies Same IC
(Same IC_i)"

Same Hardware \implies Same CPI_i
Same T (Same R)

CPI in More Detail

- If different instruction classes take different numbers of cycles

clock cycles = $IC \times CPI_{avg}$

total (pointing to the result)

avg (pointing to the CPI term)

program X

add	10
sub	5
divide	3
multiply	4
	<u>22</u>

instruction classes

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

19/4x

$$= 1 \times 10 + 1 \times 5 + 10 \times 3 + 7 \times 4$$

Weighted average CPI = $10 + 5 + 30 + 28 = 73$

10
<u>22</u>
5
<u>22</u>
3
<u>22</u>
4
<u>22</u>

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

CPU time =

$$\text{Clock Cycles} \times T$$

$$= \sum_{i=1}^n (\text{CPI}_i \times \text{IC}_i) \times T$$

$$= \text{IC} \times \text{CPI}_{avg} \times T$$

Relative frequency

$$\text{CPUtime} = \sum_{i=1}^n (\text{CPI}_i \times \text{IC}_i) \times T = \frac{\text{IC} \times \text{CPI}_{\text{avg}} \times T}{\text{IC}}$$

$$\text{CPI}_{\text{avg}} = \frac{\sum_{i=1}^n \text{CPI}_i \times \text{IC}_i}{\text{IC}}$$

$$\text{CPI}_{\text{avg}} = \sum_{i=1}^n \frac{\text{CPI}_i \times \text{IC}_i}{\text{IC}} = \sum_{i=1}^n \text{CPI}_i \times \frac{\text{IC}_i}{\text{IC}}$$

$\frac{\text{IC}_i}{\text{IC}}$: Relative frequency of instruction i

$$\text{Clock Cycles} = \underline{\text{IC}} \times \text{CPI}_{\text{avg}}$$

$$\text{CPI}_{\text{avg}} = \frac{\text{Clock Cycles}}{\text{IC}}$$

CPI Example

$$1 \times \frac{2}{5} + 2 \times \frac{1}{5} + 3 \times \frac{2}{5}$$

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

$$CPI_{avg} = \sum_{i=1}^3 CPI_i \times \frac{IC_i}{IC} = CPI_A \times \frac{IC_A}{IC} + CPI_B \times \frac{IC_B}{IC} + CPI_C \times \frac{IC_C}{IC}$$

Class	A	B	C
<u>CPI for class</u>	1 ✓	2 ✓	3 ✓
<u>IC in sequence 1</u> ✓	2	1 ✓	2 ✓
<u>IC in sequence 2</u> ✓	4	1	1

IC₁ = 5
IC₂ = 6

$$CPI_{2, avg} = 1 \times \frac{4}{6} + 2 \times \frac{1}{6} + 3 \times \frac{1}{6}$$

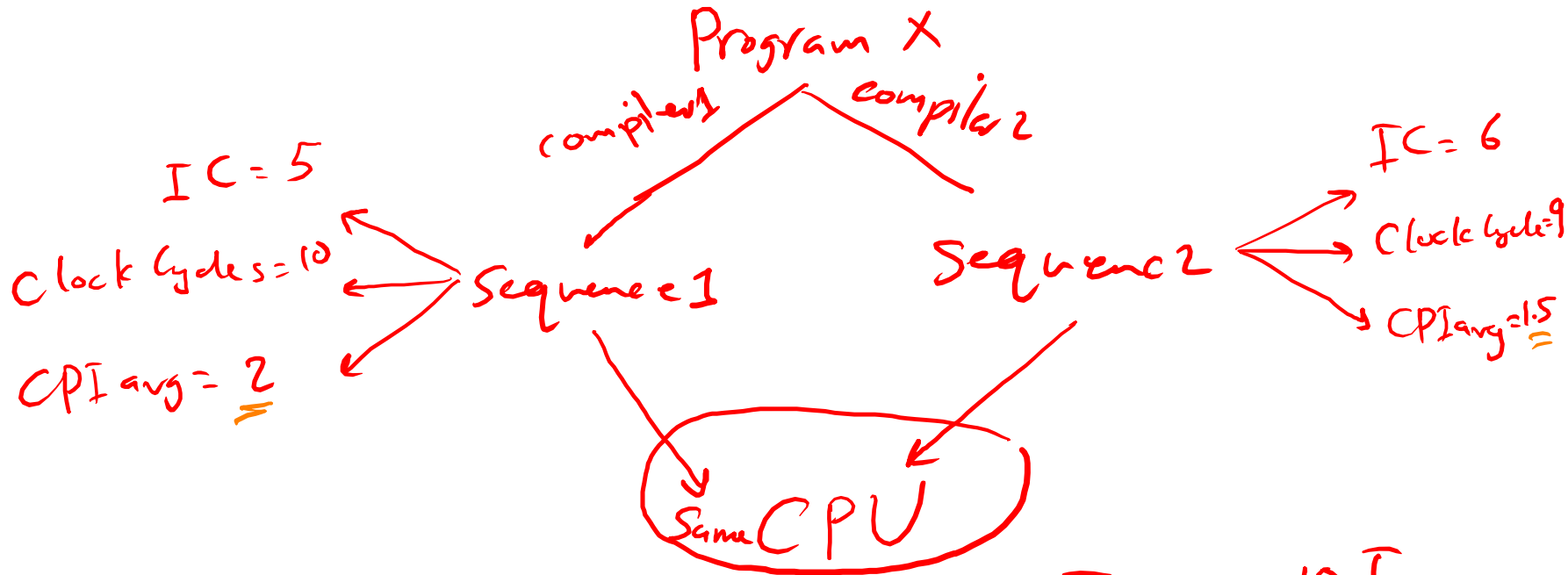
- Sequence 1: IC = 5

- Clock Cycles = $\sum CPI_i \times IC_i$
= $2 \times 1 + 1 \times 2 + 2 \times 3$
= 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

* seq 2 is $\frac{10}{9}$ faster than seq 1



$$\begin{aligned}
 \text{CPU time}_{\text{seq 1}} &= \text{Clock Cycles}_1 \times T = 10T \\
 &= IC_1 \times CPI_{\text{avg}} \times T = 5 \times 2 \times T = 10T
 \end{aligned}$$

$$\begin{aligned}
 \text{CPU time}_{\text{seq 2}} &= \text{Clock Cycles}_2 \times T = 9T \\
 &= IC_2 \times CPI_{2, \text{avg}} \times T = 6 \times 1.5 \times T \\
 &= 9T
 \end{aligned}$$

Given Two Compiled Sequences of the same program with the following information.

$$CPI_{avg_1} = \sum_{i=1}^4 CPI_i \times \frac{IC_i}{IC}$$

$$= 3 \times 0.2 + 2 \times 0.1 + 1 \times 0.4 + 4 \times 0.3$$

$$= 0.6 + 0.2 + 0.4 + 1.2 = 2.4$$

$$CPI_{avg_2} = 3 \times 0.5 + 2 \times 0.05 + 4 \times 0.45$$

$$= 1.5 + 0.1 + 1.8 = 3.4$$

$$CPU_{time_1} = IC_1 \times CPI_{avg_1} \times T_1 = IC_1 \times 2.4 \times T_1$$

class	A	B	C	D
Seq 1	<u>20%</u>	10%	40%	30%
Seq 2	50%	5%	0%	45%

on the same CPU

class	A	B	C	D
CPI _i	3	2	1	4

$$CPU_{time_2} = IC_2 \times CPI_{avg_2} \times T_2 = IC_2 \times 3.4 \times T_2$$

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?

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

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

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

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

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

	A	B	C
CPI	1	2	3

$$\frac{CPU\ time_2}{CPU\ time_1} = 2$$

$$\frac{Clock\ Cycle_2 \times \cancel{CPI_2}}{Clock\ Cycle_1 \times \cancel{CPI_1}} = 2$$

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?)
 - ① All computers are constructed from hardware technologies based on similar underlying principles
 - ② 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
- ① RISC: Reduced Instruction Set Computer
 - Fixed length, fast execution, and simple functionality
- ② 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

destination

2

← add a, b, c # a = b + c

source

assembly RISC-V addition instruction

- 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 t0, b, c
add t1, d, e
add a, t0, t1

```

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

it is ok to have the same operand as destination and as source

- 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 f, g, h
sub f, f, i
sub f, f, j

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

subtraction

source1 operation source2

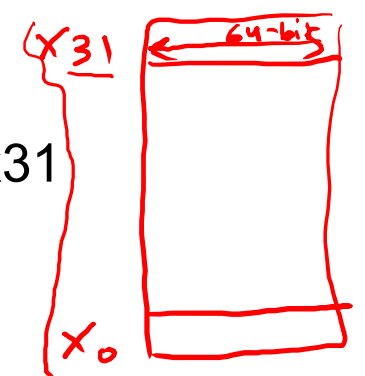
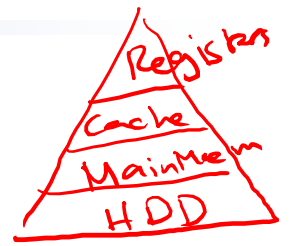
Register Operands

- 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”

$$32 = 2^5$$

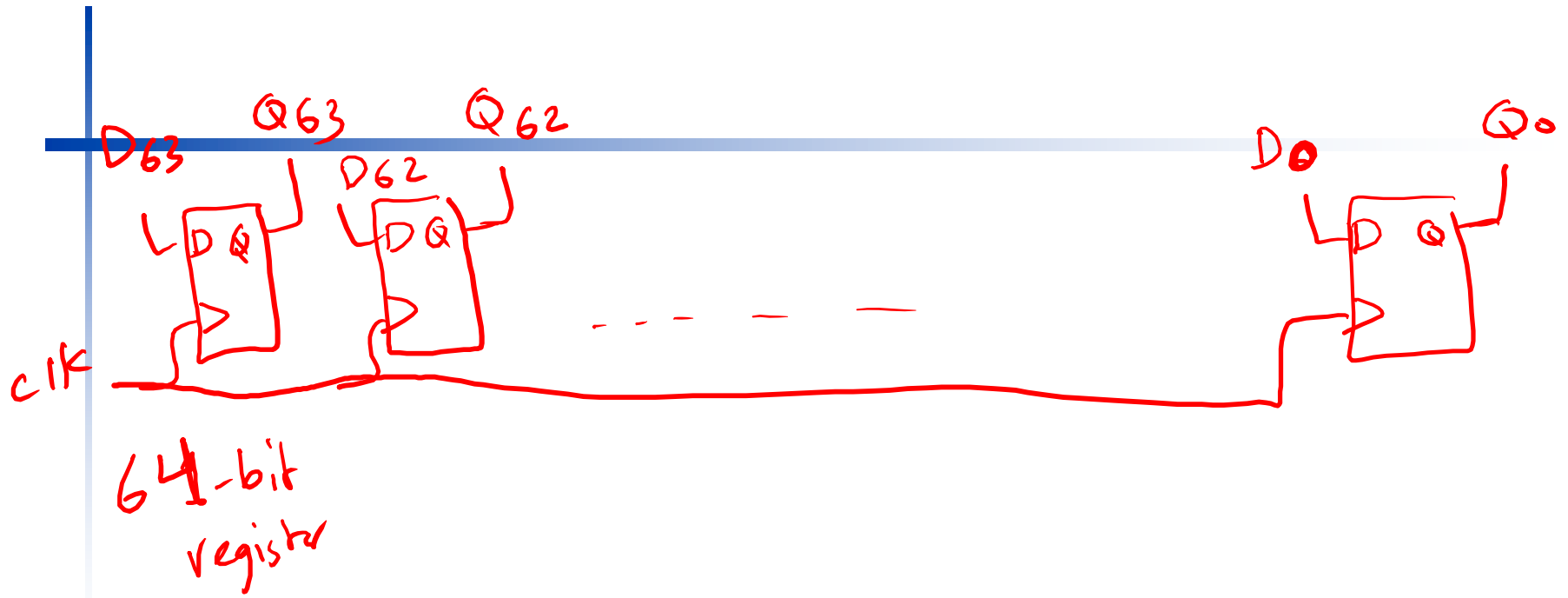


Design Principle 2: Smaller is faster

- 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

x₀ ≡ 00000
 ⋮
 x₃₁ ≡ 11111

← add a, b, c → add x₂₀, x₃₀, x₃₁
 machine code Assembly

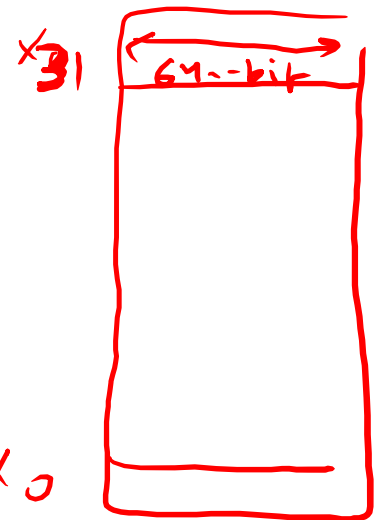


RISC-V Registers

- 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

RISC-V Fixed length
Variable length
each instruction is 32-bit

add -|-| - $\xrightarrow{\text{machine code}}$ 32-bit
sub -|-| - \longrightarrow "



RISC-V
32-bit
version

data \rightarrow 32-bit
instruction \rightarrow "

RISC-V
64-bit
version

data \rightarrow 64-bit
instructions \rightarrow 32-bit

Register Operand Example

- C code:

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

- `f, ..., j` in `x19, x20, ..., x23`

Handwritten annotations:
A red arrow points from `f` to `x19`.
A red arrow points from `g` to `x21`.
A red arrow points from `h` to `x21`.
A red arrow points from `i` to `x22`.
A red arrow points from `j` to `x23`.

- Compiled RISC-V code:

`add x5, x20, x21`

`add x6, x22, x23`

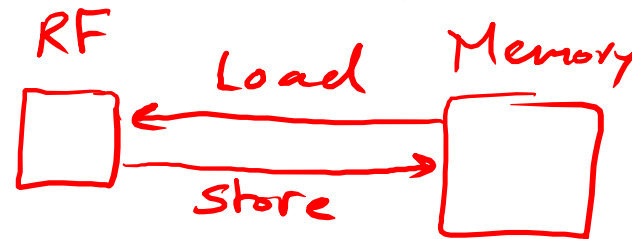
`sub x19, x5, x6`

Handwritten RISC-V code:
`add x19, x20, x21`
`sub x19, x19, x22`

Handwritten RISC-V code:
`sub x19, x19, x23`

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, ... Memory[2] = 10
 - Index used with HLL

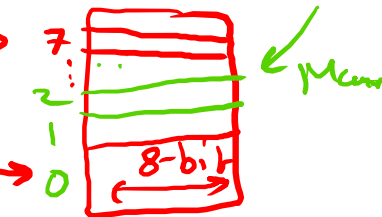
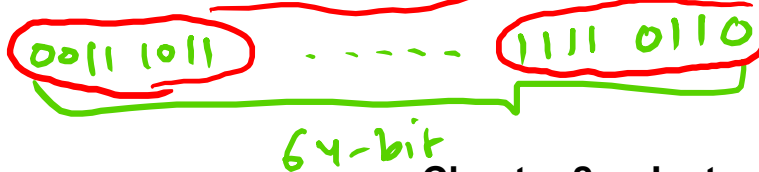
⋮	⋮
3	100
2	10
1	101
0	1
index	Data

Memory address = index × 8
 (Data Size in bytes)

- 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

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

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



$$\text{Memory address} = \underbrace{\text{starting address}} + (\underbrace{\text{index}} * \underbrace{\text{Data Size}}_{\text{in bytes}})$$

(RISC-V)

Little Endian Vs. Big Endian

64-bit data: (1A B2 0C DF 98 E5 46 8C)

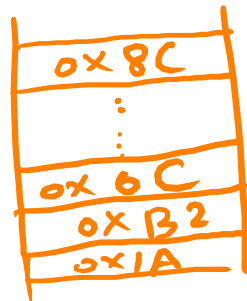
64/4 = 16 digits hexadecimal

= 0x 1A B2 0C DF 98 E5 46 8C

MSB (under 1A B2) LSB (under 46 8C)

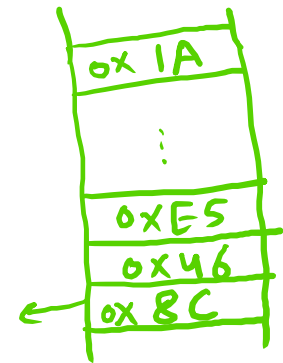
hexadecimal

Big Endian



Little Endian

1000 1100 ←



Data Transfer Instructions

- Load: copies data from a memory location to a register

$$\text{memory address} = \text{starting address} + (\text{index} * \text{Data Size in bytes}) + \text{offset}$$

- ld \equiv load doubleword

$$\text{ld } \text{x28}, \text{32}(\text{x18}) \equiv (\text{x28}) \leftarrow \text{Memory} [32 + (\text{x18})]$$

destination register

offset

source register

base register

contents of base registers represent the starting address

- Store: copies data from a register to a memory location

- sd \equiv store doubleword

$$\text{sd } \text{x28}, \text{32}(\text{x18}) \equiv \text{Memory} [32 + (\text{x18})] \leftarrow (\text{x28})$$

source register

offset

source register

base register

→ double word = 64-bit

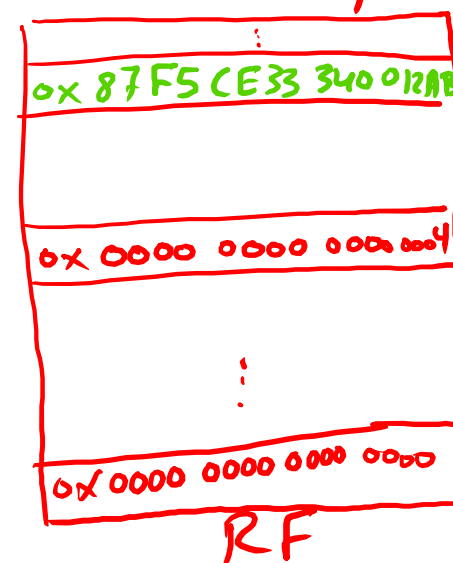
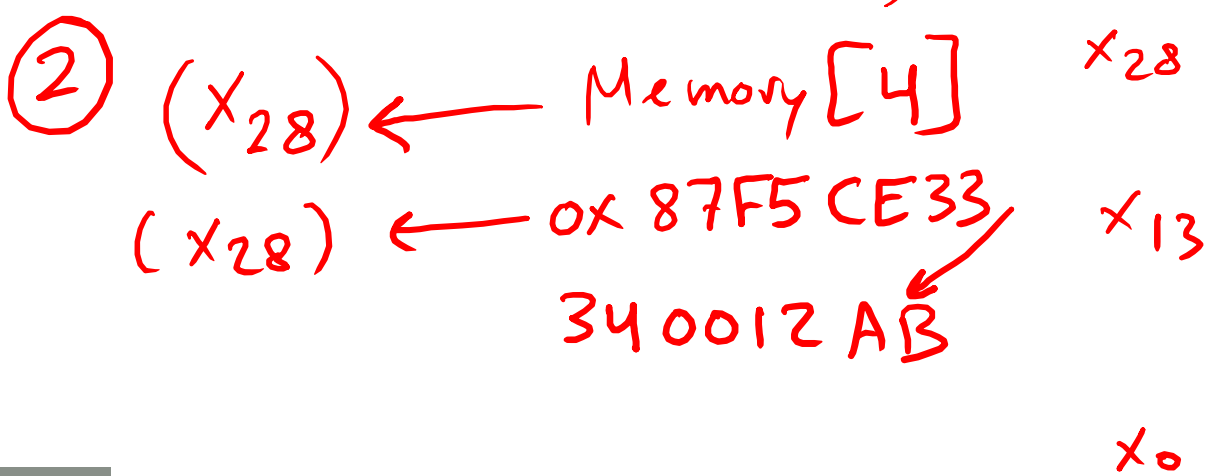
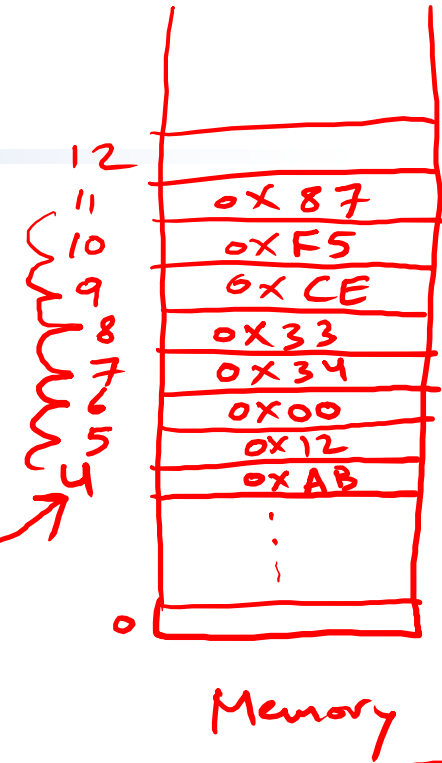
Ld X28, 0(X13)

① Memory address = offset + (contents of base register)

$$= 0 + (X_{13})$$

$$= 0 + 4$$

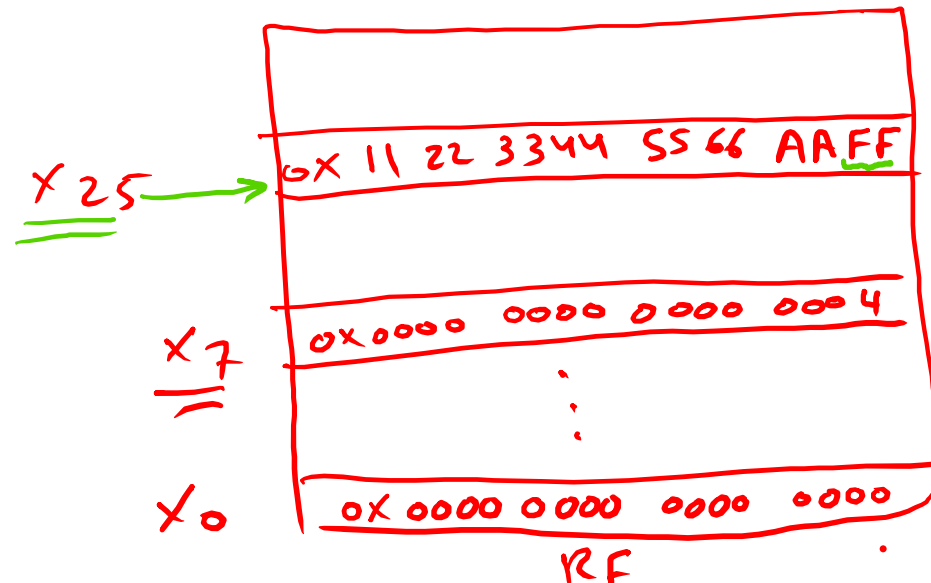
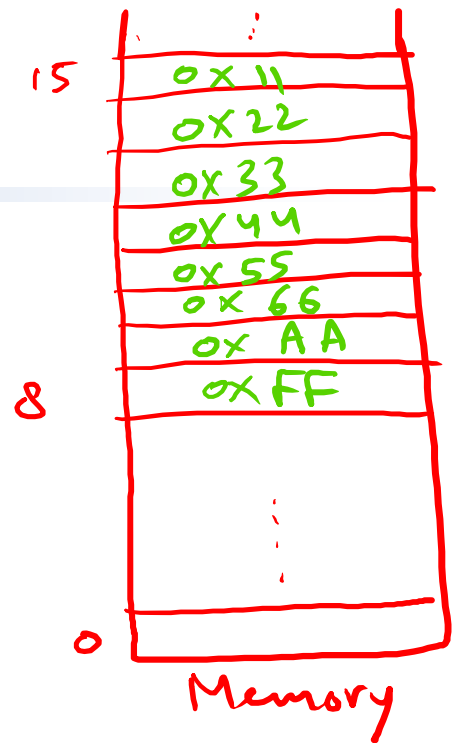
$$= 4$$



sd x25, 4(x7)

① memory address = $4 + (x7)$
 $= 4 + 4 = 8$

② Memory[8] ← (x25)



Memory Operand Example 1

- C code:
 - $g = h + A[8];$

array called "A" of type long long int 64-bit

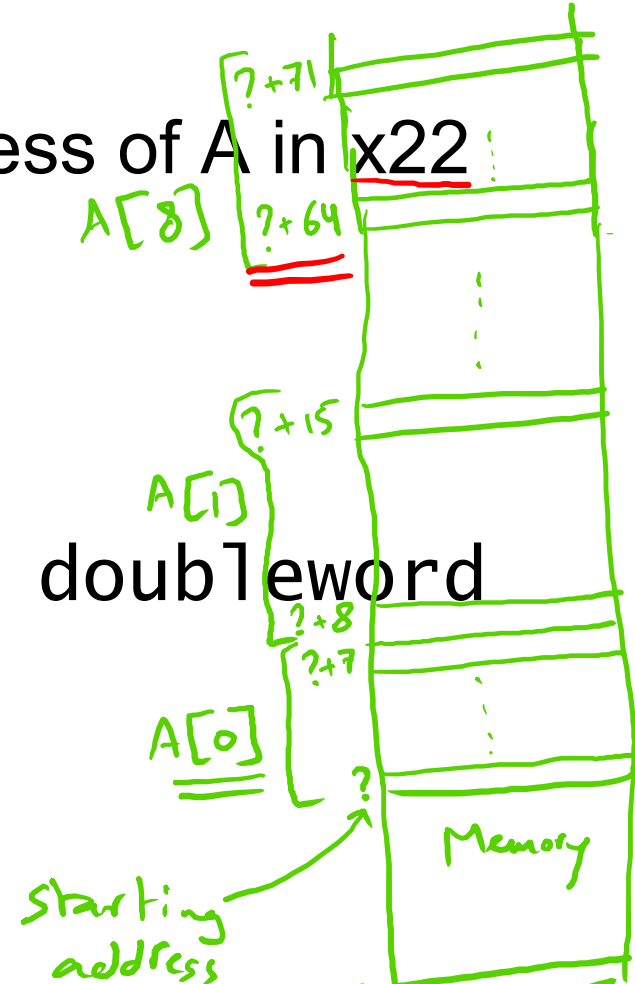
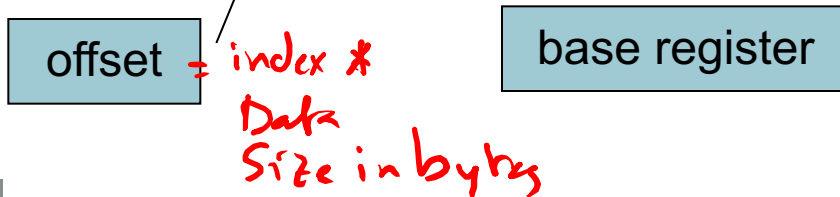
index

starting

 - 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
  
```



Memory Operand Example 2

- C code:

store $A[12] = \underline{h} + \underline{A[8]}$; *Load*

- h in x21, base address of A in x22

*array of long long
int
(64-bit)*

- Compiled RISC-V code:

- Index 8 requires offset of 64
- Index 12 requires offset of 96
 - 8 bytes per doubleword

ld x9, 64(x22) (x9) = A[8]
add x9, x21, x9 (x9) = h + A[8]
sd x9, 96(x22) (x9) = h + A[8]

$A[12] = h + A[8];$
 $\swarrow x_{21}$ \rightarrow base register of "A" is x_{22}

$a = g - A[8];$
 \downarrow \downarrow \downarrow
 x_{19} x_{20} long long int

offset = index * Data Size
 = 8 * 8
 i - bytes

Ld $x_5, 64(x_{22})$

add x_6, x_{21}, x_5

sd $x_6, 96(x_{22})$

sub x_{19}, x_{20}, x_5

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

\Rightarrow `addi x22, x22, 4`
destination source immediate
 $(x_{22}) = (x_{22}) + 4$

- No subtract immediate instruction

- Just use a negative constant

`addi x22, x22, -1`

$$(x_{22}) = (x_{22}) + (-1) = (x_{22}) - 1$$

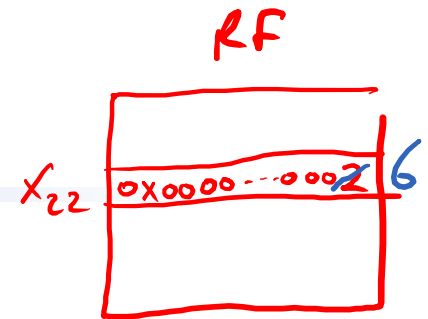
- Example of the great idea “Make the common case fast”

- Small constants are common

- More than half of the RISC-V arithmetic instructions have constants as variables

- Immediate operand avoids a load instruction

$y = z - (-7)$
 $x_{25} \quad x_{23}$
`addi x25, x23, 7`



Section 1.6
up to slide 20

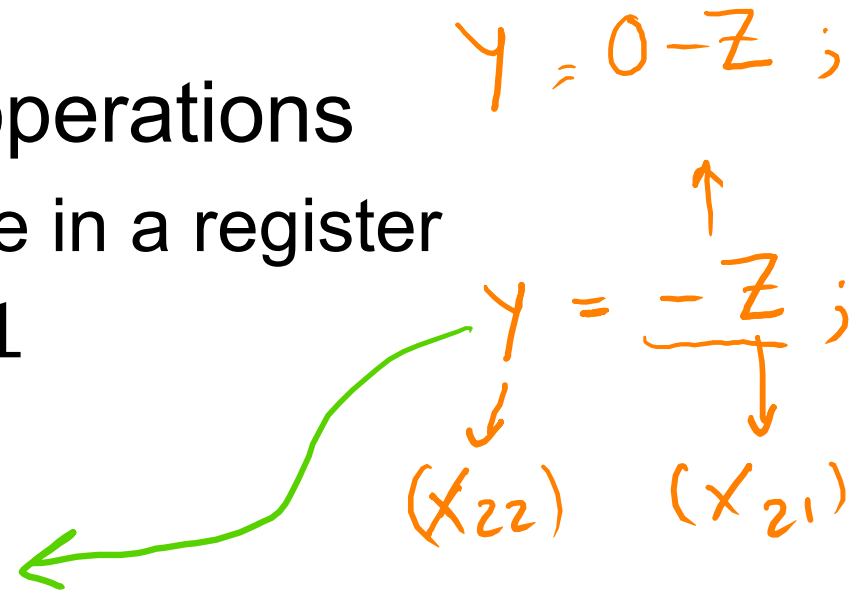
Quiz: 7%

~~Thursday~~ 29/7/2021
Wednesday 28/7/2021

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 i^{th} digit (d)

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

- i starts from 0 and increases from right to left

$(4759)_{10}$
 position 3 2 1 0

weight = r^i
 value of 7 = 7×10^2
 = 700

Unsigned Binary Integers

- Given an n-bit number

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_12^1 + x_02^0$$

- Range: 0 to $+2^n - 1$

- Example

- $0000\ 0000 \dots 0000\ 1011_2$
 $= 0 + \dots + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$
 $= 0 + \dots + 8 + 0 + 2 + 1 = 11_{10}$

- RISC-V uses 64 bits

- There are 2^{64} combinations
- 0 to $(2^{64} - 1) \rightarrow 0$ to $+18,446,774,073,709,551,615$
- Bit 0 is the least significant bit
- Bit 63 is the most significant bit

$(r=2)$ $x_{n-1} \dots x_2 x_1 x_0$ n-bit

$n=4$ $(0)_{10} = (0000)_2$

$(15)_{10} = (1111)_2$

$2^n - 1 = 2^4 - 1 = 16 - 1 = 15$

2s-Complement Signed Integers

- Given an n-bit number

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_12^1 + x_02^0$$

$x_{n-1} \equiv$ sign bit
1 (-) 0 (+)

- Range: (-2^{n-1}) to $(+2^{n-1} - 1)$

$x_{n-1} = 0 \rightarrow$ positive
 $x_{n-1} = 1 \rightarrow$ negative

- Example (32-bit number)

- $1011\ 1111\ \dots\ 1111\ 1100_2$
 $= -1 \times 2^{31} + 1 \times 2^{30} + \dots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$
 $= -2,147,483,648 + 2,147,483,644 = -4_{10}$

- Using 64 bits: $-9,223,372,036,854,775,808$
RISC-V to $9,223,372,036,854,775,807$

$n=4$ \rightarrow maximum = $2^3 - 1 = +7$
 \rightarrow minimum = $-2^3 = -8$

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

$+2^{n-1}$ X

zero
positive

$-(2^{n-1}) \rightarrow +(2^{n-1} - 1)$

- 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

$(4)_{10} = (0100)_2$

$(+4)_{10} = (0100)_2$


$-1 \times 2^{n-1} + 0 + 0 + 0 + 0$

$(1111)_2 = (-1)_{10}$
 $-1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = -4 + 2 + 1 = -1$

$(1111)_2 = (-1)_{10}$
 $-1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = -8 + 4 + 2 + 1 = -1$



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

① First approach: Complement and add 1

- Complement means $1 \rightarrow 0, 0 \rightarrow 1$

$$x + \bar{x} = (1111 \dots 11)_2 = -1$$

Handwritten note: $x + \bar{x} = -1$ with arrows pointing from the result to the terms.

Second approach:

- ① Start from the right
- ② Search for the first bit with value 1
- ③ Complement all bits to the left of the rightmost bit with value 1

$$\bar{x} + 1 = -x$$

Handwritten note: "Complement" with an arrow pointing to the bar over x.

Example: negate +2

- $+2 = 0000 \ 0000 \dots 0010_2$
- $-2 = 1111 \ 1111 \dots 1101_2 + 1$
 $= 1111 \ 1111 \dots 1110_2$

negation \equiv change the sign number

$(-5)_{10} = (11011)_2$

$(5)_{10} = (00101)_2$

Handwritten note: "negation" with an arrow pointing from the binary representation of 5 to the binary representation of -5.

Why 2s-Complement is called like this?

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

$$-x = 2^n - x$$

$$1111 \dots 1101 + 1$$

① 00100

② 1



2's Complement Representation:

MSB
(sign
bit)

-2^{n-1} to $+(2^{n-1} - 1)$

2's Complement operation:

negation in
the 2's complement

negation
 $Z \rightarrow -Z$

2's complement

11011 +
00101 +
100000

$2^6 = 64$

$$X + (-X) = \overbrace{10 \dots 0000}^{n+1}$$

$$X + (-X) = \underline{2^n}$$

$$(-X) = 2^n - X$$

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 = zero extension
- Examples: 8-bit to 16-bit
 - +2: 0000 0010 => 0000 0000 0000 0010
 - -2: 1111 1110 => 1111 1111 1111 1110 = (-2)₁₀

$(00000010)_2 = (+2)_{10}$
 $(00000010) = (+2)_{10}$
 $(0 \dots 000010) = (+2)_{10}$
 $(00001000)_2 = (8)_{10}$
unsigned $(1000)_2 = (8)_{10}$
- In RISC-V instruction set
 - 1b: sign-extend loaded byte
 - 1bu: 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!

RISC-V
↓
32-bit instruction

Hexadecimal

- Base 16

r = 16
Digits: 0, 1, ..., 9, A, B, C, D, E, F

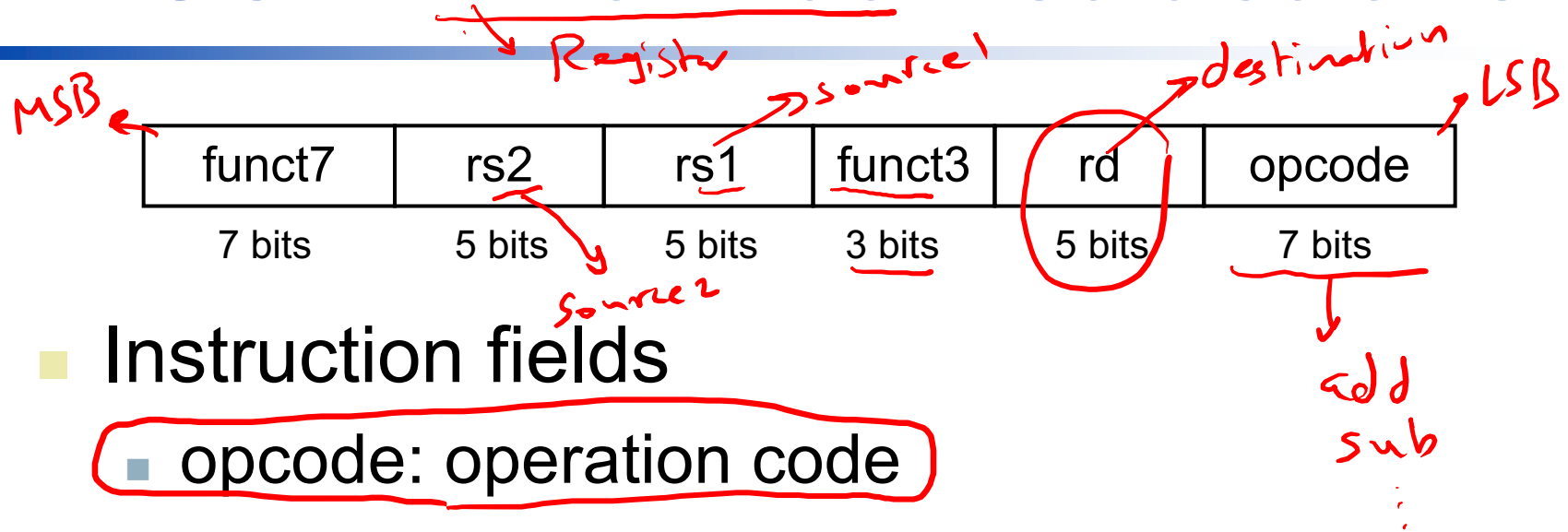
- Compact representation of bit strings
- 4 bits per hex digit

<u>0</u>	<u>0000</u>	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	<u>f</u>	<u>1111</u>

- Example: ^{0x}eca8 6420

- 1110 1100 1010 1000 0110 0100 0010 0000
e c a 8 6 4 2 0

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)

X_{31}
⋮
 X_2 RF $2^5 = 32$ 5-bit \leftarrow 00000
11111

R-format

Instr [31:0]

op code : Instr [6 : 0] 7-bit

rd : " [11 : 7] 5-bit

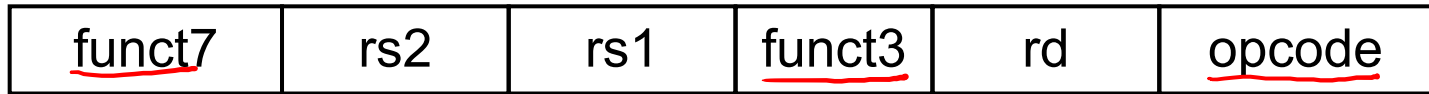
funct3 : " [14 : 12] 3-bit

rs1 : " [19 : 15] 5-bit

rs2 : " [24 : 20] 5-bit

funct7 : " [31 : 25] 7-bit

R-format Example



7 bits 5 bits 5 bits 3 bits 5 bits 7 bits

add rd x9, ^{rs1} x20, ^{rs2} x21



← decimal



Machine code

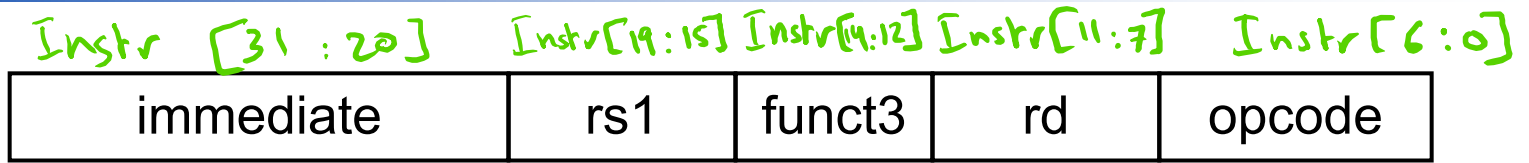
← binary

$32 + 16 + 2 + 1 = 51$

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

015A04B3₁₆

RISC-V I-format Instructions



12 bits

"addi"

5 bits

"Load"

3 bits

5 bits

7 bits

addi rd, rs1, imm
Ld rd, offset(rs1)

Immediate arithmetic and load instructions

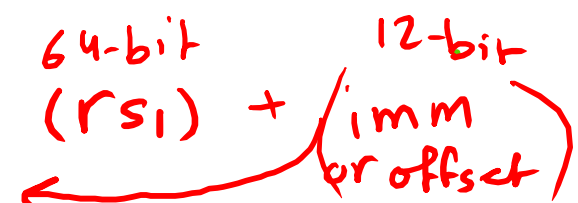
- rs1: source or base address register number
- immediate: constant operand, or offset added to base address
- 2s-complement, **sign extended**, (-2^{11}) to $(+2^{11} - 1) \equiv -2048$ to $+2047$
- The load doubleword instruction can refer to any doubleword within a region of $\pm 2^{11}$ or 2048 bytes ($\pm 2^8$ or 256 doublewords) of the base address in the base register rs1

$n-1$
 -2 to $+(2^{n-1} - 1)$
 $64\text{-bit} = 8\text{ bytes}$

$$\frac{2^{11}}{2^3} = 2^8 = 256$$

Design Principle 3: Good design demands good compromises

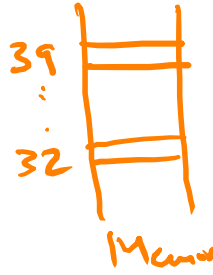
- Different formats complicate decoding, but allow 32-bit instructions uniformly
- Keep formats as similar as possible



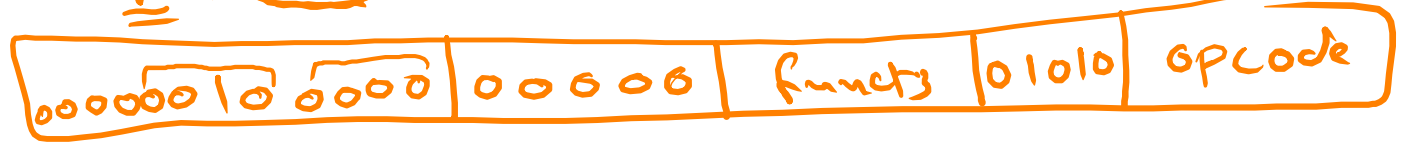
rd
 addi x₅, x₇, -16
 ↓
 -2048 to +2047



$$(x_5) = (x_7) - 16$$



Ld x₁₀, (32) (x₀) ⇒ (x₁₀) = Memory[32 + (x₀)]

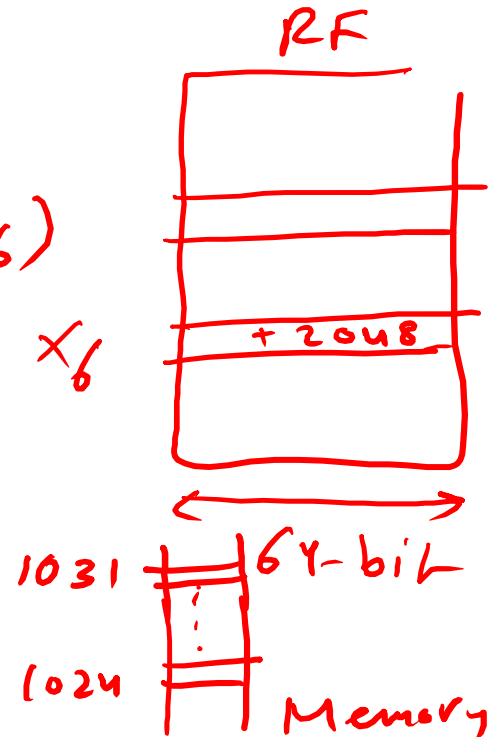


~~Ld x5, -2048(x6)~~

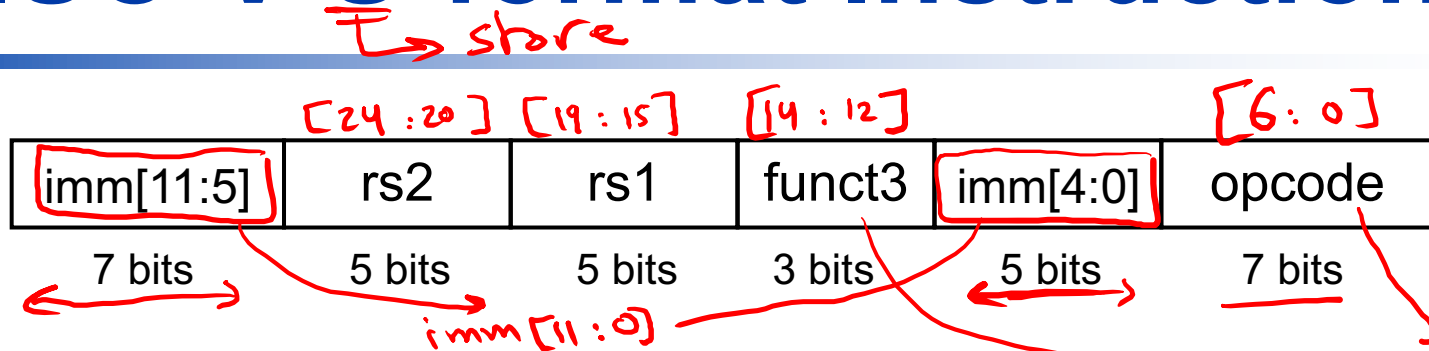
$$\text{Memory address} = -2048 + (x6) = -2048$$

✓ Ld x5, -1024(x6)

$$\begin{aligned} \text{Memory address} &= -1024 + (x6) \\ &= 1024 \end{aligned}$$



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

store
↓
S-format

sd rs2, offset(rs1)
Memory[offset + (rs1)] ← (rs2)

Instructions Opcodes

Instruction	Format	funct7	rs2	rs1	funct3	rd	opcode
add (add)	R	0000000	reg	reg	000	reg	0110011 = 51
sub (sub)	R	0100000	reg	reg	000	reg	0110011 = 51
Instruction	Format	immediate	rs1	funct3	rd	opcode	
addi (add immediate)	I	constant	reg	000	reg	0010011 = 19	
ld (load doubleword)	I	address	reg	011	reg	0000011 = 3	
Instruction	Format	immediate	rs2	rs1	funct3	immediate	opcode
sd (store doubleword)	S	address	reg	reg	011	address	0100011 = 35

- Opcode helps to distinguish which format should be used

opcode = 51 → R-format

opcode = 3, 19 → I-format

opcode = 35 → S-format

Instruction Representation Example

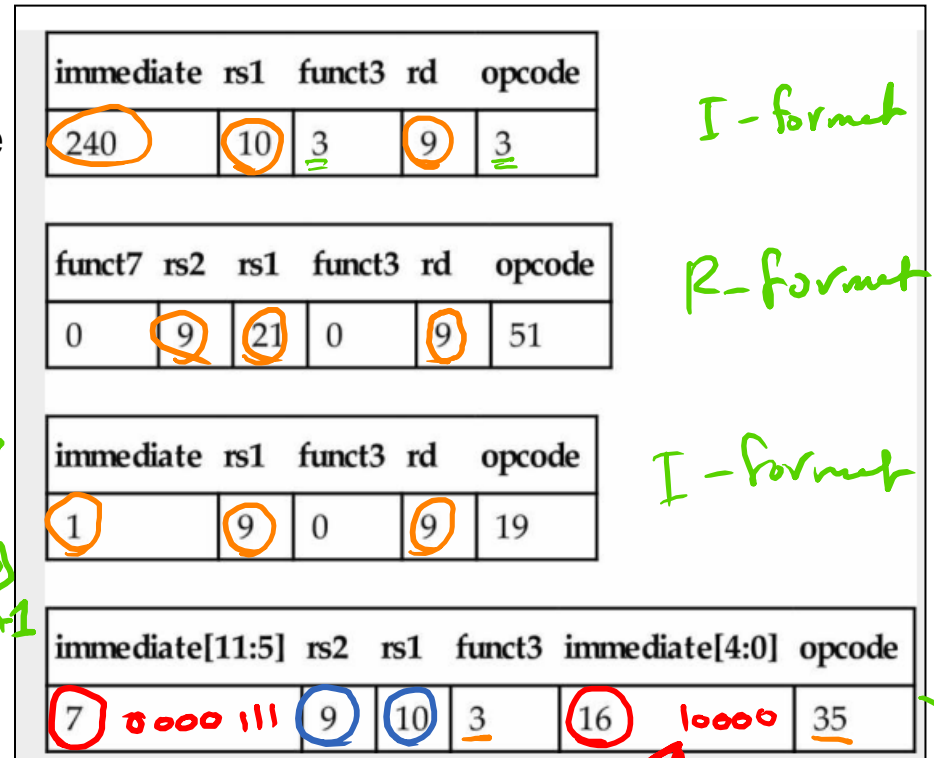
- Translate the C-statement below to RISC-V machine code:

- $A[30] = h + A[30] + 1;$
- Assume that $x10$ contains the base address of A and h is mapped to $x21$

Solution:

- First: translate to RISC-V assembly
 $offset = index \times 8$
 $ld\ x9,\ 240\ (x10)\ (x9) = A[30]$
 $add\ x9,\ x21,\ x9\ (x9) = h + A[30]$
 $addi\ x9,\ x9,\ 1\ (x9) = h + A[30] + 1$
 $sd\ x9,\ 240\ (x10)$

- Second: translate to RISC-V machine code

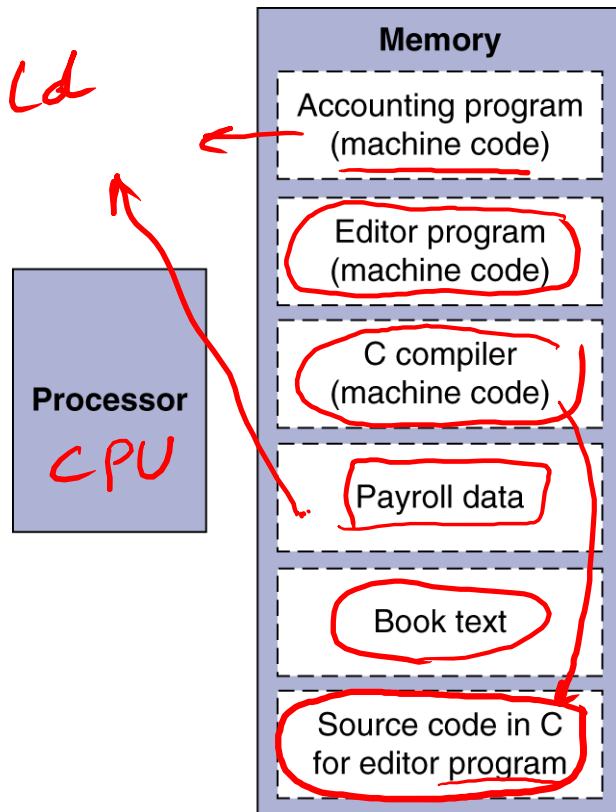


240
128
112
64
48
16

7 6 5 4 3 2 1 0
0 0 0 0 1 1 1 1 0 0 0 0

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

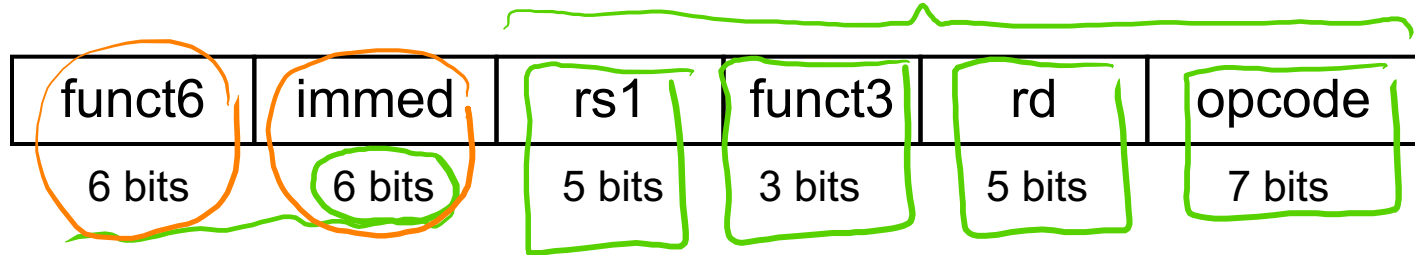
Operation	<u>C</u>	<u>Java</u>	<u>RISC-V</u>
Shift left	<<	<<	<i>shift left</i> <u>sll, slli</u>
Shift right	>>	>>>	<i>logical</i> <u>sr1, srl</u>
Bit-by-bit AND	&	&	<i>right</i> <u>and, andi</u>
Bit-by-bit OR			<u>or, ori</u>
Bit-by-bit XOR	^	^	<u>xor, xori</u>
<u>Bit-by-bit NOT</u>	~	~	<u>Xori</u>

add and

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

Shift Immediate Operations

R-format \equiv I-format = Shift format



- immed: how many positions to shift
- Shift left logical
 - Shift left and fill with 0 bits
 - $sll\ i$ by i bits multiplies by 2^i
- Shift right logical
 - Shift right and fill with 0 bits
 - $srl\ i$ by i bits divides by 2^i (unsigned only)

000000 = (0)₁₀
⋮
111111 = (63)₁₀

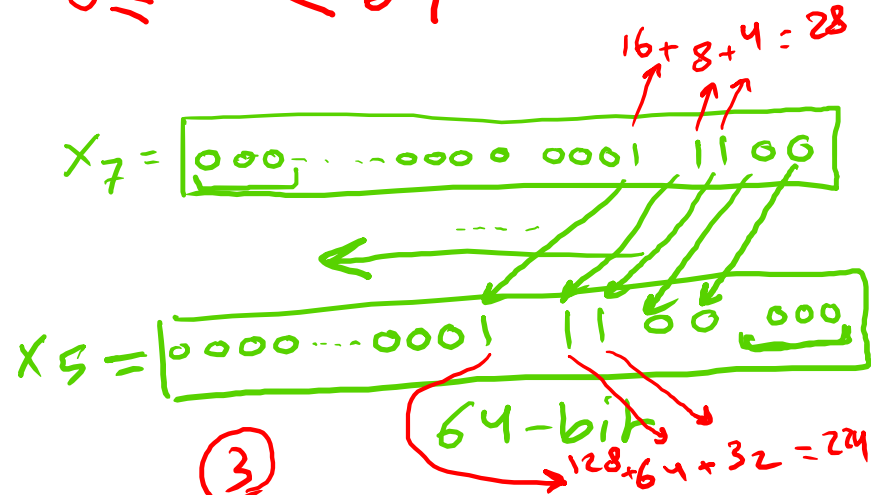
slli rd, rs1, constant

↓
shift left logical immediate

slli x5, x7, 3

↓
stays the same

$$0 \leq < 64$$



$$224 = 28 \times 8$$

$$= 28 \times 8 = 224$$

$$g = h * 16 \Rightarrow$$

Annotations: $g \rightarrow x_{23}$, $h \rightarrow x_{22}$, $16 \rightarrow 2^4$

slli x23, x22, 4

slli rd, rs1, 64 \Rightarrow (rd) = 0

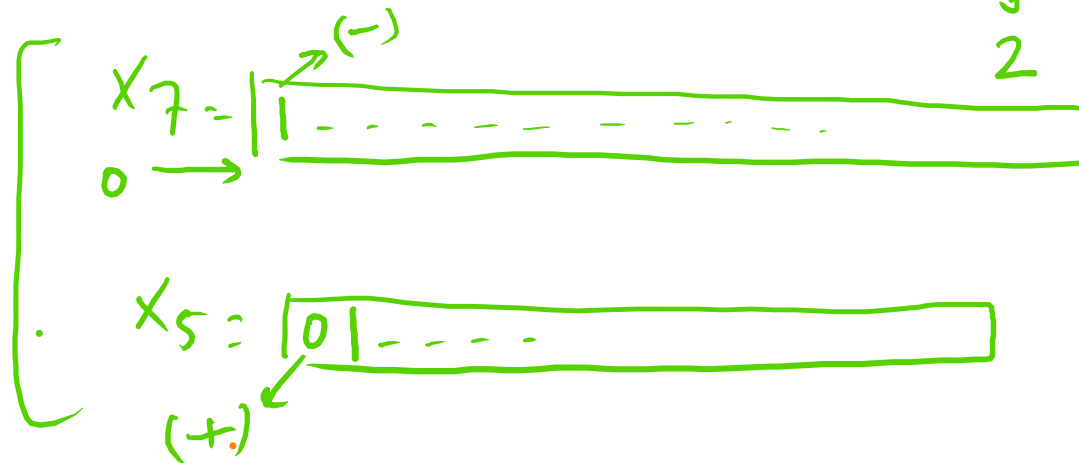
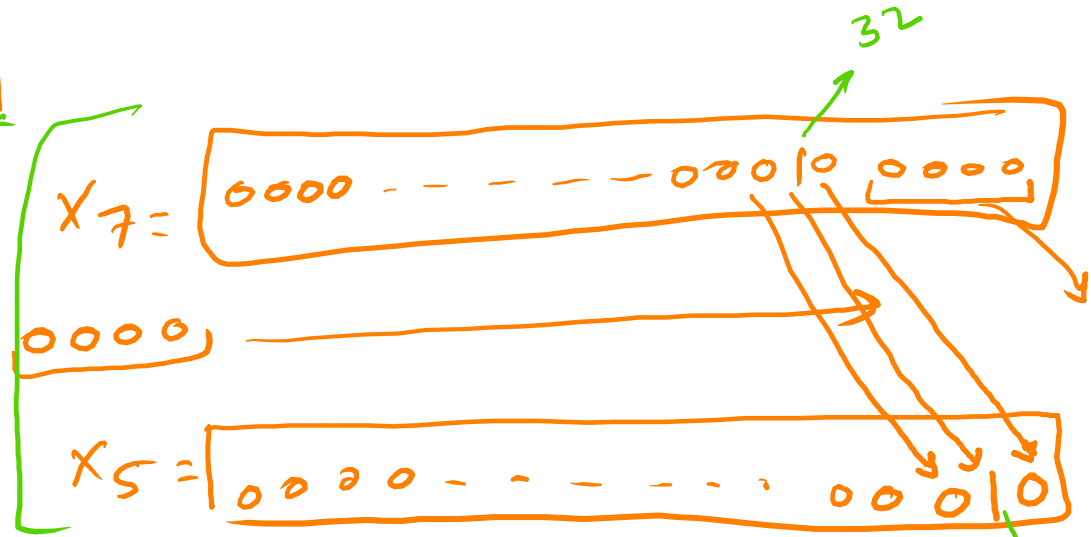
{ add rd, x0, x0
 { addi rd, x0, 0



srl rd, rs1, constant

srl x5, x7, 4

$$32 / 2^4 = 2$$



$\text{sll rd, rs}_1, \text{rs}_2$
 srl " , " , " } R-format

(rs_2) represents the shift amount

$\text{sll x}_5, \text{x}_7, \text{x}_6$

$\equiv \text{slli x}_5, \text{x}_7, 8$

$$x_6 = \boxed{0 \text{ --- } 00001000}$$
$$= (8)_{10}$$

arithmetic shift : The inserted bits equal the sign bit of rs_1

$\text{sla rd, rs}_1, \text{rs}_2$

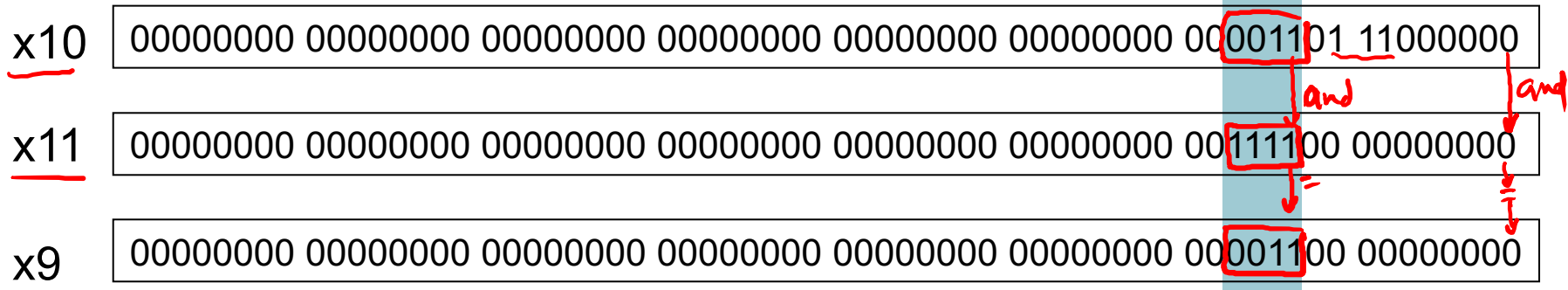
sra

AND Operations

Bitwise AND operation

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

and rd x9, ^{rs1} x10, ^{rs2} x11 *R-format*



- There is an AND Immediate instruction (andi) *I-format*

- Sign-extension for the immediate value

andi rd x9, ^{rs1} x10, ^{imm(12-bit)} 0xFFF *hexadecimal*

x10 = [64-bit register]

(x9) = (x10)

64-bit

||| - - - ||| | ||| | ||| | |||

addi x9, x10, 0x00F

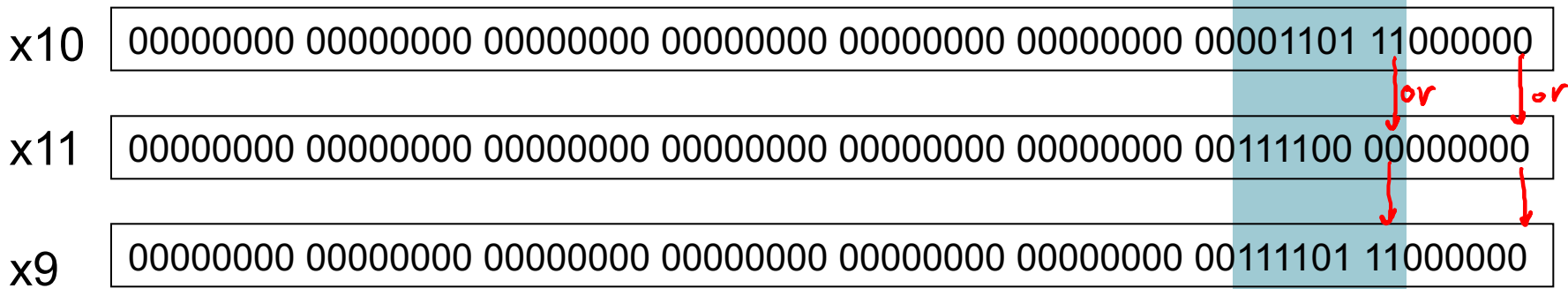


OR Operations

"Bitwise operation"

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

or x9, x10, x11 *R-format*
rd rs1 rs2



- There is an OR Immediate instruction (ori) *I-format*
 - Sign-extension for the immediate value *64-bit*

ori x9, x10, 0xFFFF
rd rs1 imm



XOR Operations

"Bitwise"

$$1 \oplus X = \overline{X}$$

$$0 \oplus X = X$$

- Differencing operation

- Invert some bits, leave others unchanged

```
addi x11, x0, 0xFFF
```

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

rd r3, rs2 R-format

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

$$(X_9) = \overline{(X_{10})}$$

- There is an XOR Immediate instruction (xori)

I-format

- Sign-extension for the immediate value

```
xori x9, x10, 0xFFFF => (X9) = overline(X10)
```


Conditional Operations

- Branch to a labeled instruction if a condition is true

- Otherwise, continue sequentially

branch if equal

- beq ^{x5}rs1, ^{x10}rs2, L1

- if (rs1) == (rs2) branch to instruction labeled L1

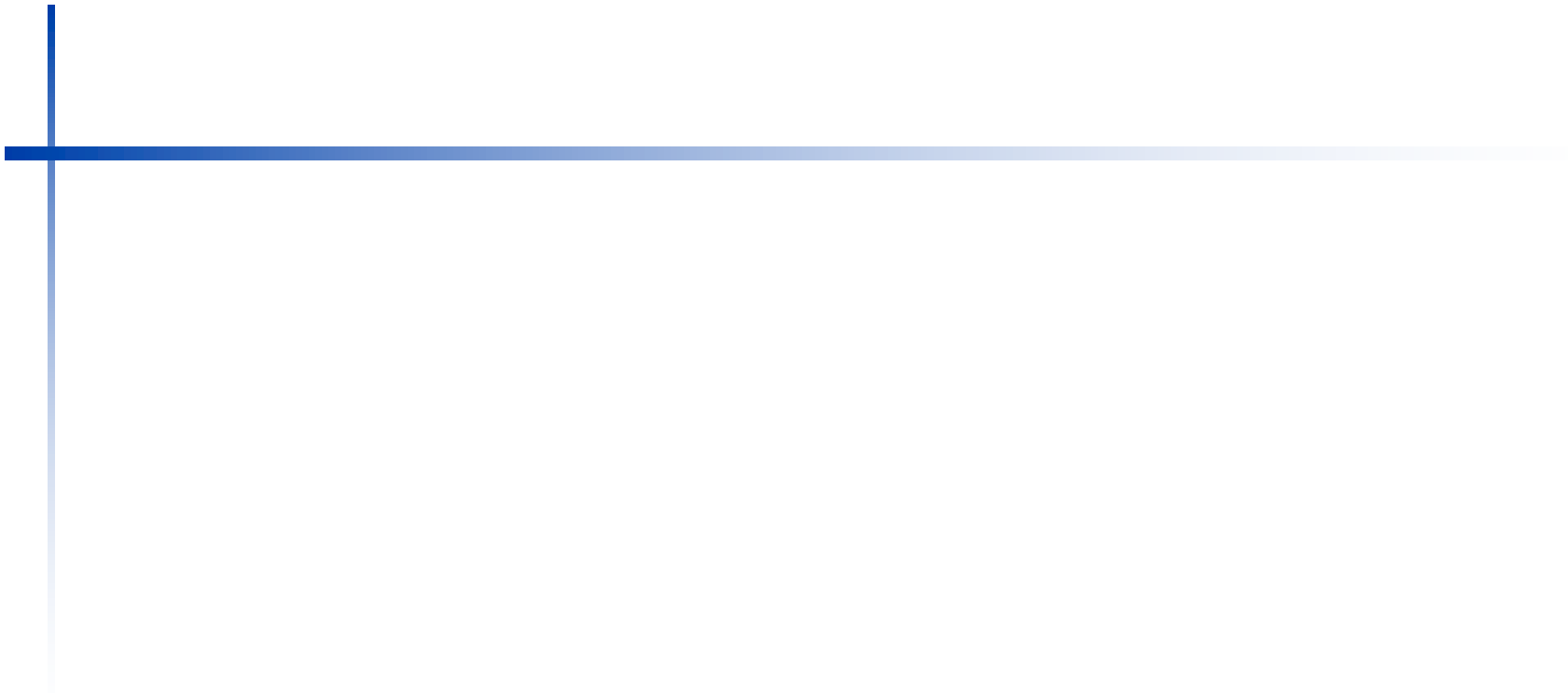
otherwise continue sequentially

branch if not equal

- bne rs1, rs2, L1

- if (rs1) != (rs2) branch to instruction labeled L1





Compiling If Statements

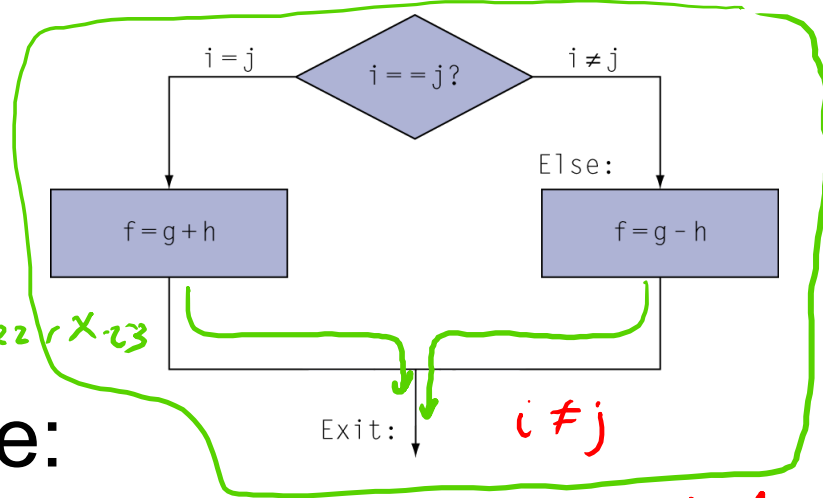
- C code:

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

- f, g, h, i, j in x19, x20, x21, x22, x23

- Compiled RISC-V code:

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



Handwritten RISC-V code annotations:

```
beq x22, x23, L1
sub x19, x20, x21
beq x0, x0, Exit
L1: add x19, x20, x21
Exit: - - - - -
```

Assembler calculates addresses

beq —, —, LI ⇒ X
LI: - - - -

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) (x9) = save[i]

bne x9, x24, Exit

addi x22, x22, 1

beq x0, x0, Loop

Exit: ...

beq rs1, rs2, L1
bne rs1, rs2, L1

ld x5, x10 (x25)

offset = i * 8

slli x10, x22, 3

offset

beq x9, x24, L1
beq x0, x0, Exit
memory: offset + base register
address

L1: addi x22, x22, 1
beq x0, x0, Loop = (x10) + (x25)

Exit: ...

More Conditional Operations

- b<t rs1, rs2, L1 *branch if less than*
 - if $(rs1) < (rs2)$ branch to instruction labeled L1
- bge rs1, rs2, L1 *branch if greater than or equal*
 - if $(rs1) \geq (rs2)$ branch to instruction labeled L1

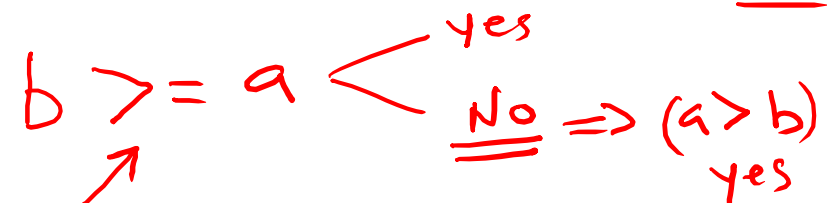
Example

- if $(a > b)$ $a += 1$;
- a in x22, b in x23

bge x23, x22, Exit // branch if $b \geq a$

↓
addi x22, x22, 1

Exit:



blt x23, x22, L1
 bge x0, x0, Exit

L1: 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 **blt x22, x23, L1**
"true"
 - $-1 < +1$
- $x22 > x23$ // unsigned **bltu x22, x23, L1**
"false"
 - $+4,294,967,295 > +1$

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.

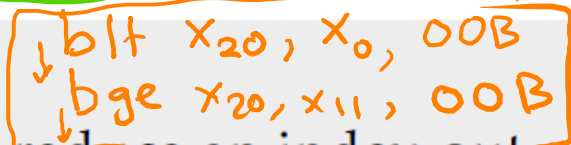
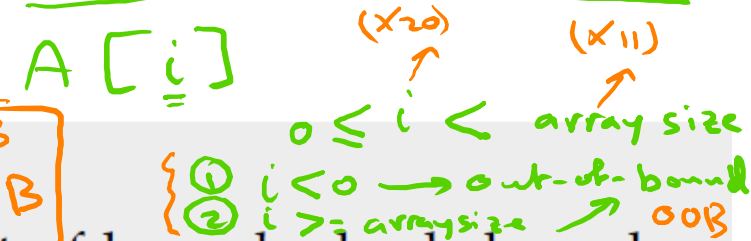
Example

Use this shortcut to reduce an index-out-of-bounds check: branch to `IndexOutOfBounds` if $x_{20} \geq x_{11}$ or if x_{20} 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
```



$x_{20} \geq 0$ AND $x_{20} < x_{11}$
continue sequentially

Compiling Loop Statements

- For Loop (C code):

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

- Case/Switch:

① Simplest way is via sequence of conditional tests →
chain of if-else statements

② 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)

RISC-V Assembly

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

store

load
shift left

$i \rightarrow x_{20}$
Base address of array save
is in x_{10}

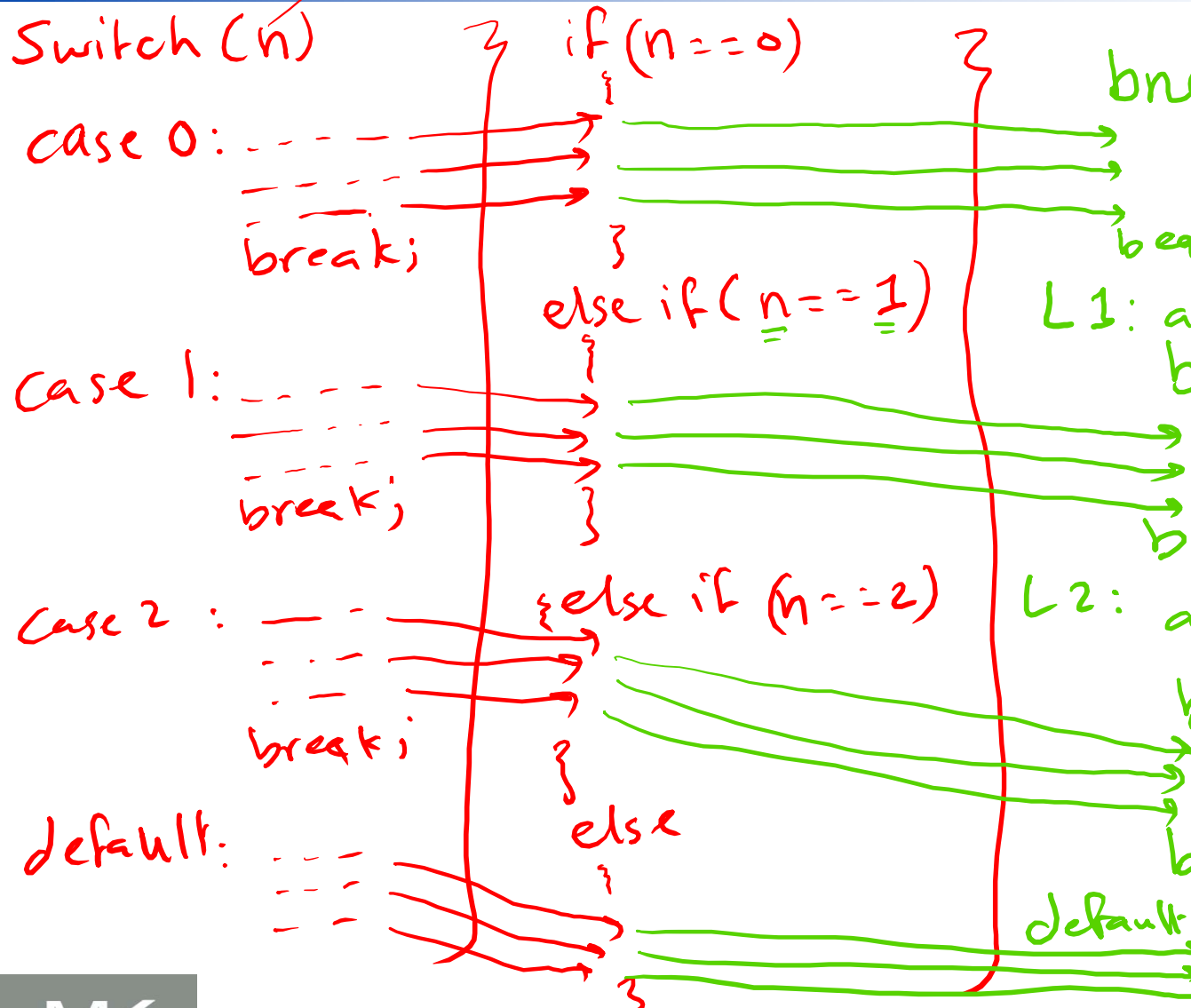
$$(x_6) = i * 8$$
$$(x_6) = (x_6) + (x_{10})$$
$$(x_7) = \text{save}[i]$$

```
Loop: blt x20, x5, L1  
      beq x0, x0, Exit  
L1: slli x6, x20, 3
```

```
Loop: bge x20, x5, Exit  
      slli x6, x20, 3  
      add x6, x6, x10  
      ld x7, 0(x6)  
      slli x7, x7, 1  
      sd x7, 0(x6)  
      addi x20, x20, 1  
      beq x0, x0, Loop  
Exit: . . . . .
```



Switch (n) $\xrightarrow{x20}$



RISC-V Assembly
`bne x20, x0, L1`

`beq x0, x0, Exit`

`L1: addi x5, x0, 1`
`bne x20, x5, L2`

`beq x0, x0, Exit`

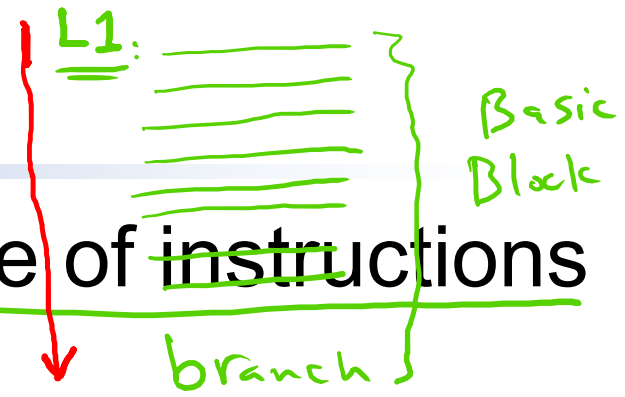
`L2: addi x5, x0, 2`
`bne x20, x5, default`

`beq x0, x0, Exit`

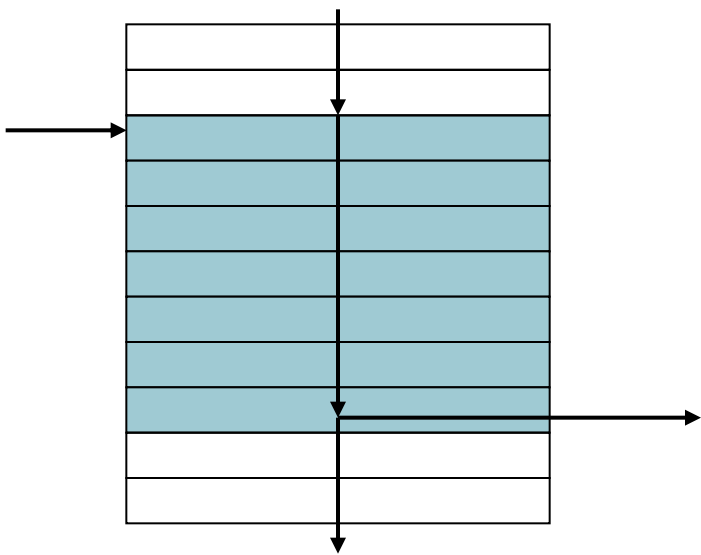
default:



Basic Blocks



- A basic block is a sequence of instructions with:
 - ① No embedded branches (except at end)
 - ② 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 \Rightarrow x10, x11
6. Return to place of call (address in x1)

```

long long int Summation (int x,
                          int y,
                          int z);
{
  long long int sum;
  sum = x + y + z;
  return sum;
}

```

Procedure Call Instructions

- Procedure call: jump and link *jal rd, Label*

✓ *jal x1, ProcedureLabel*

- ① Address of following instruction (PC+4) put in x1 *return address register*
- ② Jumps to target address

- Can also be used for unconditional branch *beq x0, x0, Label*
 - e.g., *jal x0, Label* (x0 cannot be changed)

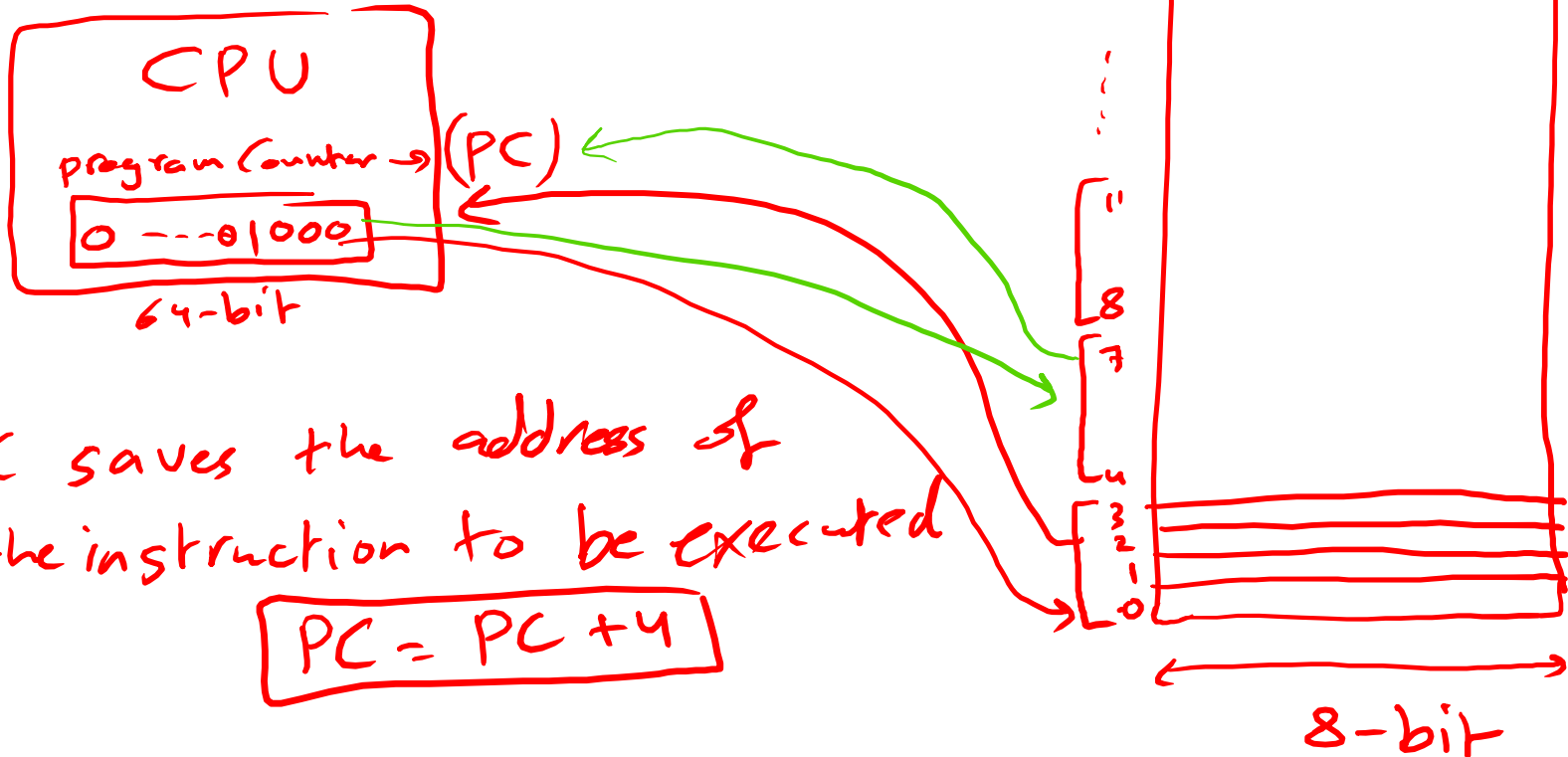
- Procedure return: jump and link register *jump*

jalr x0, 0(x1) *jalr rd, offset(rs1)* *x0 ≠ PC+4*

- Like jal, but jumps to 0 + address in x1
- Use x0 as rd (x0 cannot be changed) *① (rd) = PC+4*
- Can also be used for computed jumps *② jump to memory*
 - e.g., for case/switch statements *address = offset + (rs1)*

Instruction Memory (IM)

Instructions are encoded using
32-bit



PC saves the address of
the instruction to be executed

$$PC = PC + 4$$

(PC=0) add

(PC=4) sub

(PC=8) ld

⋮

(PC=60) jal rd, Label

(PC=64)



$$(rd) = PC + 4$$

$$(rd) = 60 + 4 = 64$$

$$(X1) = 64$$

Using More Registers: Stack

- Stack is needed to:

① Spill registers during procedure execution

- More registers are needed for execution other than the eight argument registers

② Saving return address or arguments (Non-leaf procedures)

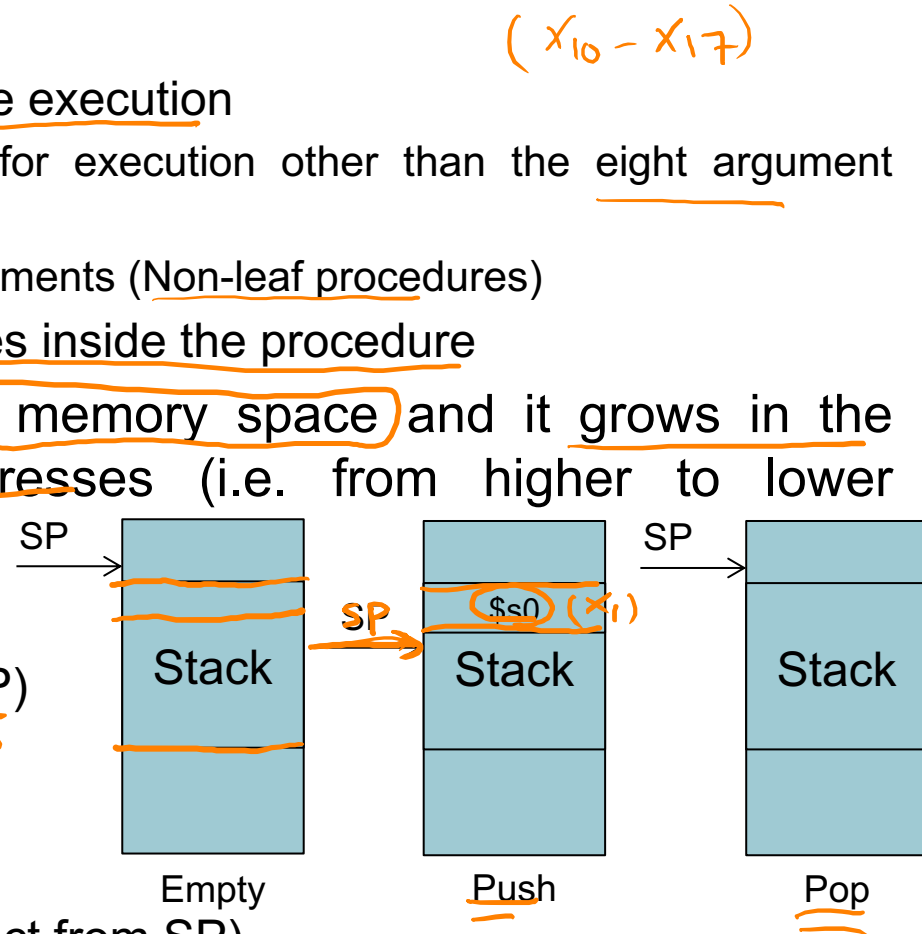
Define local arrays or structures inside the procedure

- 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 $x_2 = SP$
- 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;  
}
```

leaf
Non-leaf

return type

Procedure name

body of procedure

equals $x_{10} - x_{17}$

- Arguments g, h, i, j in x_{10}, \dots, x_{13}
- f in x_{20}
- temporaries x_5, x_6
- Result in x_{10}

return ((g+h) - (i+j));

x_{10}, x_{11}

Leaf Procedure Example

RISC-V code:

PC leaf_example:

100 addi sp, sp, -24

104 sd x5, 16(sp)

108 sd x6, 8(sp)

112 sd x20, 0(sp)

116 add x5, x10, x11

120 add x6, x12, x13

124 sub x20, x5, x6

128 addi x10, x20, 0

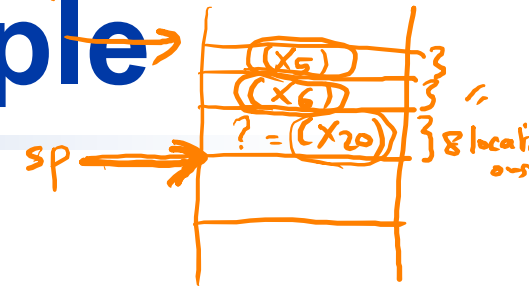
132 ld x20, 0(sp)

136 ld x6, 8(sp)

140 ld x5, 16(sp)

144 addi sp, sp, 24

148 jalr x0, 0(x1)



Save x5, x6, x20 on stack *push*

body of procedure
 $x5 = g + h$
 $x6 = i + j$
 $f = x5 - x6$
 copy f to return register

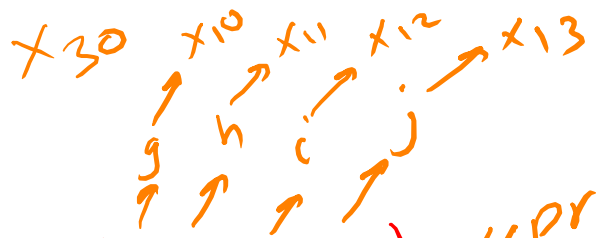
Restore x5, x6, x20 from stack

Return to caller

PC = 0 + 20 = 20

IM

main:
 long long int z;
 z = leaf-example(5, 7, 3, 4); // procedure call



Assembly

addi sp, sp, -24
 100

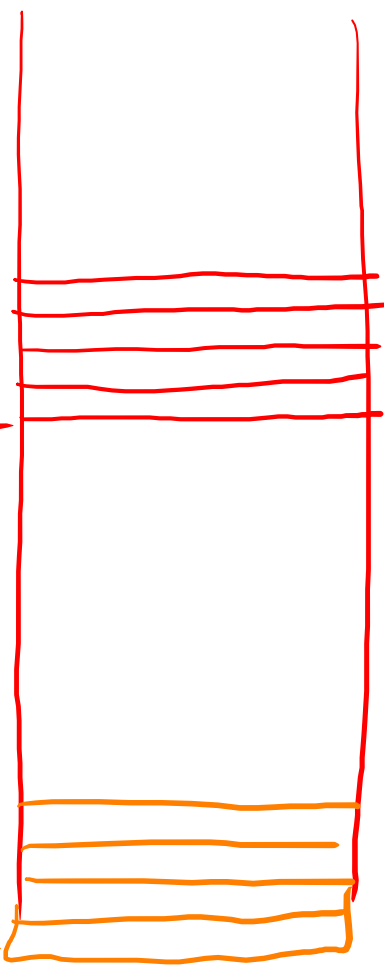
```

PC
0 addi x10, x0, 5
4 " x11, x0, 7
8 " x12, x0, 3
12 " x13, x0, 4
16 jal x1, leaf-example
20 addi x30, x10, 0

```

PC [16] 100 + 48 = 20
 x1 [20]

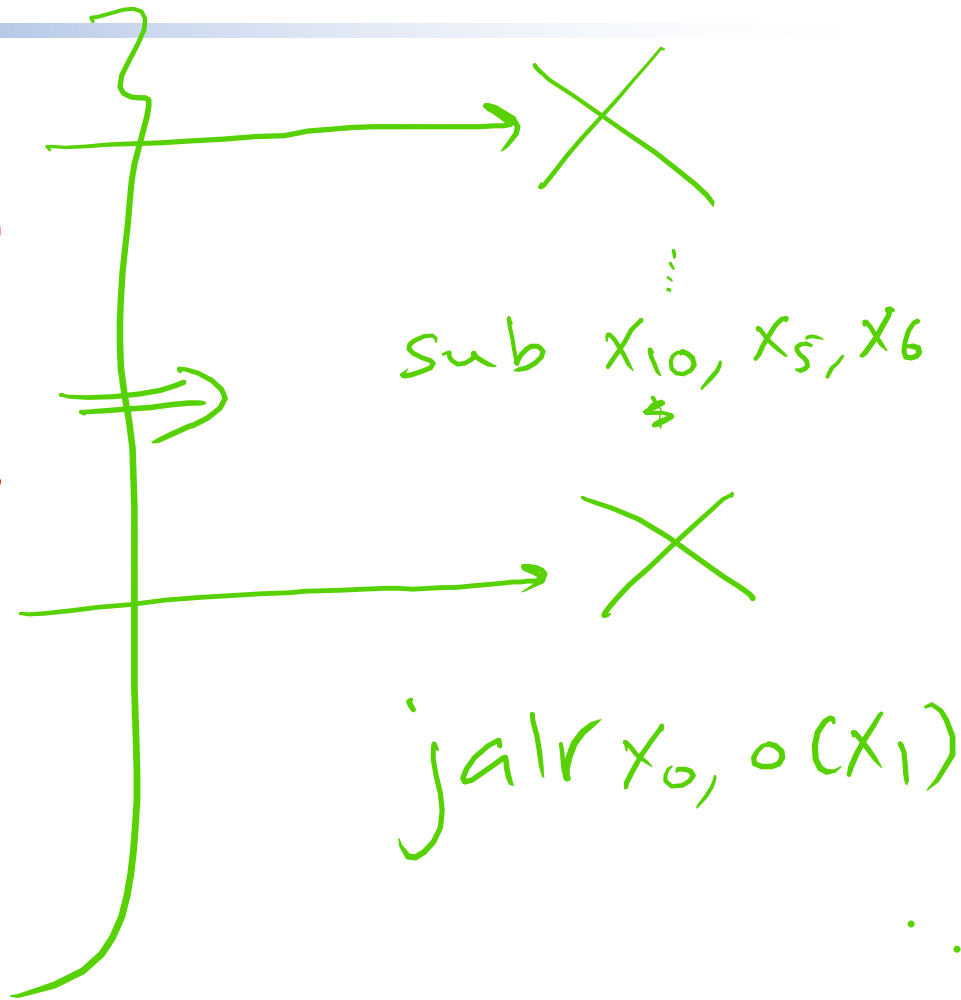
addi x10, x0, 5
 0



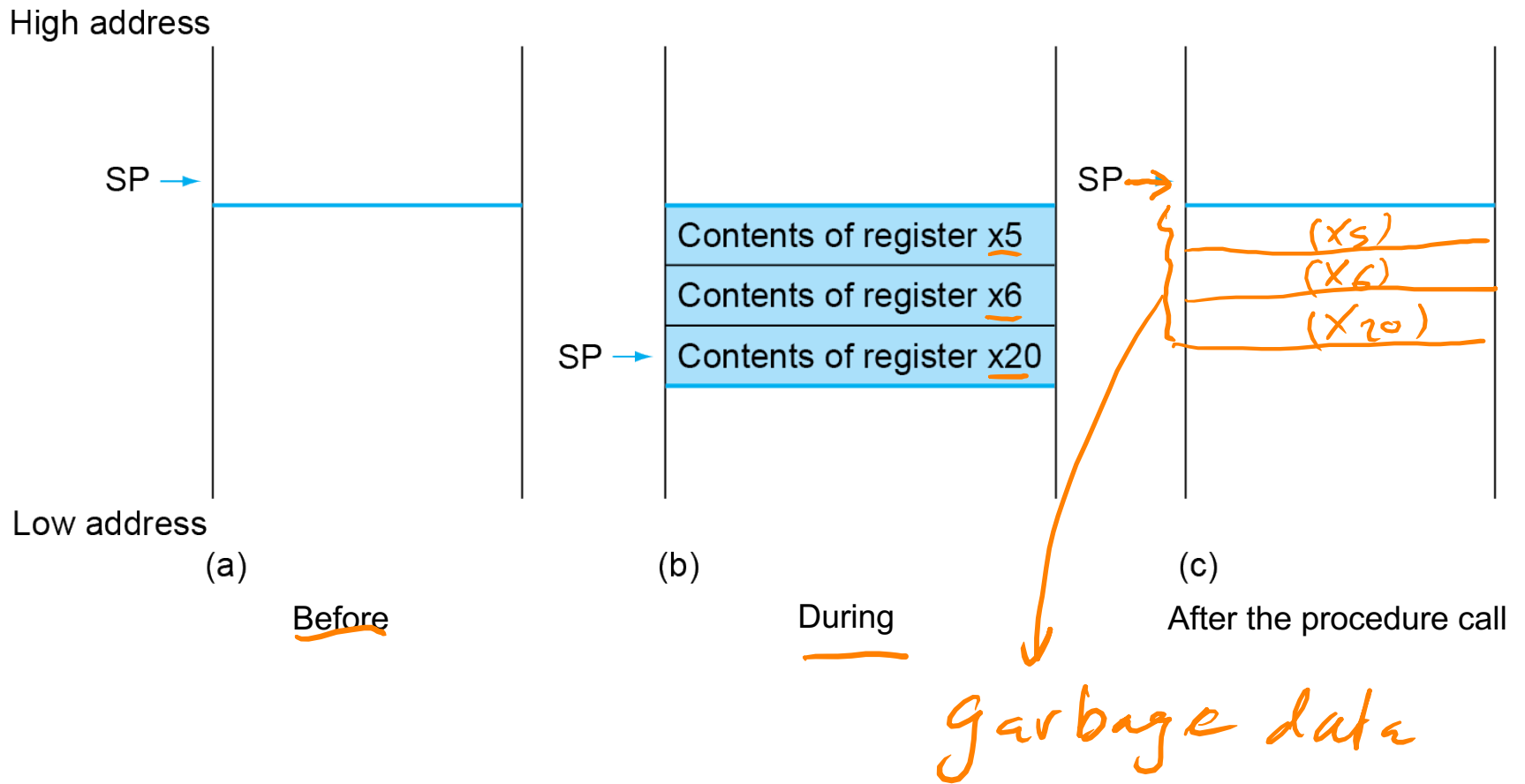
```
addi sp, sp, -8  
sd x20, 0(sp)]
```

body of procedure
sub x20, x5, x6
addi x10, x20, 0

```
ld x20, 0(sp)  
addi sp, sp, 8  
jalr x0, 0(x1)]
```



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:
 - ② ■ Its return address
 - ③ ■ Any arguments and temporaries needed after the call
- Restore from the stack after the call

Non leaf procedures

- "saved" registers
- X1
- arguments & temporaries

leaf procedures: ^{that}

- "saved" registers are written in the procedure [↑] need to be saved in the stack

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);  
}
```

recursive procedure

body of the procedure

nested call

Multiplication

- Argument n in x10
- Result in x10

x₁₀ must be saved in the stack

always x₁ must be saved in the stack

$n!$ \equiv factorial of n

$$n! = n * (n-1) * (n-2) * \dots * 1 = n * (n-1)! \\ = n * (n-1) * (n-2)!$$

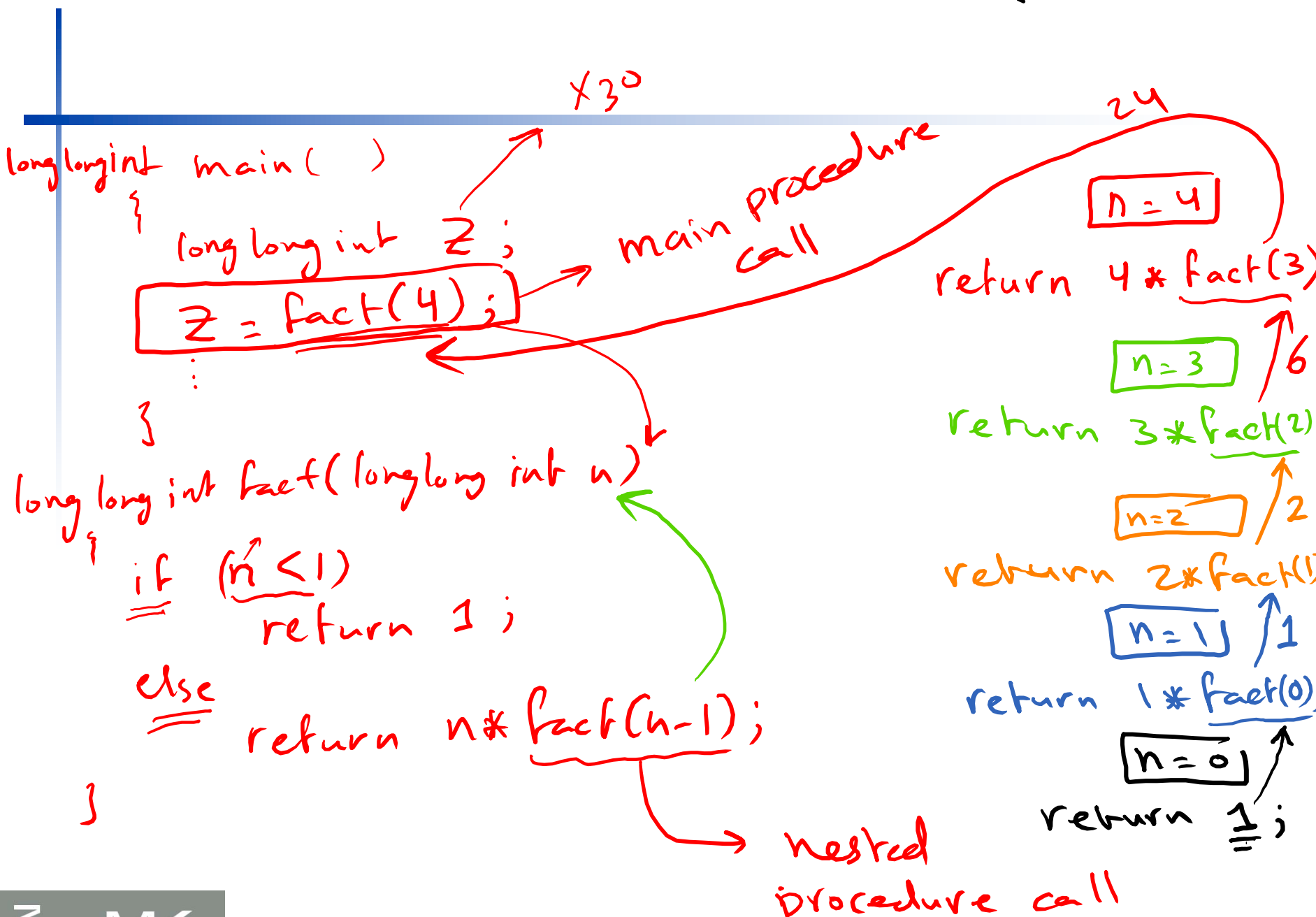
$$1! = 1$$

$$2! = 2 * 1 = 2$$

$$3! = 3 * 2 * 1 = 6$$

$$4! = 4 * 3! = 4 * 6 = 24$$

$$0! = 1$$



Non-Leaf Procedure Example

RISC-V code:

fact:

addi sp, sp, -16

sd x1, 8(sp)

sd x10, 0(sp)

addi x5, x10, -1

bge x5, x0, L1

addi x10, x0, 1

addi sp, sp, 16

jalr x0, 0(x1)

L1: addi x10, x10, -1

jal x1, fact

addi x6, x10, 0

ld x10, 0(sp)

ld x1, 8(sp)

addi sp, sp, 16

mul x10, x10, x6

jalr x0, 0(x1)

Save return address and n on stack

x5 = n - 1

if n >= 1, go to L1

Else, set return value to 1

Pop stack, don't bother restoring values

Return

n = n - 1

call fact(n-1)

move result of fact(n - 1) to x6

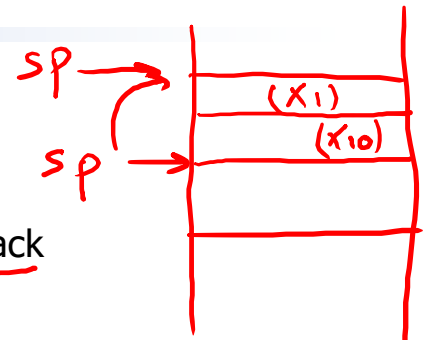
Restore caller's n

Restore caller's return address

Pop stack

return n * fact(n-1)

return



$n < 1 \equiv n - 1 < 0$
opposite
 $n - 1 \geq 0$

$n * \text{fact}(n-1)$

$(x_{10}) = \text{fact}(n-1)$

$n * \text{fact}(n-1)$

$(x_6) = \text{fact}(n-1)$

Save on stack

body

Else

Restore from stack return

addi x5, x0, 1
bge x5, x0, L1

IF

Multiply

fact:

```
addi x5, x10, -1
```

```
bge x5, x0, Else
```

```
addi x10, x0, 1
```

```
jalr x0, 0(x1)
```

Else:

```
[addi sp, sp, -16  
sd x1, 8(sp)  
sd x10, 0(sp)
```

```
addi x10, x10, -1
```

```
jal x1, fact
```

```
addi x6, x10, 0
```



```
Ld x1, 8(sp)
```

```
Ld x10, 0(sp)
```

```
addi sp, sp, 16
```

```
mul x10, x10, x6
```

```
jalr x0, 0(x1)
```

```

PC fact:
100 addi sp, sp, -16
104 sd x1, 8(sp)
108 sd x10, 0(sp)
112 addi x5, x10, -1
116 bge x5, x0, L1
120 addi x10, x0, 1
124 addi sp, sp, +16
128 jalr x0, 0(x1)

132 L1: addi x10, x10, -1
136 jal x1, fact
140 addi x6, x10, 0
144 ld x1, 8(sp)
148 ld x10, 0(sp)
152 addi sp, sp, 16
156 mul x10, x10, x6
160 jalr x0, 0(x1)

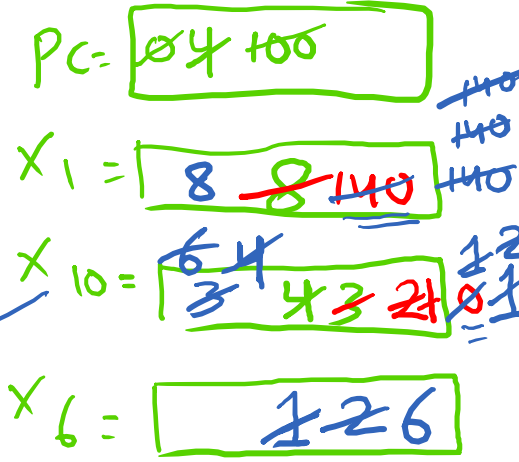
```

```

PC main:
0 addi x10, x0, 4
4 jal x1, fact
8 addi x30, x10, 0

```

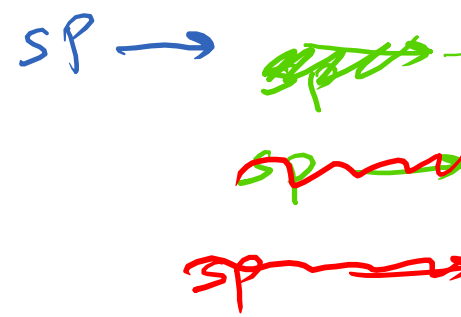
136	100
132	140
116	144
112	8
108	
104	



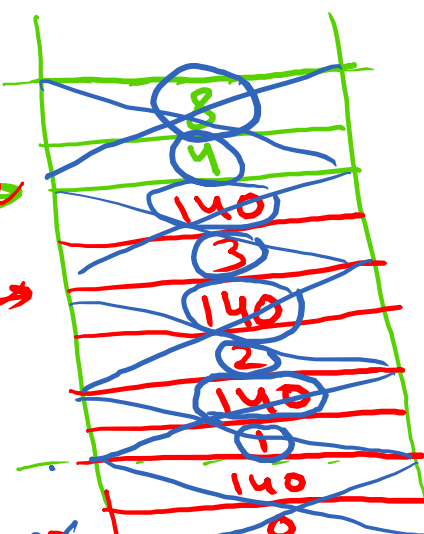
X5 = 32

① x0 = PC + 4 * X⁻¹

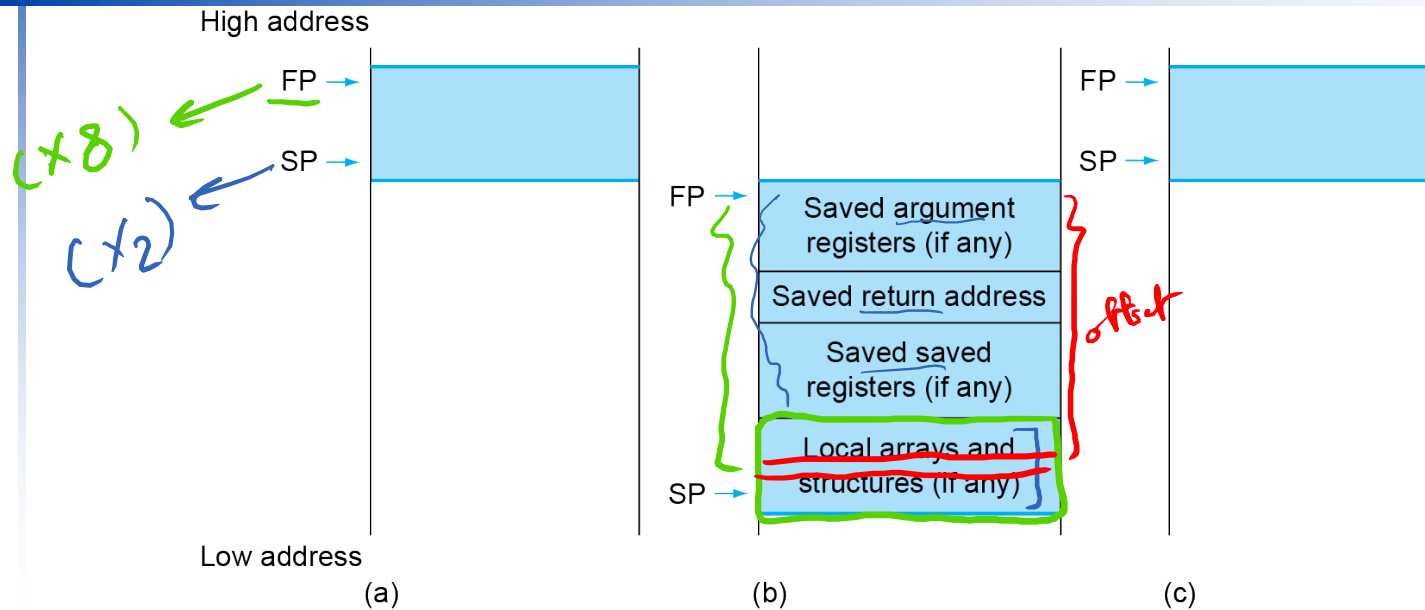
② PC = 0 + (X₌)



4 * 3 * 2 * 1 = 24



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

① Text: program code *machine language*

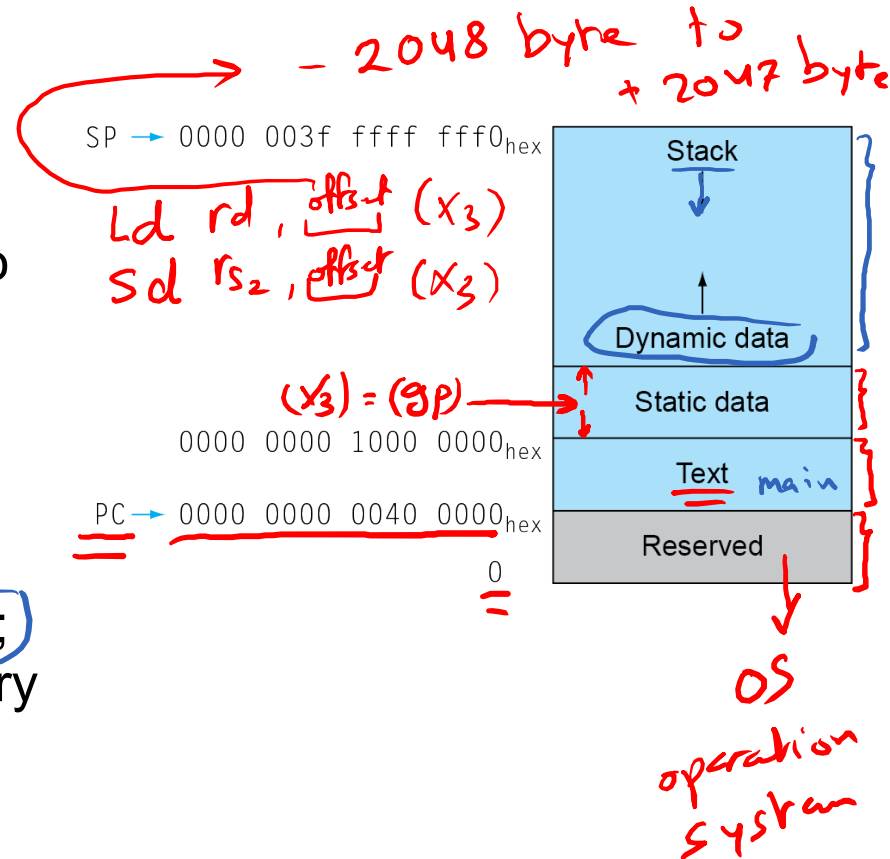
② Static data: global variables

- e.g., static variables in C, constant arrays and strings
- x3 (global pointer) initialized to address allowing \pm offsets into this segment

③ 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

④ Stack: automatic storage



Iteration Vs. Recursion

main:
 addi x10, x0, 4
 addi x11, x0, 0
 jal x1, Sum

- 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;
}
    
```

Handwritten notes: $n \leq 0$, (x_{10}) , (x_{11}) , 14 , 10 , $n \leq 0$, $n > 0$, $n = n - 1$, $n = 0$, $n = 1$, $n = 2$, $n = 3$, $n = 4$, $n = 5$, $n = 6$, $n = 7$, $n = 8$, $n = 9$, $n = 10$.

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

$x_{12} = 10$

$4 + 3 + 2 + 1 = 10$

Iteration	Recursion
<p>Sum: bge x0, x10, Exit add x11, x11, x10 addi x10, x10, -1 jal x0, Sum</p> <p>Exit: add x12, x11, x0 jalr x0, 0(x1)</p> <p><i>Handwritten notes:</i> $n \leq 0$, while (n > 0), $\rightarrow acc = acc + n$, $n = n - 1$, <u>return acc;</u></p>	<p>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</p> <p>Else: addi x12, x11, 0</p> <p>Exit: ld x1, 0(sp) addi sp, sp, 8 jalr x0, 0(x1)</p> <p><i>Handwritten notes:</i> $acc = acc + n$, $n = n - 1$, $n = 0$, $n = 1$, $n = 2$, $n = 3$, $n = 4$, $n = 5$, $n = 6$, $n = 7$, $n = 8$, $n = 9$, $n = 10$, nested</p>

tracking



```
long long int fact( long long int n)
```

```
{ if (n > 0)
```

```
    return n * fact(n-1)
```

```
else
```

```
    return 1;
```

```
}
```

$n \times (n-1) \times (n-2) \times \dots \times 1$

```
long long int ans = 1;
```

```
while ( n > 0 )
```

```
{ ans = ans * n;
```

```
  n = n - 1;
```

```
}
```

```
return ans;
```

RISC-V Register Conventions

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

7-bit / character

- ASCII: American Standard Code for Information Interchange
- 95 graphic, 33 control
- Size of (1000000000) in ASCII = 10 char * 8 bits = 80 bits = 10 bytes
- Size of (1000000000) in Binary = 32 bits

1 000 000 000

int → size of int = 4 bytes

- 2³¹ - to - (2³¹ - 1)

- Latin-1: 256 characters

- ASCII, +96 more graphic characters

8-bit / character

■ Unicode: 32-bit character set

2³² = 4 billion character

- Used in Java, C++ wide characters, ...
- Most of the world's alphabets, plus symbols
- UTF-8, UTF-16: variable-length encodings

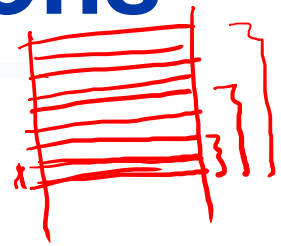
UTF-32

1000 000 000

Data size = 10 char x 32-bit = 320 bit = 80 byte



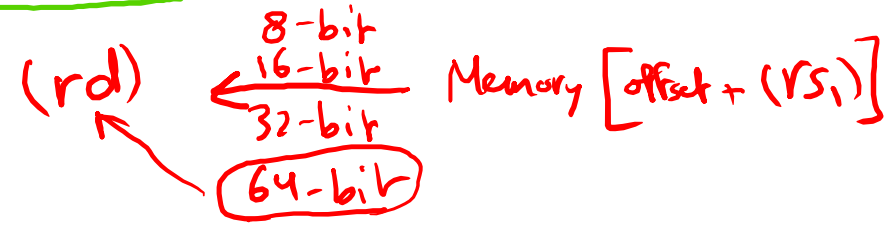
Byte/Halfword/Word Operations



RISC-V byte/halfword/word load/store

- Load byte/halfword/word: Sign extend to 64 bits in rd

ASCII UTF-16 UTF-32
load byte ← `lb rd, offset(rs1)`
load half word ← `lh rd, offset(rs1)`
load word ← `lw rd, offset(rs1)`
(rd, offset(rs1))

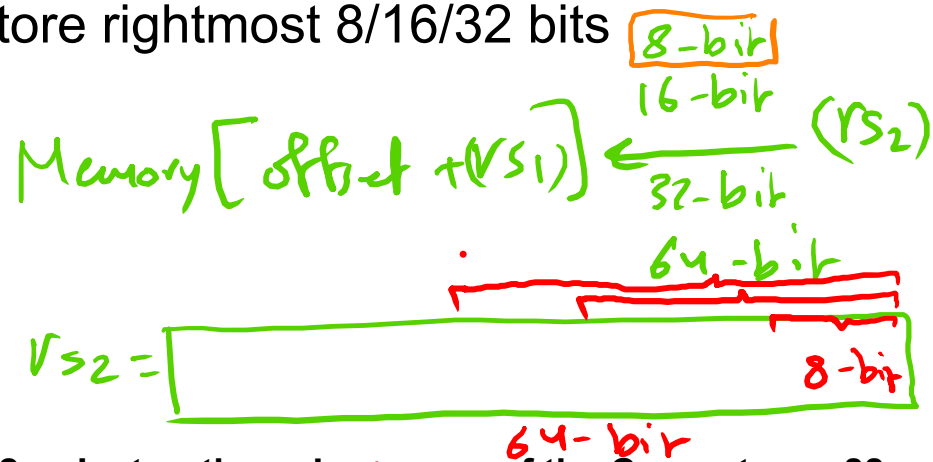


- Load byte/halfword/word unsigned: Zero extend to 64 bits in rd

unsigned
`lbu rd, offset(rs1)`
`lhu rd, offset(rs1)`
`lwu rd, offset(rs1)`

- Store byte/halfword/word: Store rightmost 8/16/32 bits

store byte ← `sb rs2, offset(rs1)`
store half word ← `sh rs2, offset(rs1)`
store word ← `sw rs2, offset(rs1)`
`sd rs2, offset(rs1)`



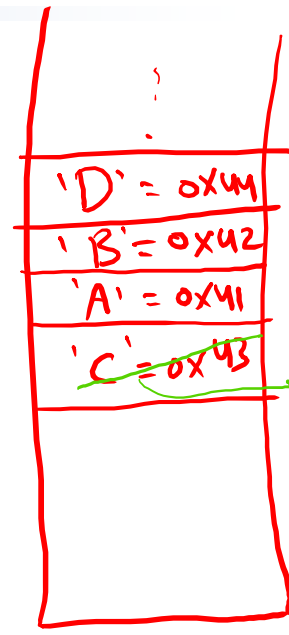
lb x5, 7(x0)

memory = 7 + (x0) = 7
address

x5 = 0x ~~0000 0000 0000 0043~~
FFFF FFFF FFFF FF86

1000

10
9
8
7



0x86

lh x6, 7(x0)

x6 = 0x ~~0000 0000 0000 4143~~
86

0100

lw x7, 7(x0)

x7 = 0x ~~0000 0000 4442 4143~~
86

'C' = (0100 0011)

Memory

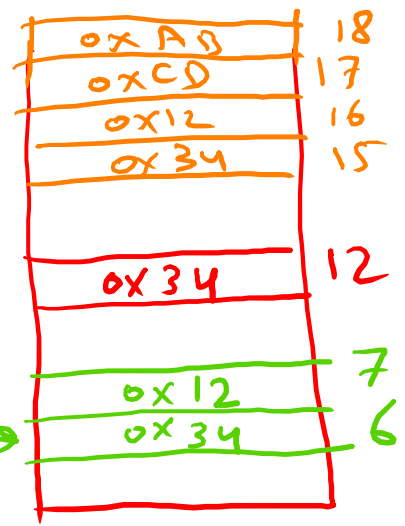
32-bit

$x_5 =$ 0x EF09 5678 ABCD 1234

$sb\ x_5, 12(x_0)$

$sh\ x_5, 6(x_0)$

$sw\ x_5, 15(x_0)$



Memory

String Copy Example

ASCII = 0x00

'0' = null character

- C code: → array of characters

- Null-terminated string

$x[i] = y[i]$

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

leaf procedure

doesn't equal

Load store

size_t = unsigned int

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

■ i is in x19

x19 is a saved registers

String Copy Example

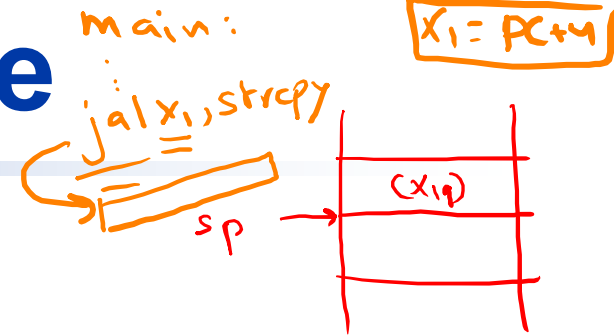
RISC-V code:

strcpy:

```

    addi sp, sp, -8           // adjust stack for 1 doubleword
    sd   x19, 0(sp)         // push x19
    add  x19, x0, x0        // 0 PC=0 addi x19, x0, 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
  
```

ASCII



unsigned

$y[i]$

memory address = offset + (rs1)

offset = $i * \text{data size in bytes}$

offset = i

memory address = $i + (rs1)$

memory address = offset + base address
 $i + (x_{10}) \leftarrow x[i]$

8-bit (x_6)

`lui X5, 0xF832A` !!!



store the constant `0xF832A471` in `X5`

`lui X5, 0xF832A`
`addi X5, X5, 0x471`



32-bit Constants (2)

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

20-bit 12-bit

$$0xFFFF\ FFFF\ FFFF = (-1)_{10}$$

$$0x003D0 + 1 = 0x003D1$$

```
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
---------------------	---------------------	--------------------------	----------------

$x_{19} \left[\begin{array}{|l} 0x \\ \hline 0000\ 0000\ 003D\ 0000 \\ \hline \end{array} \right] \leftarrow \begin{array}{l} \text{lui } x_{19}, 0x003D0 \\ \text{addi } x_{19}, x_{19}, 0x800 \end{array}$

$0xFFFF\ FFFF\ FFFF\ 800 + 1 = 0x0000\ 0000\ 003CF\ 800$

$0x0000\ 0000\ 003D0 + 800 = 0x0000\ 0000\ 003D0\ 800$

$(16)_{10} = (10)_{16}$
 $D10$
 1



x_{20} the constant $0x97AB2C34$

$0x97AB2 + 1 = 0x97AB3$ 32-bit constant

lui $x_{20}, 0x97AB3$
addi $x_{20}, x_{20}, 0xC34$

$x_{20} = 0xFFFFFFFF97AB3000 +$
 $(-1)_{10} \leftarrow 0xFFFFFFFFFFFFFFFFC34$

$x_{20} = 0xFFFFFFFF97AB2C34$

Branch Addressing (backward) Label:

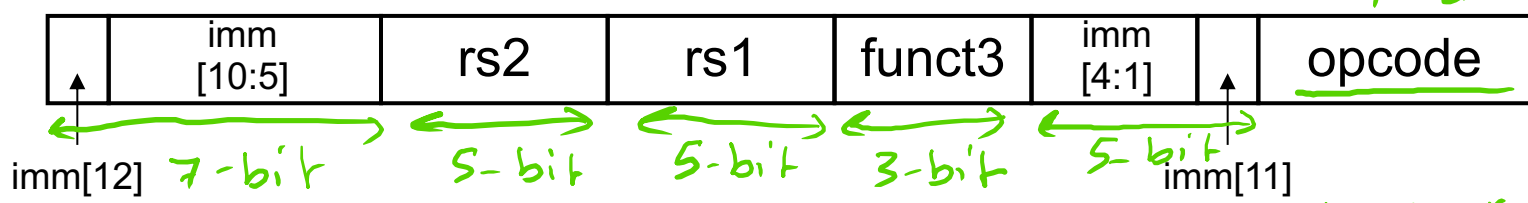
- Branch instructions specify
 - Opcode, two registers, target address
- Most branch targets are near branch
 - ① Forward or ② backward
- SB format:

bne
bne
bll
bll
bge
bge
bge
bge

rs₁, rs₂, Label

12-bit

(Forward) Label:



PC-relative addressing

of the branch (memory address of the branch)

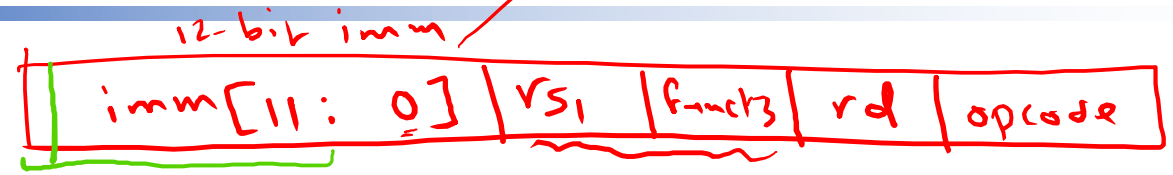
Target address = PC + immediate^{12-bit} × 2

- The immediate represents the offset in halfwords

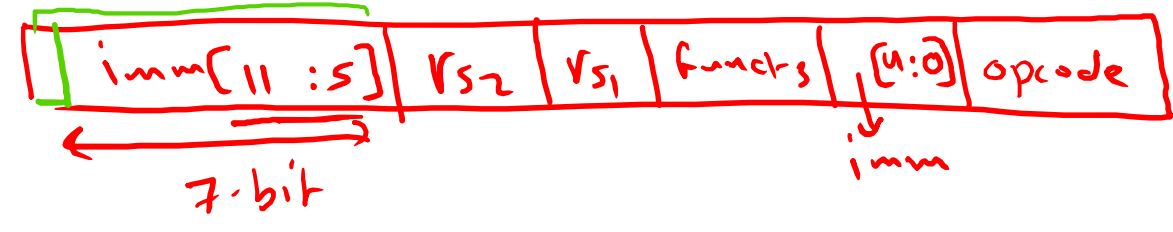
memory
→ address of the target instruction

I-format

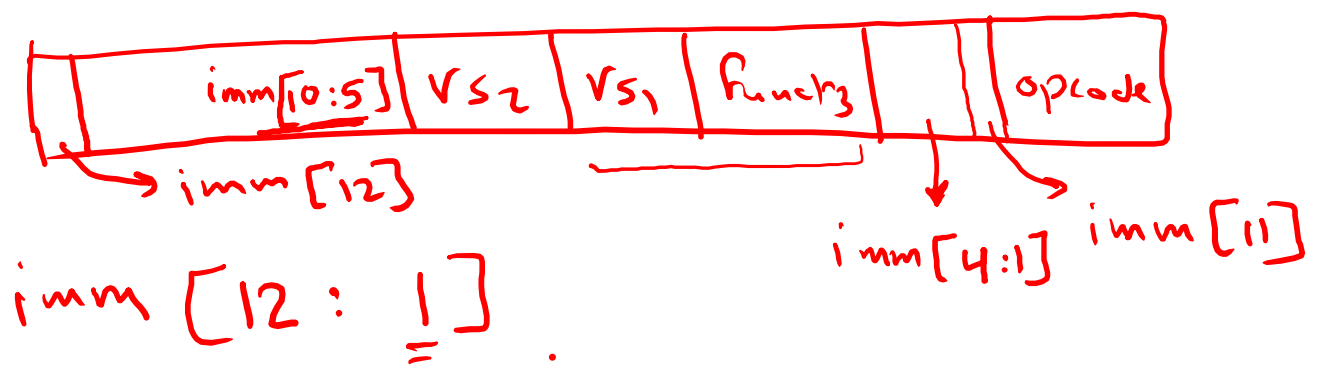
I-format =



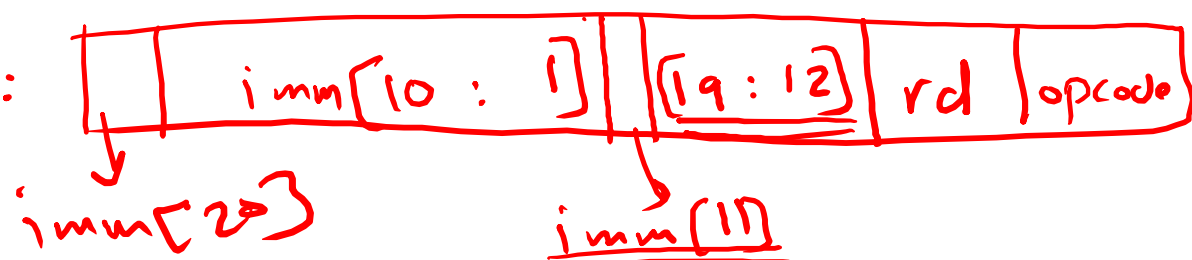
S-format =



SB-format =



UI-format =



40 beq x0, x0, Exit
 44 ----- ✓
 48 ----- ✓
 52 ----- ✓
 56 ----- ✓
 60 Exit: ----- ✓

Target address = PC of branch + imm × 2
 60 = 40 + imm × 2
imm = 10

40 beq x0, x0, (10)₁₀
 44 ----- ✓
 48 ----- ✓
 52 ----- ✓
 56 ----- ✓
 60 ----- ✓

$(0000\ 0000\ 1010)_2$
 $\rightarrow 0x00A$

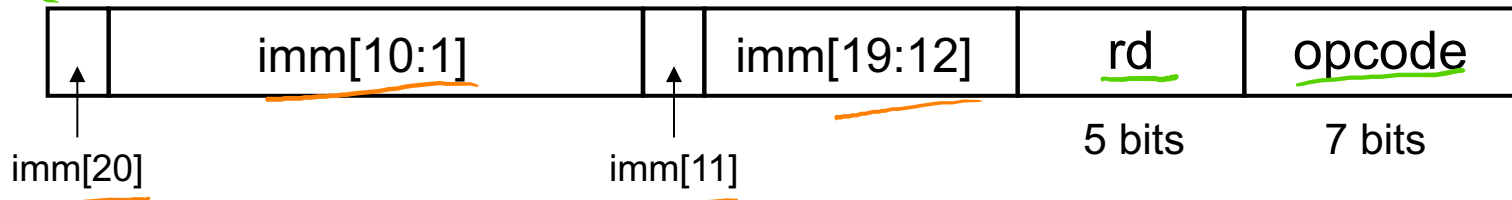
Target address = 40 + 10 × 2
 = 60

half words

Unconditional Jump-and-Link Addressing

jal rd, label
20-bit

- 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

of jal

imm[20:1]

40 jal x0, Exit

44 -----

48 -----

52 -----

56 -----

60 -----

64 Exit: -----

$$64 = 40 + \text{Exit} * 2$$

$$\text{Exit} = \frac{24}{2} = (12)_{10}$$

40 jal x0, (12)₁₀

44 ----- ✓

48 ----- ✓

52 ----- ✓

56 ----- ✓

60 ----- ✓

64 ----- ✓

$(0000\ 0000\ 0000\ 0000\ 1100)_2$
 $0x0000C$

$$\text{Target} : 40 + 12 * 2 = 64$$

branch $rs_1, rs_2, \text{immediate}$ } $\xrightarrow{12\text{-bit}}$ 52-bit

jal $rd, \text{immediate}$ } $\xrightarrow{20\text{-bit}}$ 44-bit

Target address = $\underbrace{PC}_{64\text{-bit}} + \text{immediate} \times 2$

= $PC + (\text{sign-extension immediate}) \times 2$
 (64-bit)

more accurate equation



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

2^{20} bytes

- An alternative is to specify a register that would always be added to the branch/jal offset:

64-bit \Rightarrow 2^{64} bytes

- Allows the program to be as large as 2^{64} bytes
- PC contains the address of the current instruction
- PC is the ideal choice

16-bit

- 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 ^{mega}

+ 1K words
+ 256K words
- 20-bit \Rightarrow

-2^{19} to $+(2^{19}-1)$ half words | 12-bit \Rightarrow -2^{11} to $+(2^{11}-1)$ half words
 -2048 to 2047 //

+ 1024 K

Branch Offset in Machine Language

$(-10)_{10} = (1011\ 1111\ 0110)_2$ $(6)_{10} = (0000\ 0000\ 0110)_2$
 (Note: The binary for -10 is shown with a leading 0, and the binary for 6 is shown with a leading 0. The bit positions 12, 11, and 10 are indicated.)

- Loop code from earlier example

- Assume Loop at location 80000

$80000 = 80020 + \text{loop} * 2$

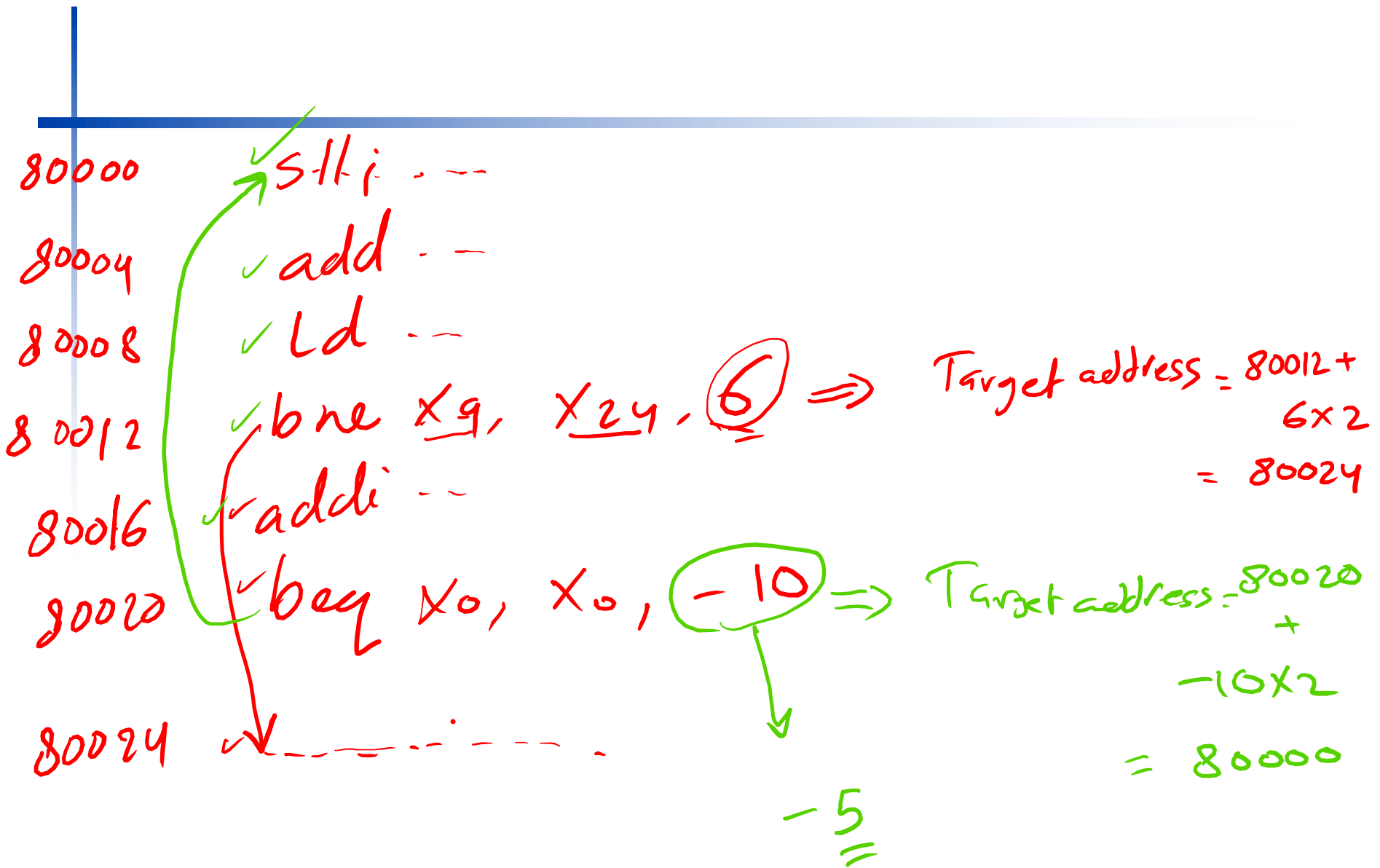
Loop = -10

Address	Instruction	rs	func3	rd/imm	opcode
80000	slli x10, x22, 3	0000000	00011	10110	001010011
80004	add x10, x10, x25	0000000	11001	01010	0110011
80008	ld x9, 0(x10)	0000000	00000	01010	011000011
80012	bne x9, x24, Exit (Forward)	0000000	11000	01001	01100110011
80016	addi x22, x22, 1	0000000	00001	10110	001010011
80020	beq x0, x0, Loop (Backward)	1111111	00000	00000	01101110011

```

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

Target address = PC of branch + Exit * 2
 $80024 = 80012 + \text{Exit} * 2$
Exit = 6



Long Jumps and Branching Far Away

$PC = Target = sign_extension(0x650) + (X5)$

- For long jumps, e.g., to **32-bit absolute address** of a procedure
 Target address: 0x000000003F21A650

- lui: load address[31:12] to temp register
- jalr: add address[11:0] and jump to target

```

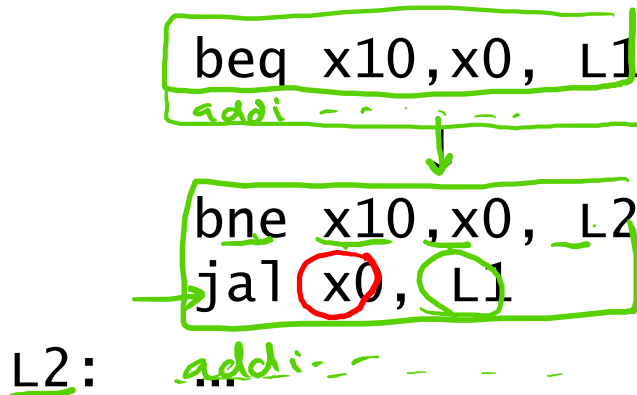
    lui x5, 0x3F21A
    jalr x0, 0x650(x5)
  
```

20-bit 12-bit

$X5 = 0x000000003F21A000$
 $0x00000000000000650 \Rightarrow PC = 0x000000003F21A650$

- If branch target is too far to encode with 12-bit offset, assembler rewrites the code

- Example



what if L1 is further than $\pm 1MB$?

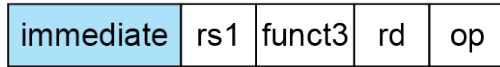
```

    bne x10, x0, L2
    jal x0, L3
    L2: ---
    L3: jal x0, L1
  
```

branch $\pm 4KB$
 jal $\pm 1MB$

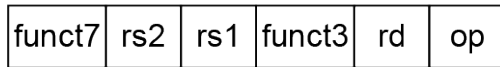
RISC-V Addressing Summary

1. Immediate addressing



*Immediate - format, S-format, SB-format
UJ-format, lui-format*

2. Register addressing

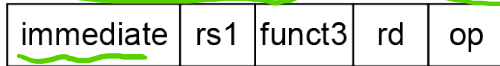


Registers *File*

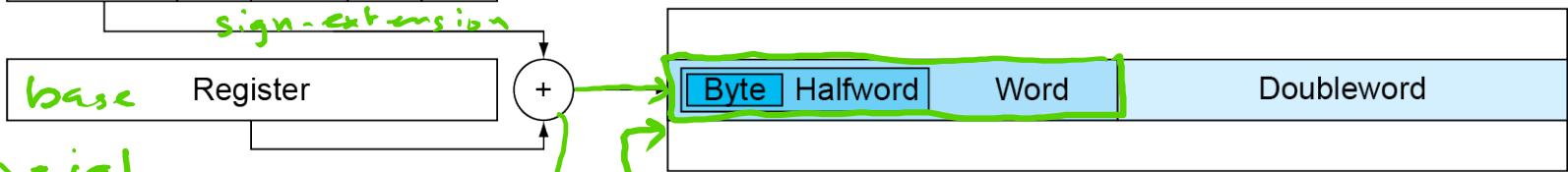


3. Base addressing

store, load, jalr

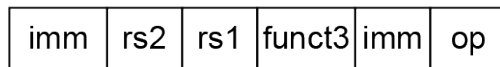


Memory

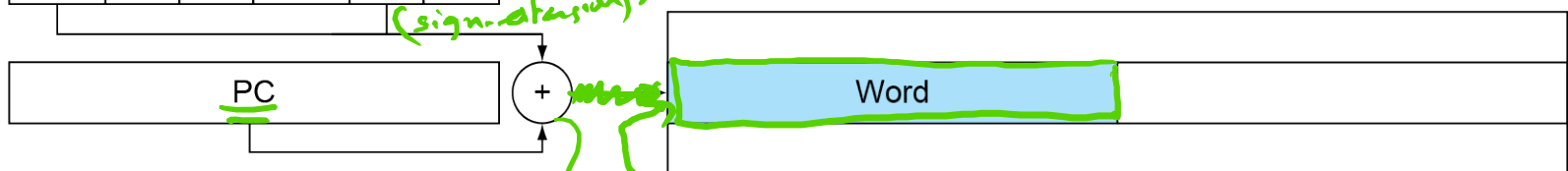


branch jal

4. PC-relative addressing



Memory



RISC-V Encoding Summary

imm[20:13]

Name (Field Size)	Field						Comments
	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

funct6 | shift amount | rs1 | funct3 | rd | opcode | slli, srli
6-bit

jal rd, imm
lui rd, imm

lui x5, 0xF53B0



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
 - ✓ ① ■ A simplified version (i.e. Single-Cycle implementation)
 - ② ■ Multi-Cycle implementation *Appendix C*
 - ✓ ③ ■ 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

- ① PC → instruction memory, fetch instruction
- ② Register numbers → register file, read registers

③ Depending on instruction class

- Use ALU to calculate

- Arithmetic result *R-format* (rs_1) operation (rs_2)
- Memory address for load/store $Memory = (rs_1) + \text{sign-ext}(\text{offset})$
- Branch comparison $(rs_1) - (rs_2) \Rightarrow \text{Zero Flag}$

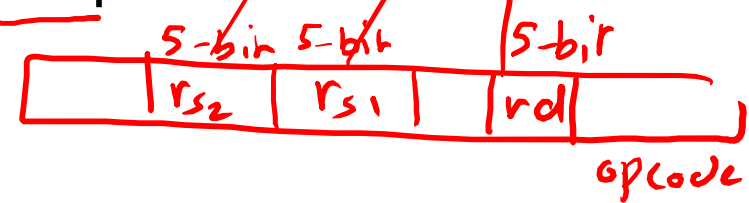
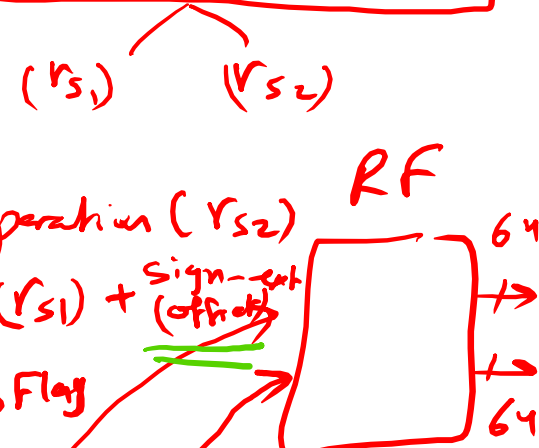
calculation
"beq"

- Access data memory for load/store

- ⑤ PC ← target address or PC + 4

⑤.1 Write Result to "rd"
⑤.2 *machine code*

code
- lui rd, imm
- jal rd, imm

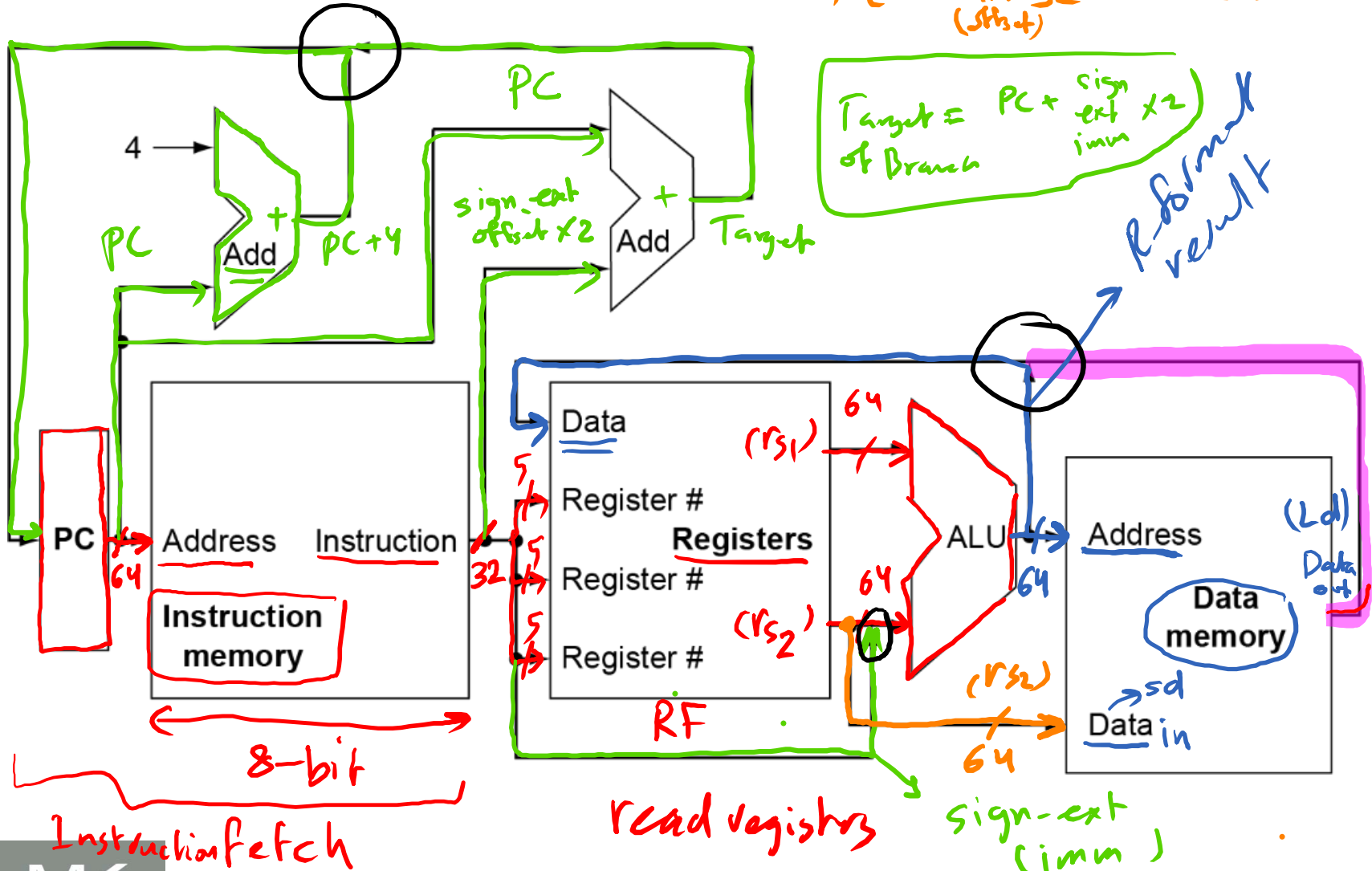


R-format I-format

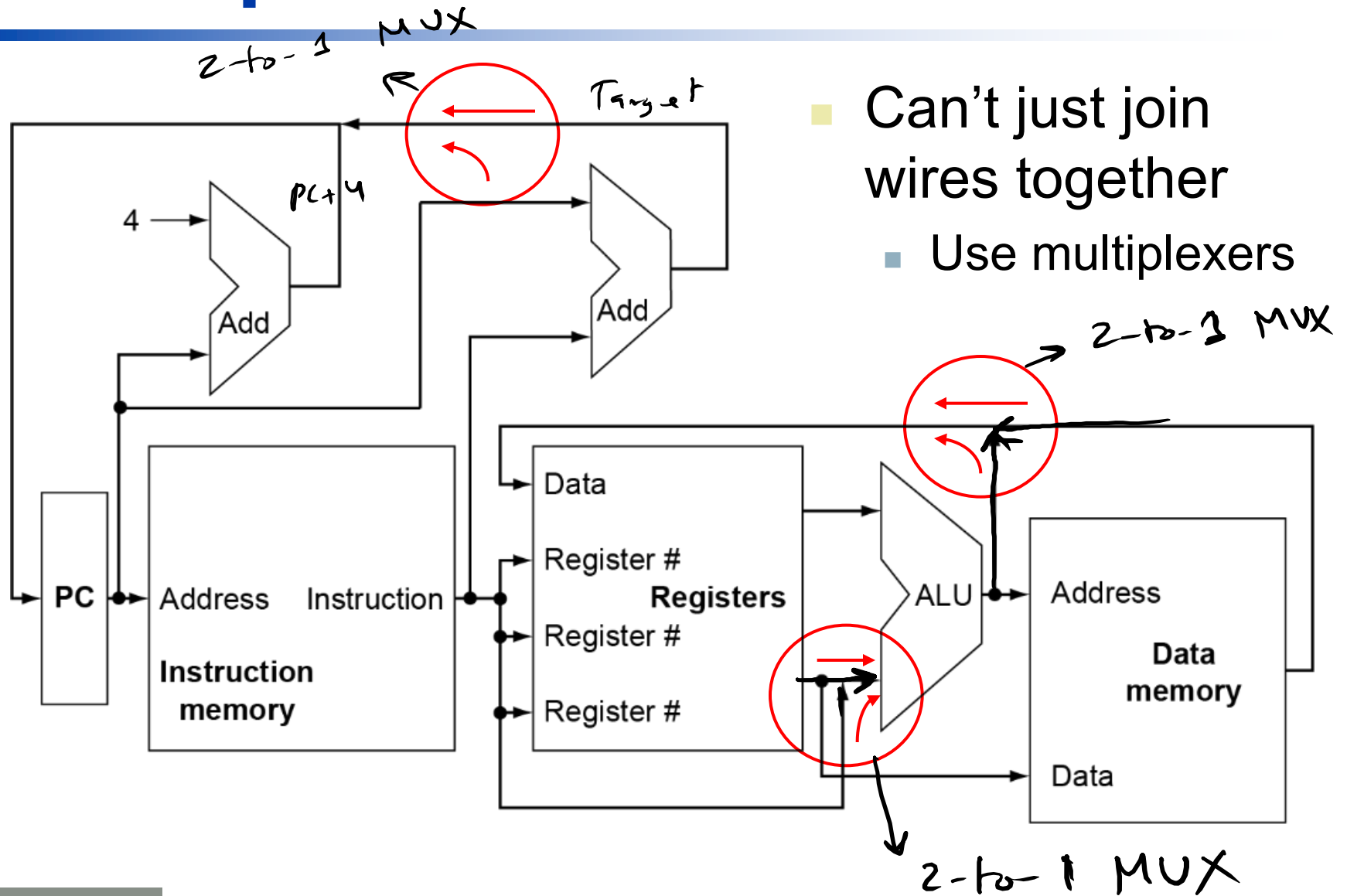
CPU Overview

$sd\ rs_2, offset(rs_1)$
 Memory $[(rs_1) + sign_ext(rs_2)]$

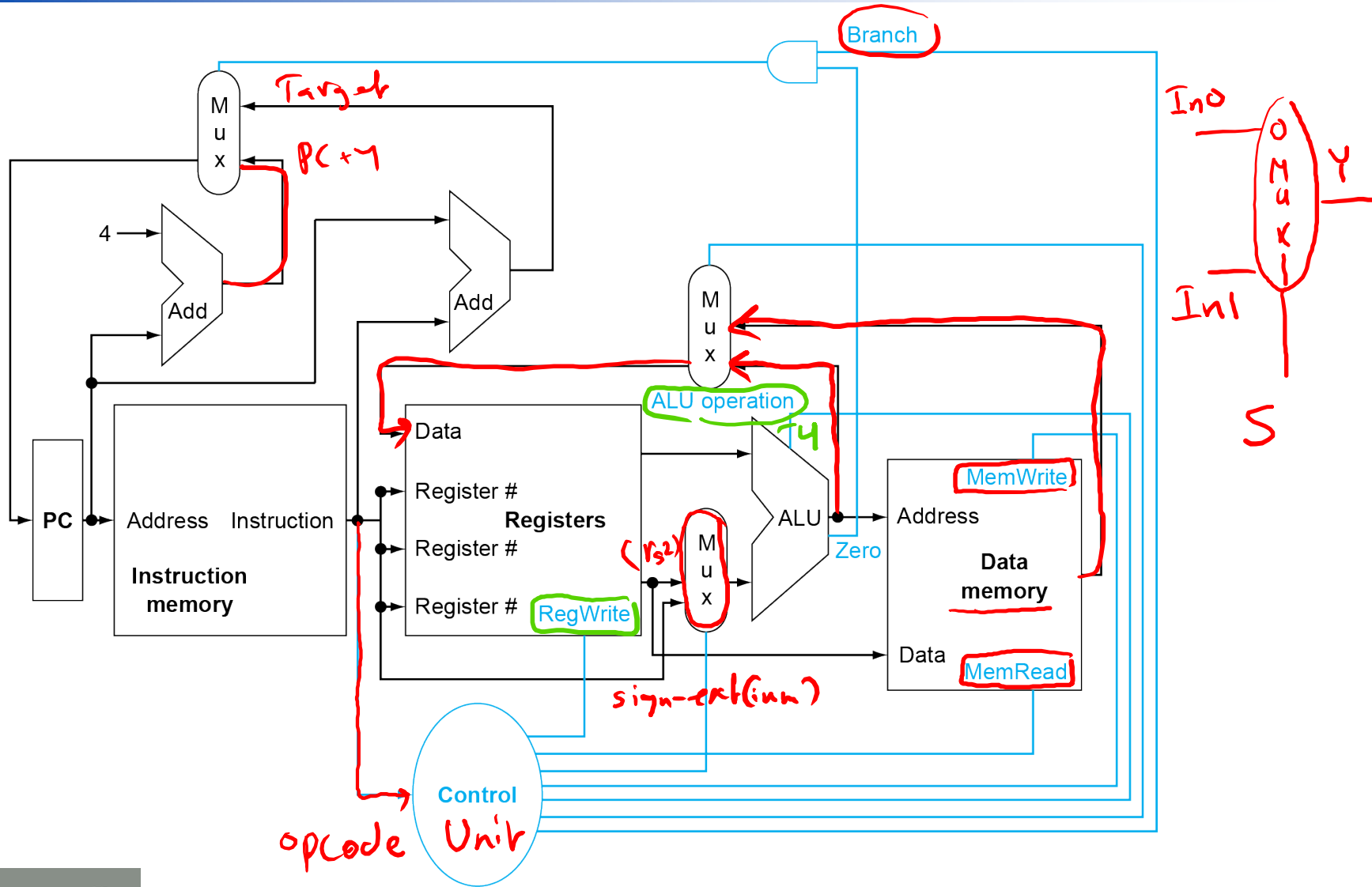
Target = $PC + sign_ext(immed)$
 of Branch
 Read back result



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

ACW
Muxes
address
D

Combinational element

64 wires bus



- Operate on data
- Output is a function of input

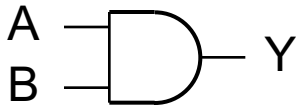
State (sequential) elements

- Store information: RF, Instructions & Data memories, PC

Combinational Elements

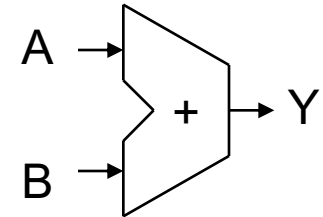
■ AND-gate

- $Y = A \& B$



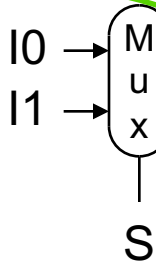
■ Adder

- $Y = A + B$



■ Multiplexer

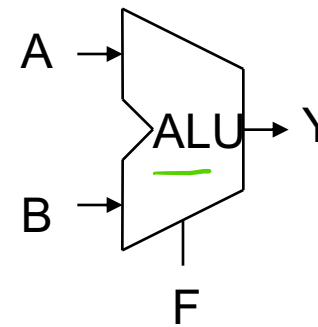
- $Y = S ? I1 : I0$



S = 1
S = 0

■ Arithmetic/Logic Unit

- $Y = F(A, B)$

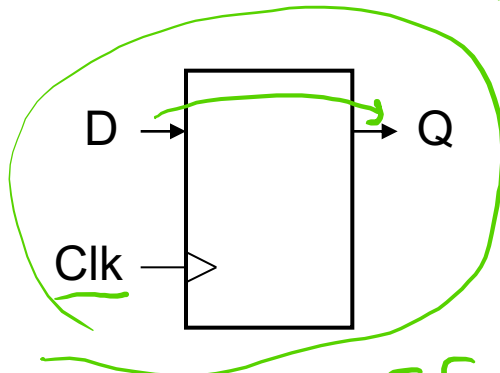


function:
and
or
add/sub
slt

Sequential Elements

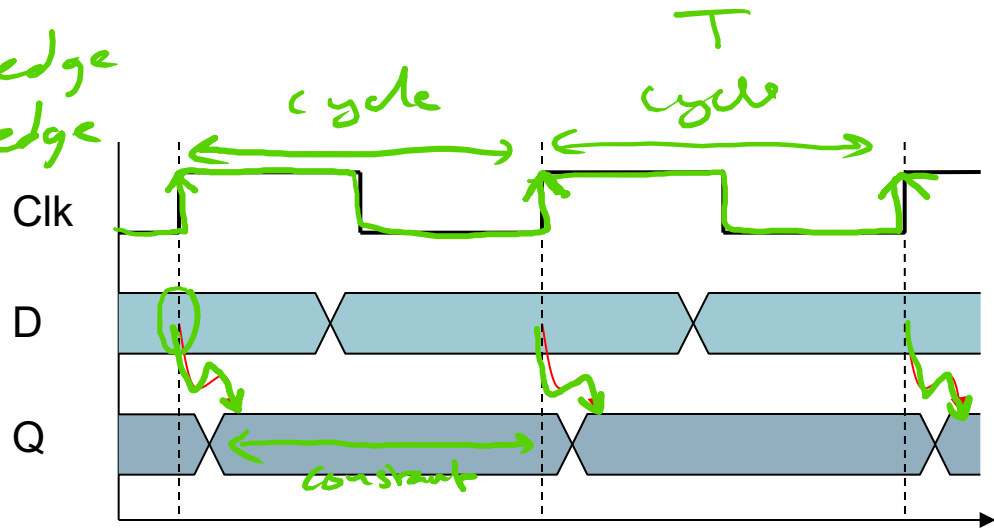
$$\text{clock Rate (F)} = \frac{1}{T}$$

- 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



rising edge
+ve edge

+ve edge triggered D-FF

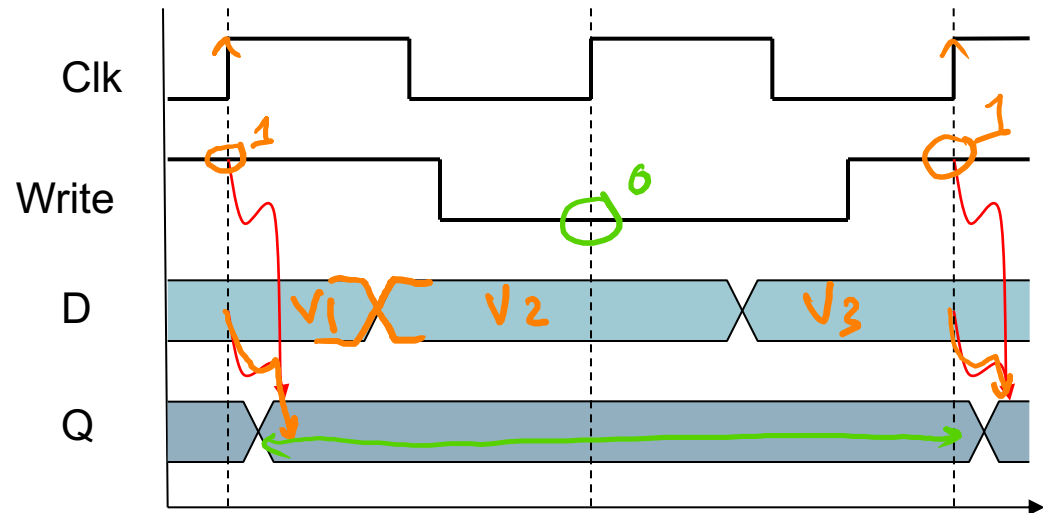
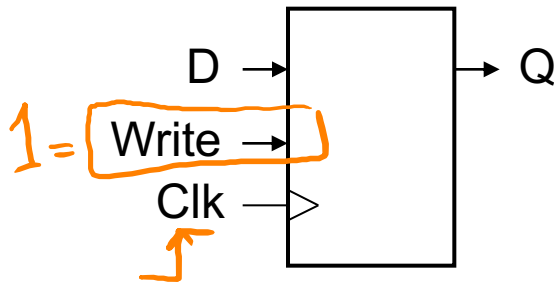


negative-edge triggered D-FF
1 → 0

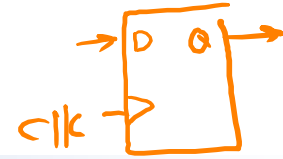
Sequential Elements

beq rs1, rs2, label
sd rs2, offset(rs1, ...)

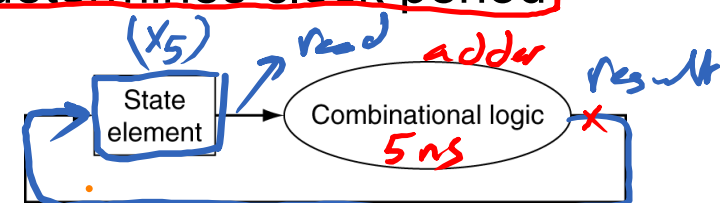
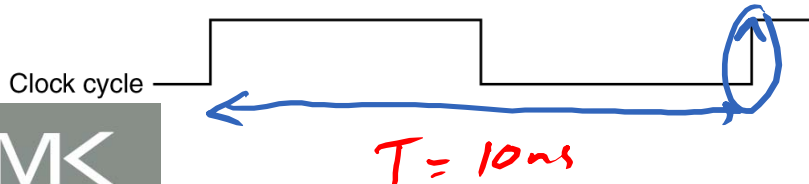
- Register with write control
 - Only updates on clock edge when write control input is 1
 - Used when stored value is required later



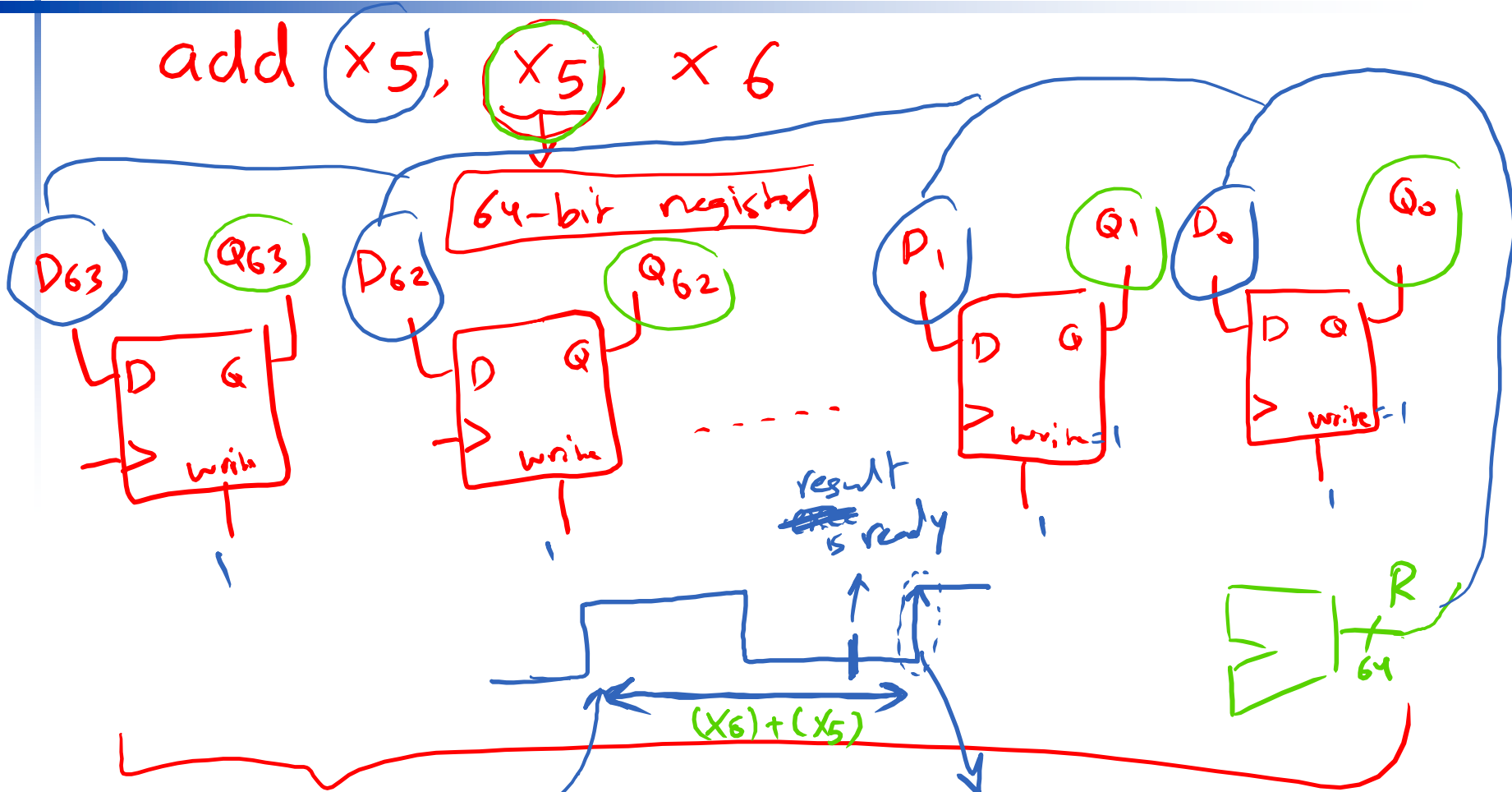
Clocking Methodology



- Clocking methodology** defines when signals can be read or written *from storage elements* - *while reading we read the value of "Q"*
 - 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 - *while writing we write the value of "D" on "Q"*
- 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 *Critical path*



add x_5 , x_5 , x_6



read registers (old value) x_5

write to register when write control = 1

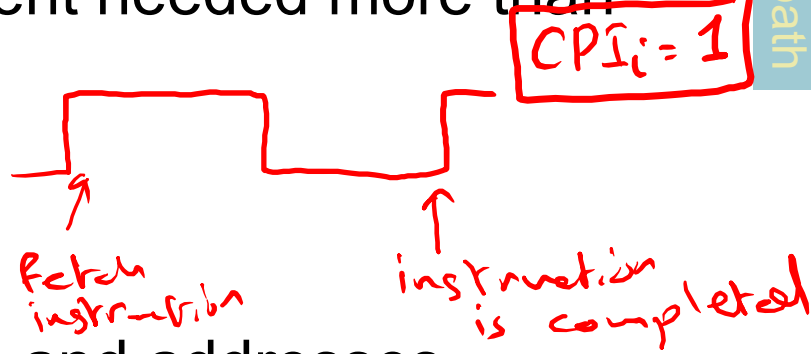
Building Datapath for Single-Cycle CPU

- Every instruction is executed in one cycle $CPI_{avg} = 1$

- No datapath resource can be used more than once per instruction \rightarrow any element needed more than once must be duplicated

$IC = 100$
Single_Cycle CPU \Rightarrow

CPU clock = 100
Cycles = $IC \times CPI_{avg}$
 $100 = 100 \times CPI_{avg}$

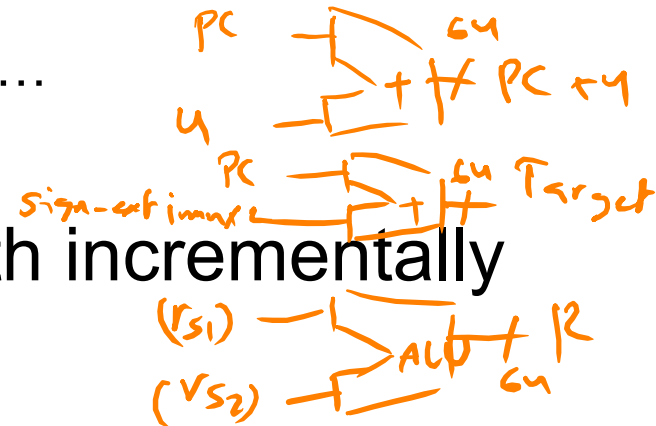


Datapath

- Elements that process data and addresses in the CPU
 - Registers, ALUs, mux's, memories, ...

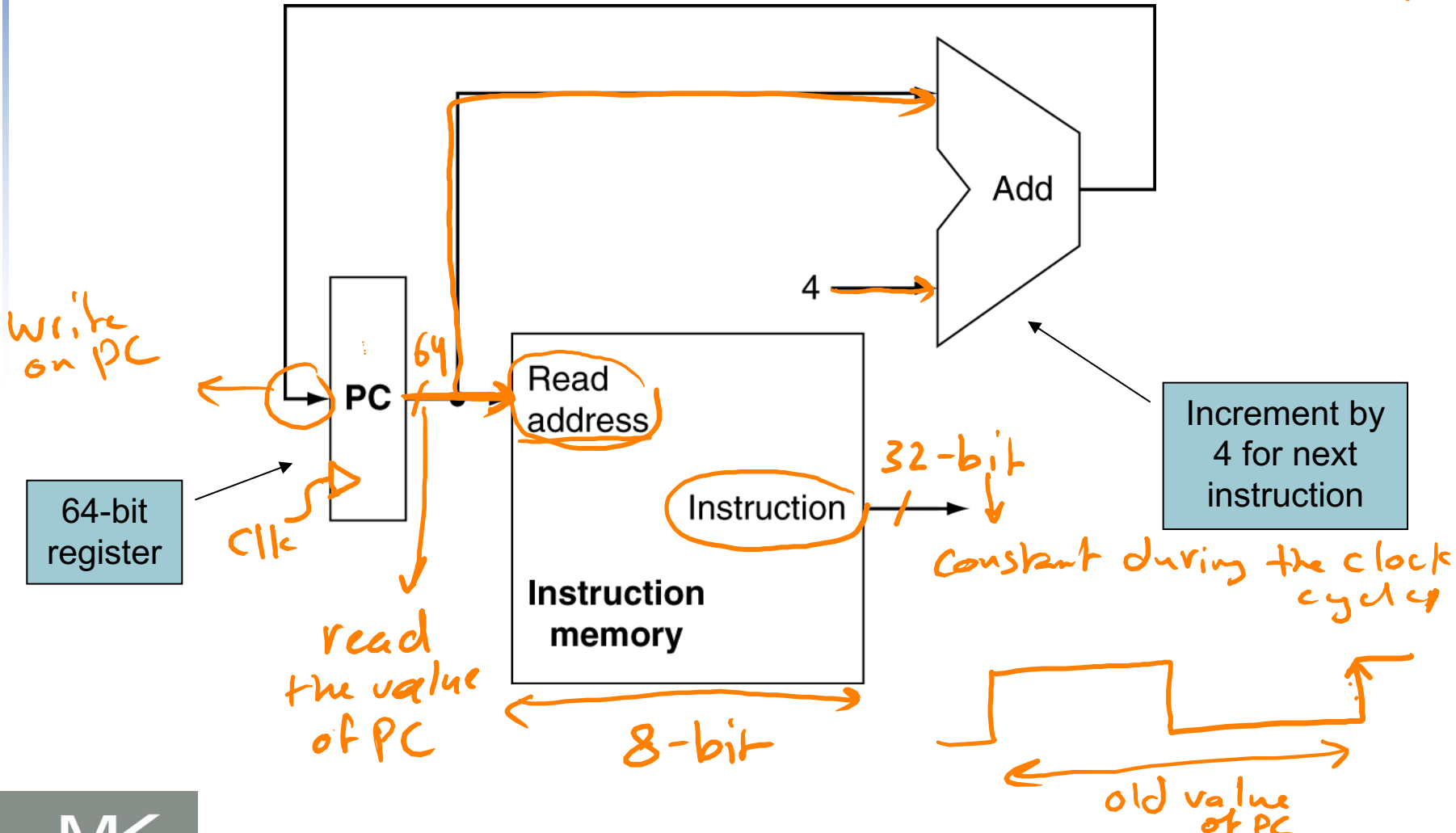
We will build a RISC-V datapath incrementally

- Refining the overview design



Instruction Fetch

read ^{the} instruction to be executed from instruction memory



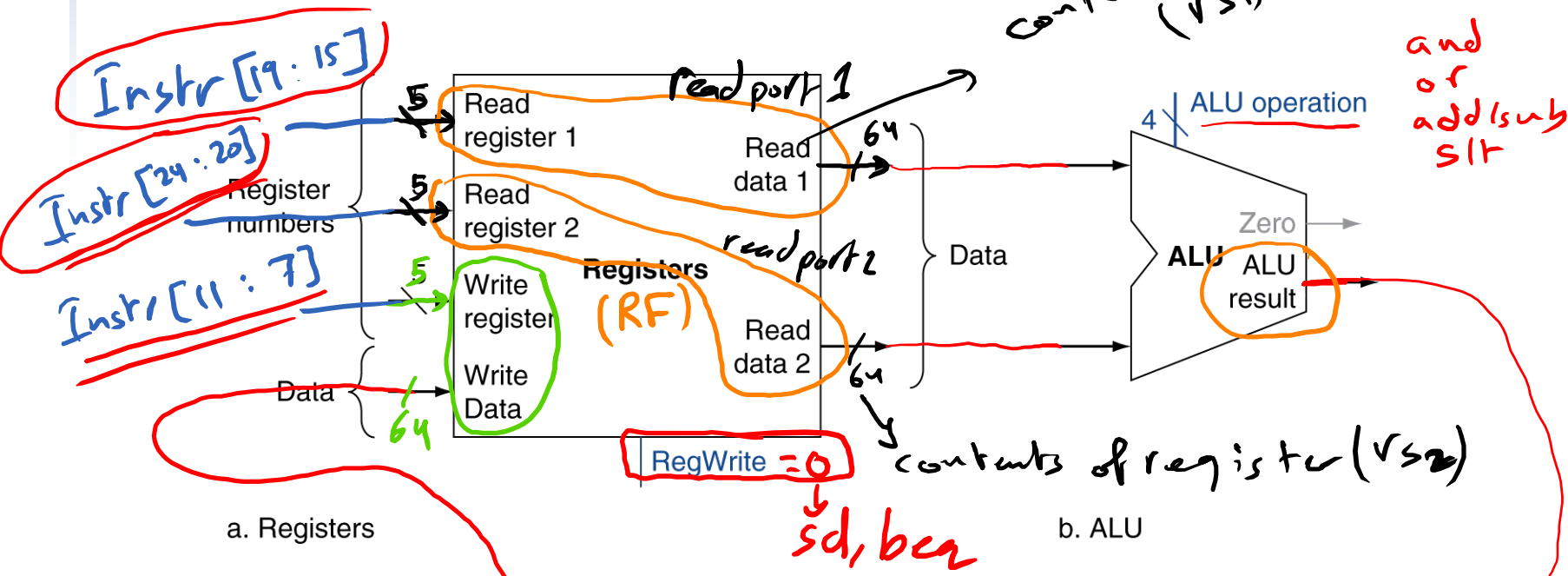
R-Format Instructions

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

RF has two read ports & one write port

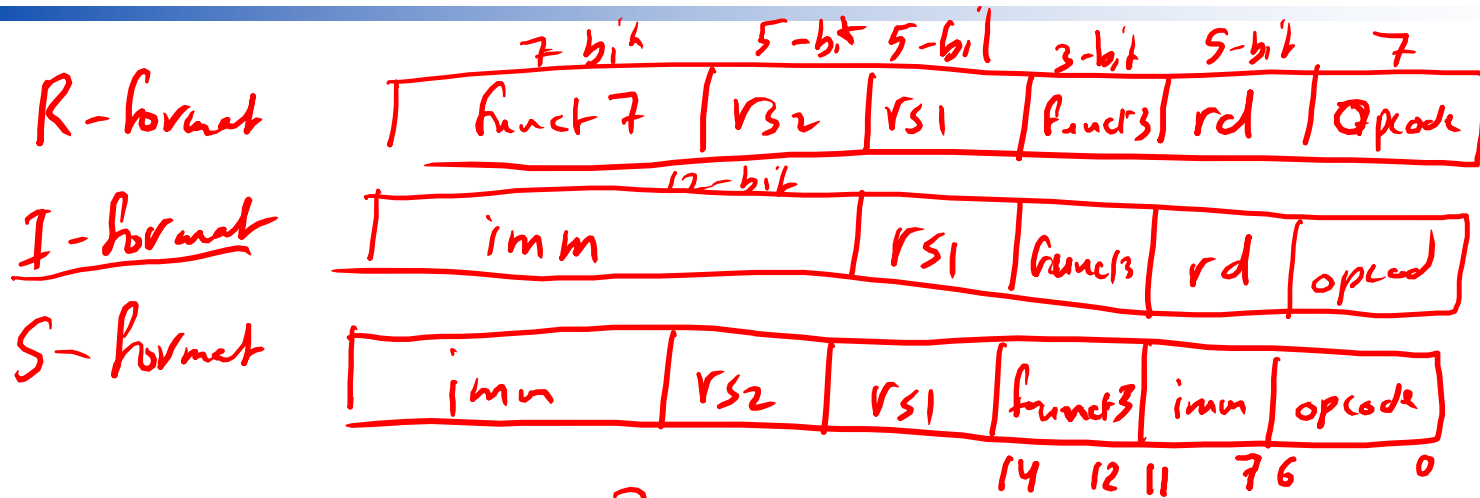
content of register 1 (rs1)

and/or address bit



a. Registers

b. ALU



Instr [31:0]

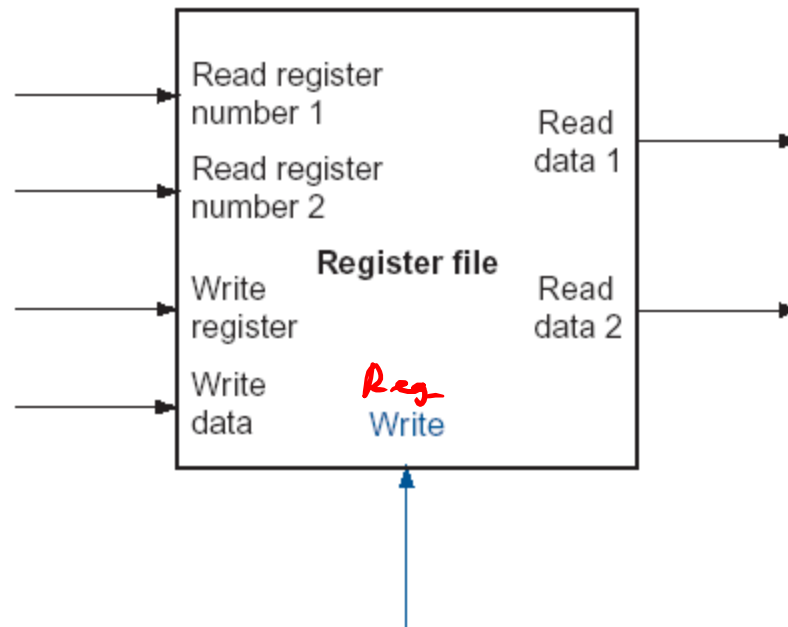
Instr [11:7] ⇒ "rd" number

Instr [19:15] ⇒ "rs1" "

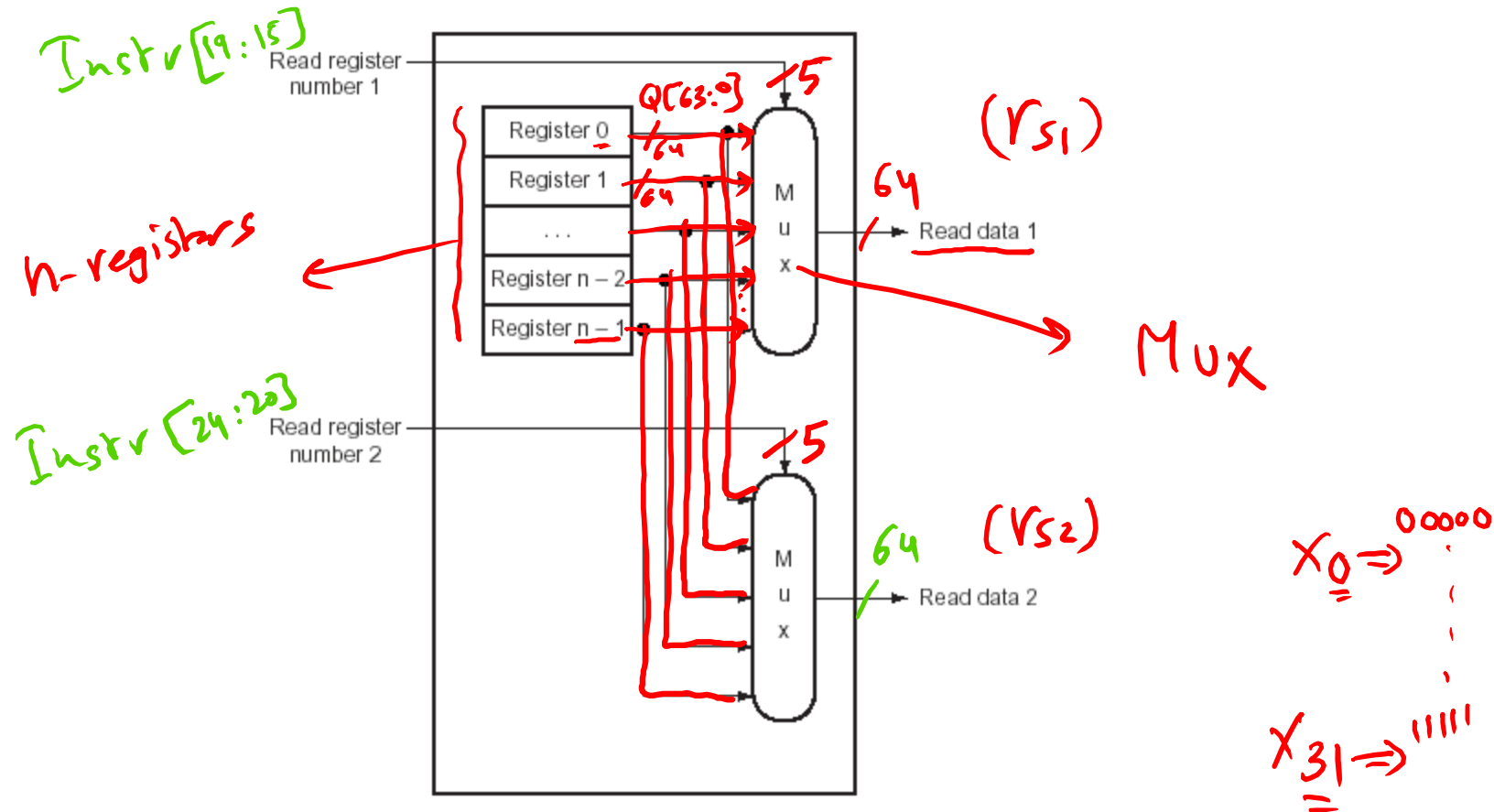
Instr [24:20] ⇒ "rs2" "

read is harmless but write is harmful

Register File

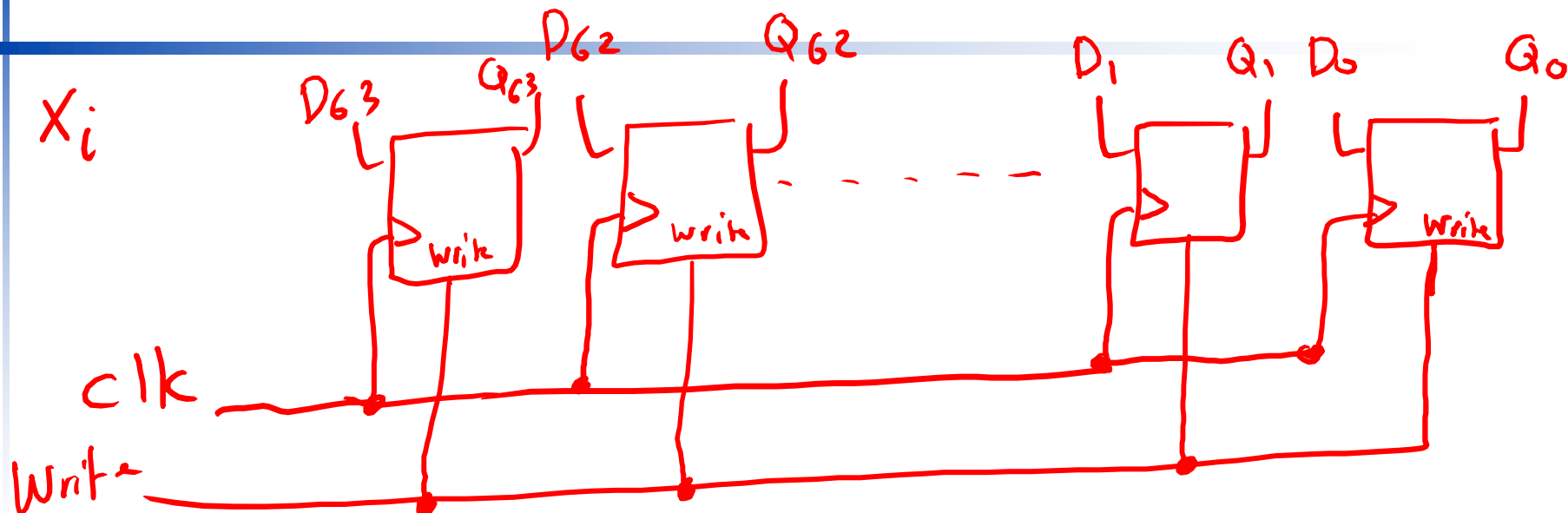


Register File – Read Ports

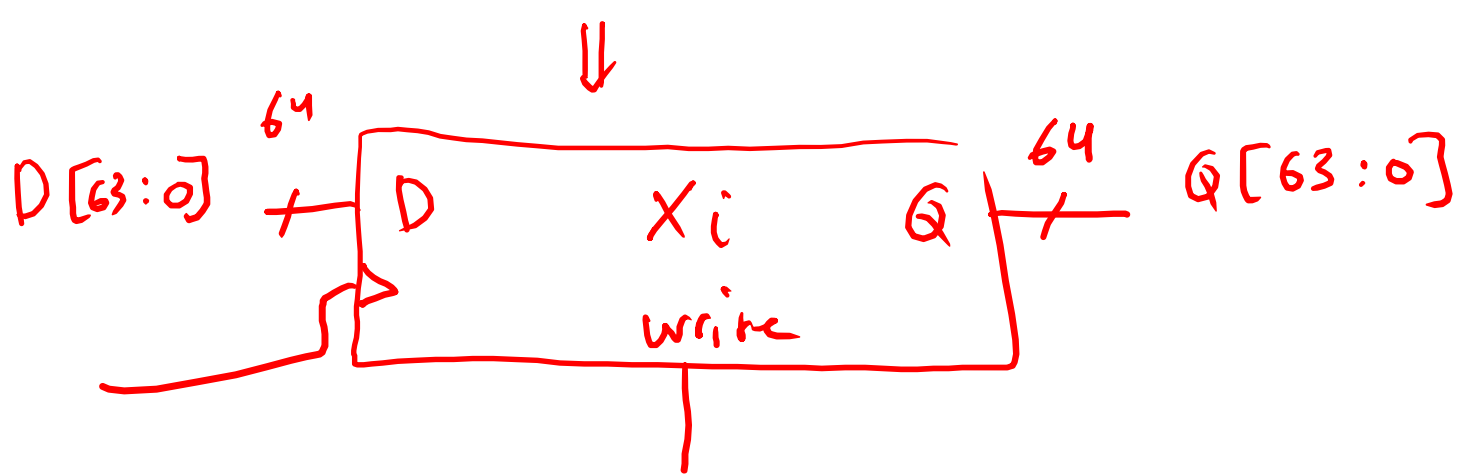


- Use two 64-bit 32-to-1 multiplexers whose control lines are the register numbers.

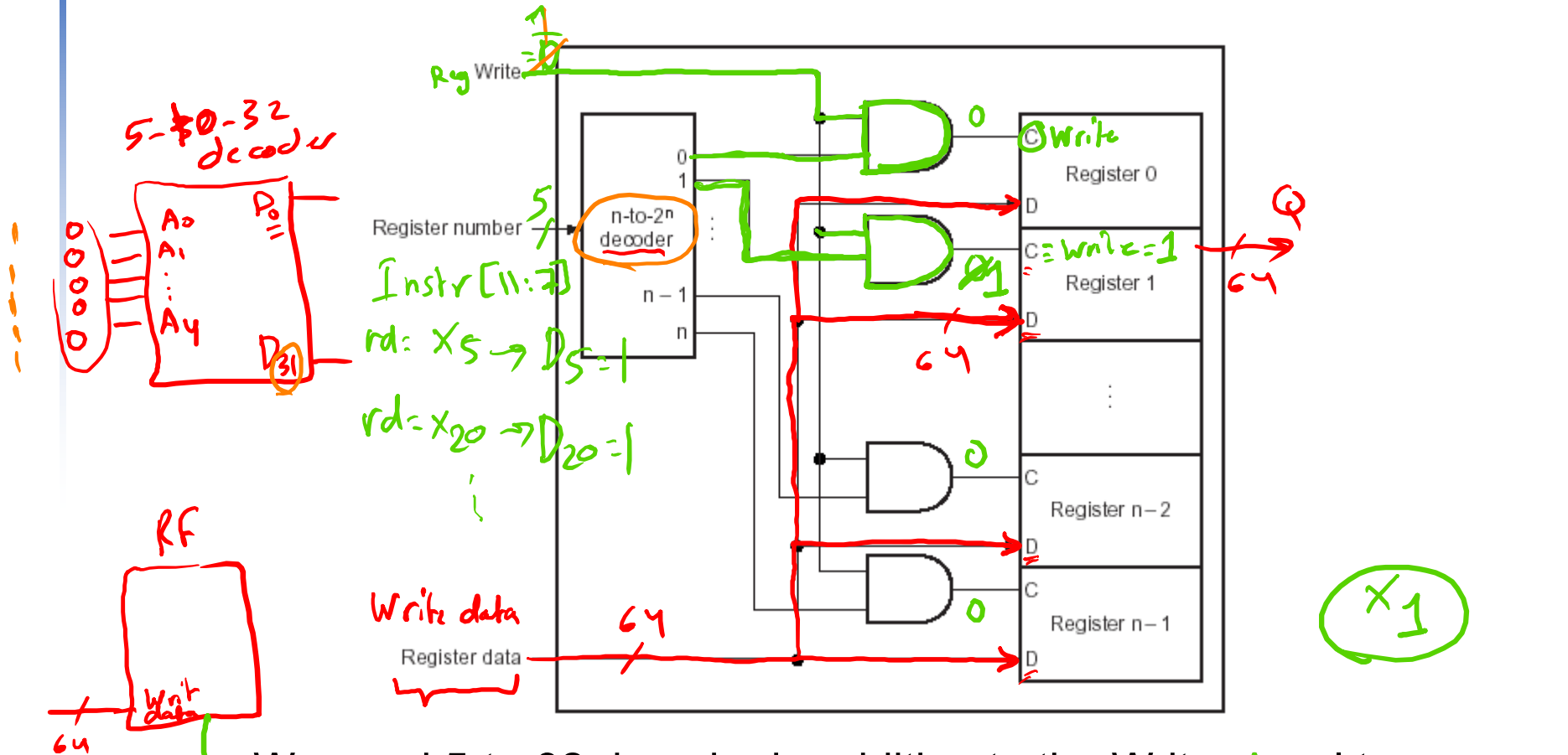
$$32 = 2^5$$



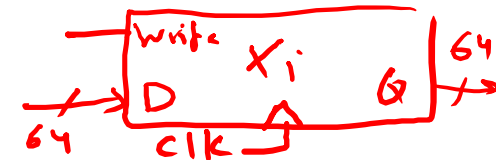
64-bit register



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



For which instructions $\text{RegWrite} = 1$?

- Load

- addi

- jal

- R-format

- jalr

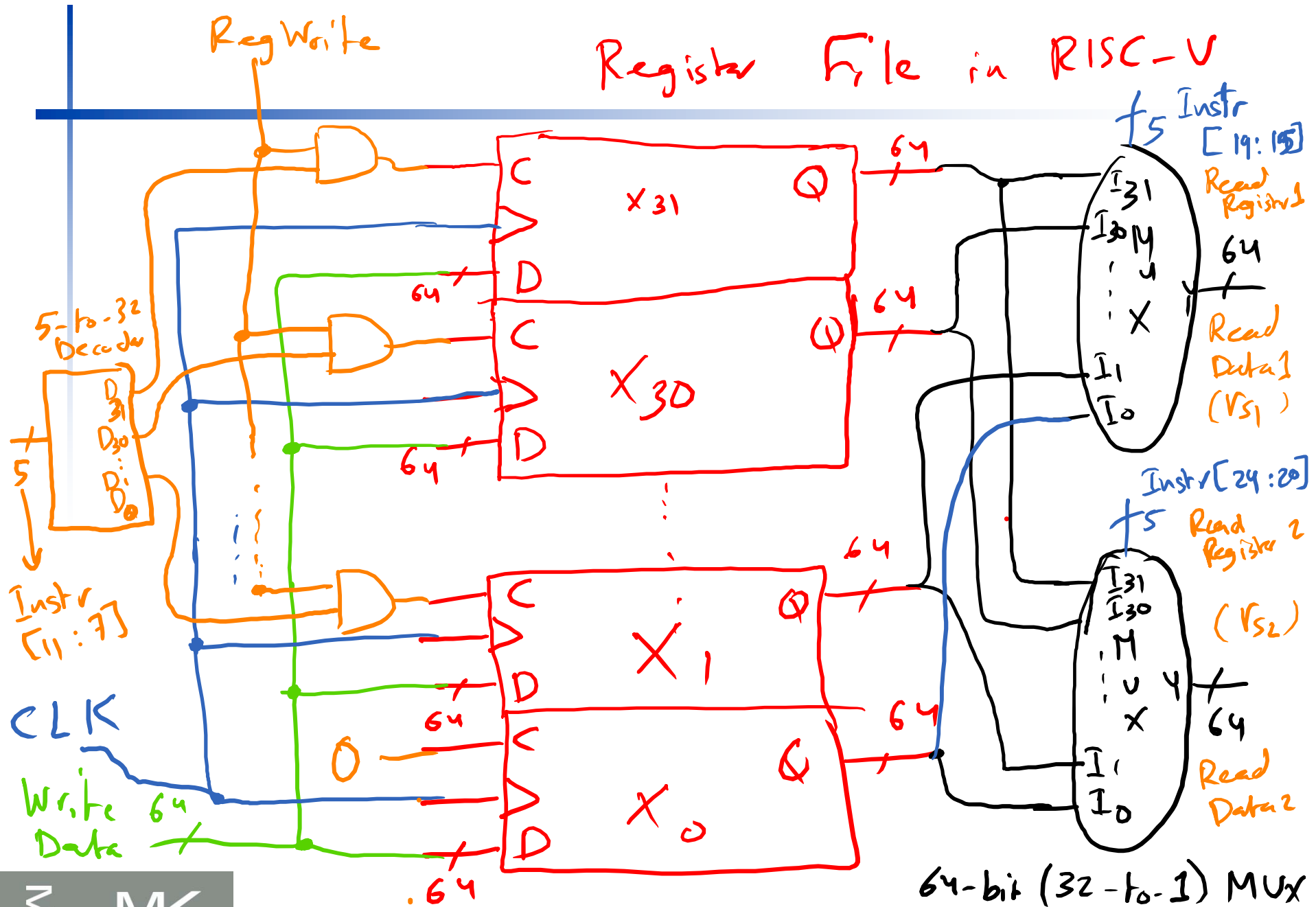
- lui

instructions $\text{RegWrite} = 0$?

- store

- branch

Register File in RISC-V

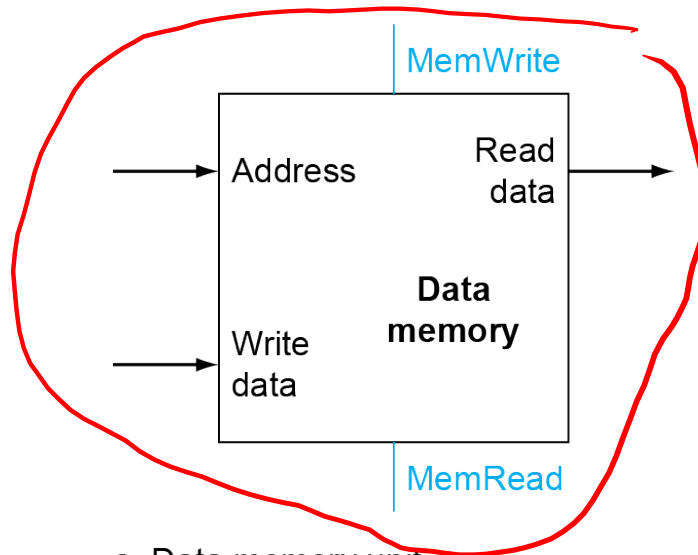


64-bit (32-to-1) MUX
Chapter 4 — The Processor — 22

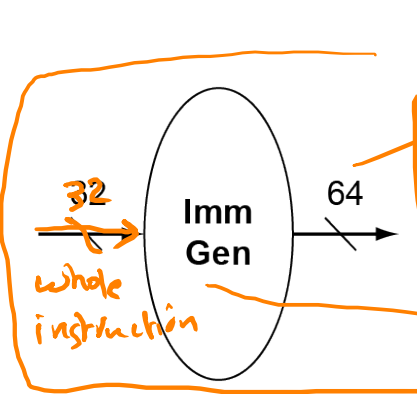
Load/Store Instructions

- 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

*memory address = (base register) + sign-ext offset
= (rs₁) + sign-ext offset*



a. Data memory unit



b. Immediate generation unit

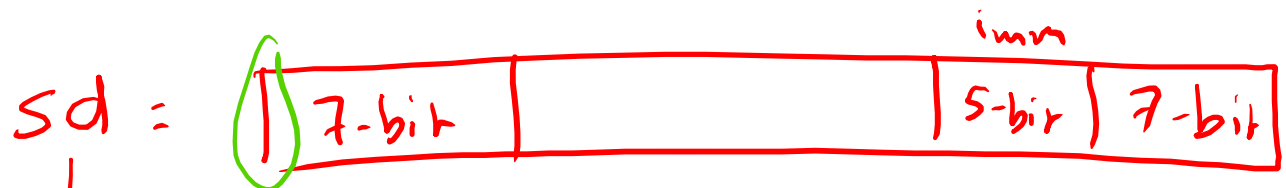
sign-extended imm
Immediate generation

Immediate Generation

① Extract immediate from instruction



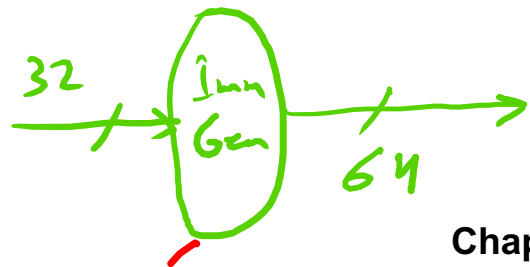
↳ instr[31:20]

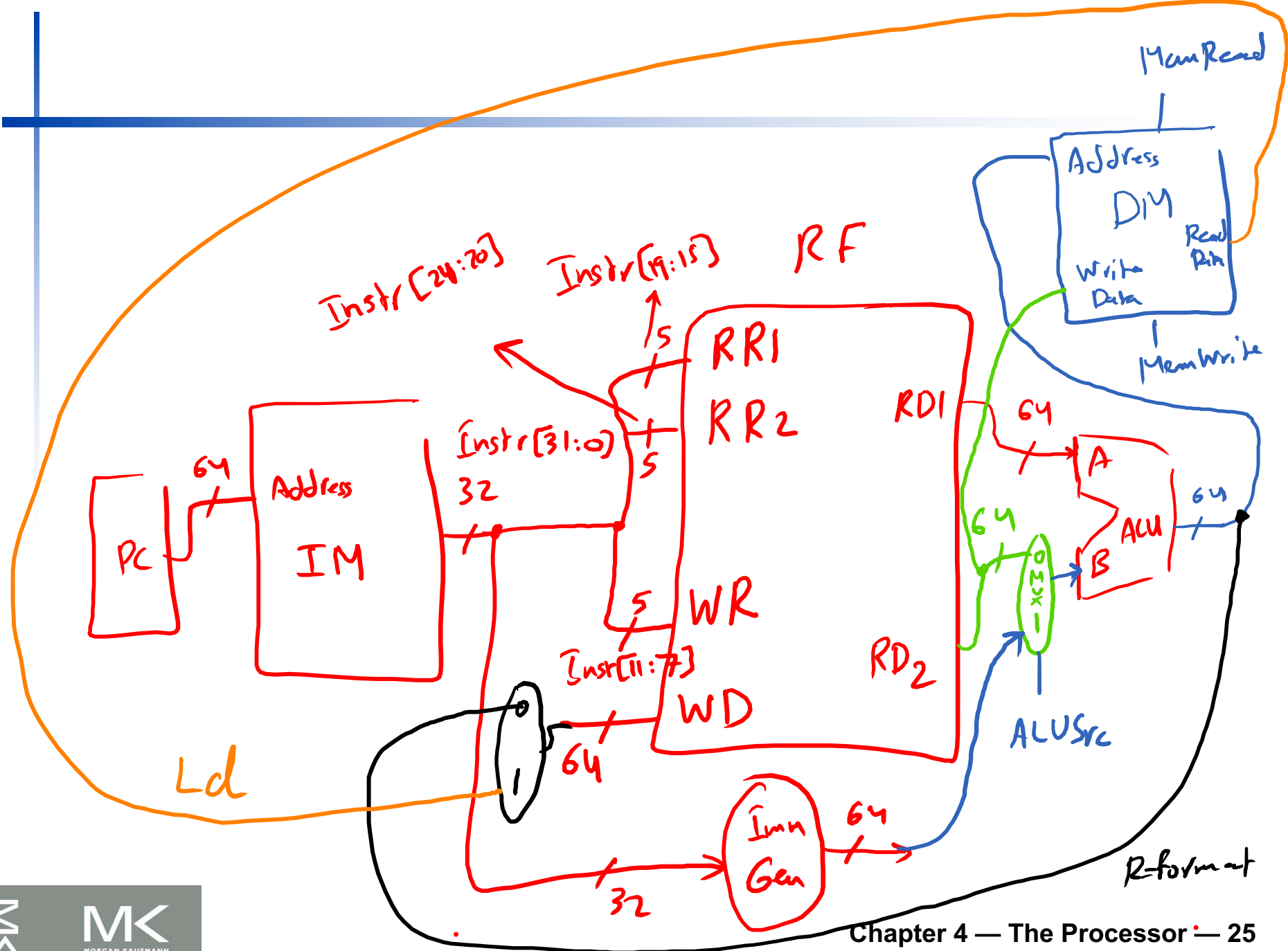


↳ Instr[31:25], Instr[11:7]

② ^{52 times} sign-extension

Replicate Instr[31] 52 times



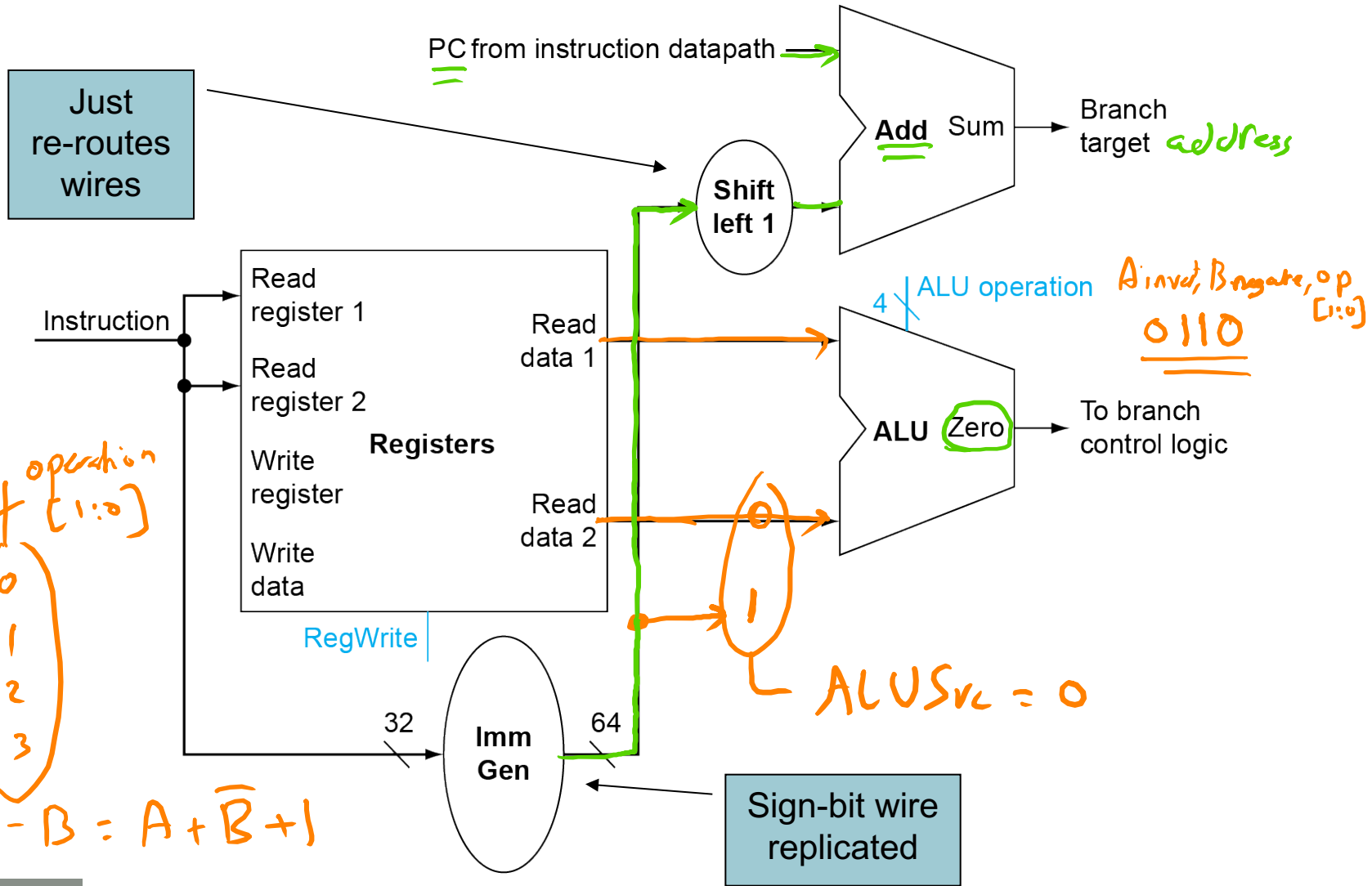


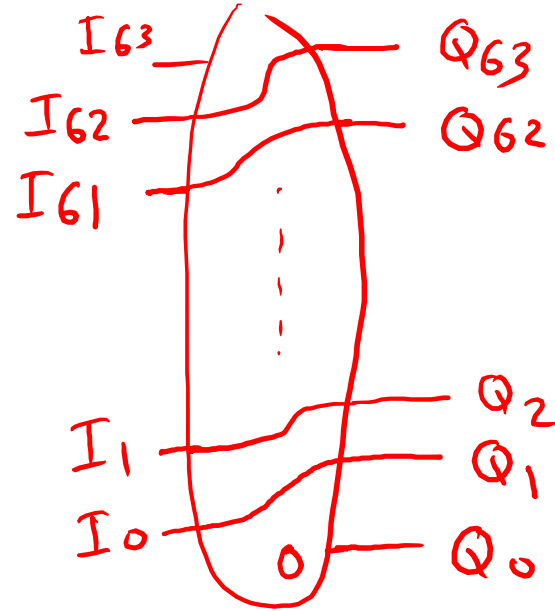
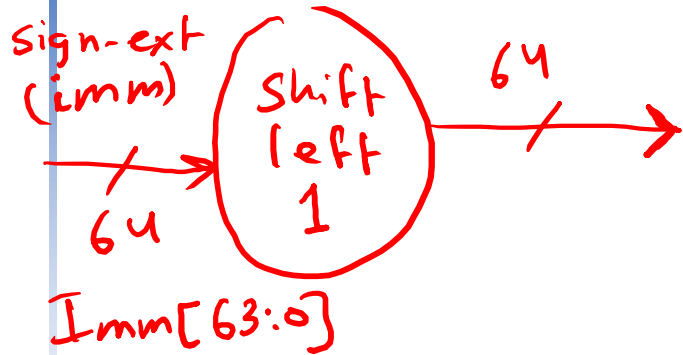
Branch Instructions

"beq"

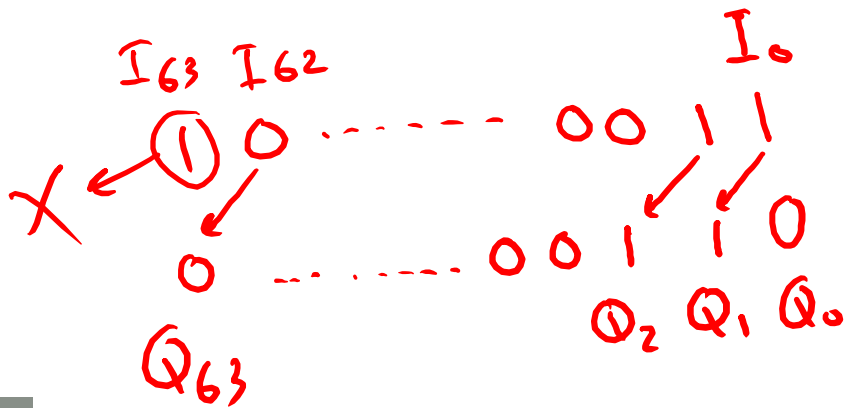
- Read register operands $RD1$ $RD2$
- Compare operands $(RS1) - (RS2)$
 - Use ALU, subtract and check Zero output
 - Taken Branch: Condition is true → Jump to branch target
 $Zero = 1$
 - Not-Taken Branch: Condition is false → Execute instruction next to branch
 $Zero = 0$
- Calculate target address $Branch\ Target\ Address \equiv BTA$
 - Sign-extend displacement $BTA = PC + \text{sign-ext}(imm) \times 2$
 - Shift left 1 place (halfword displacement)
↓
imm Gen
 - Add to PC value

Branch Instructions





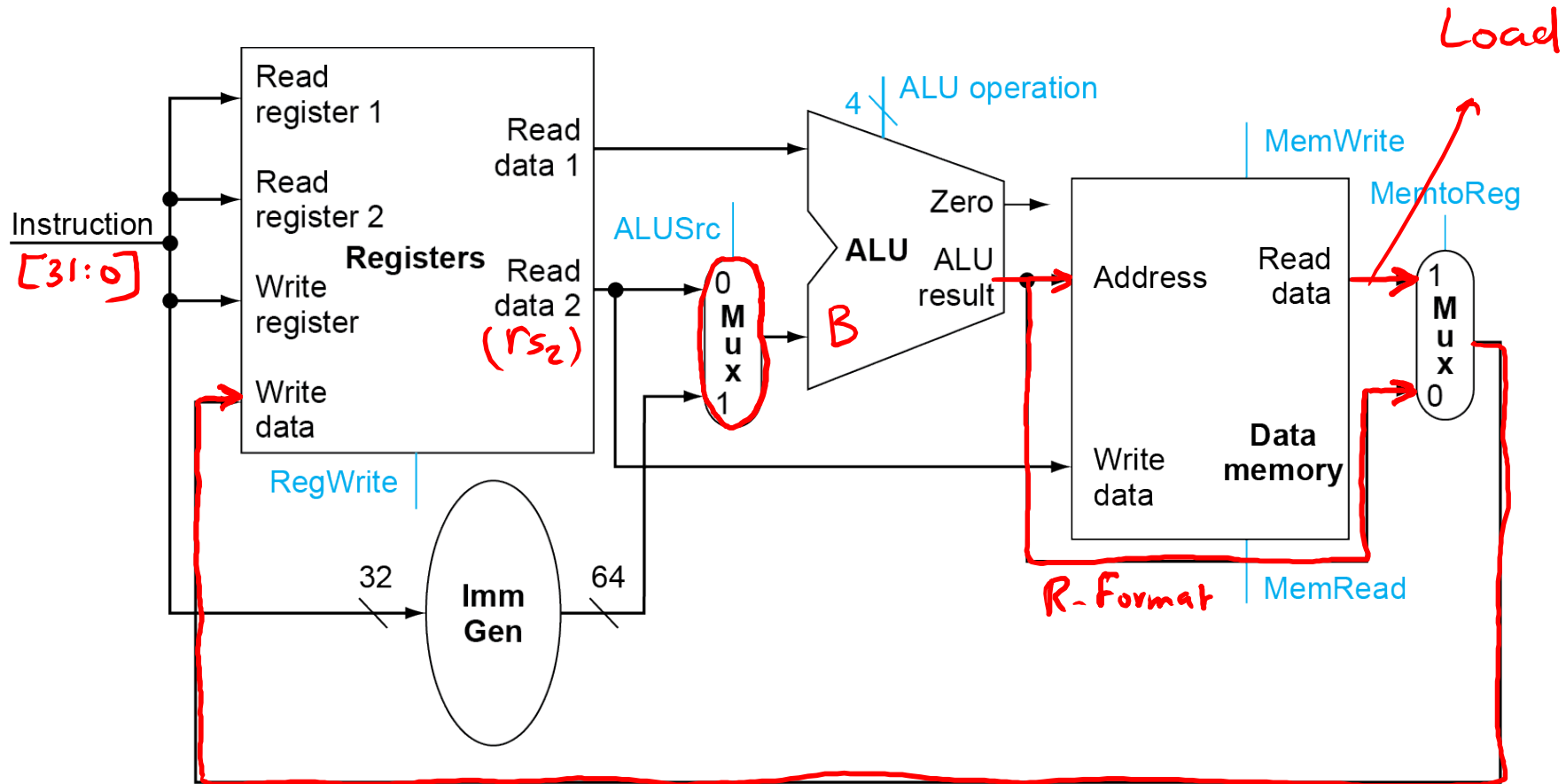
Shift by 1 to the left



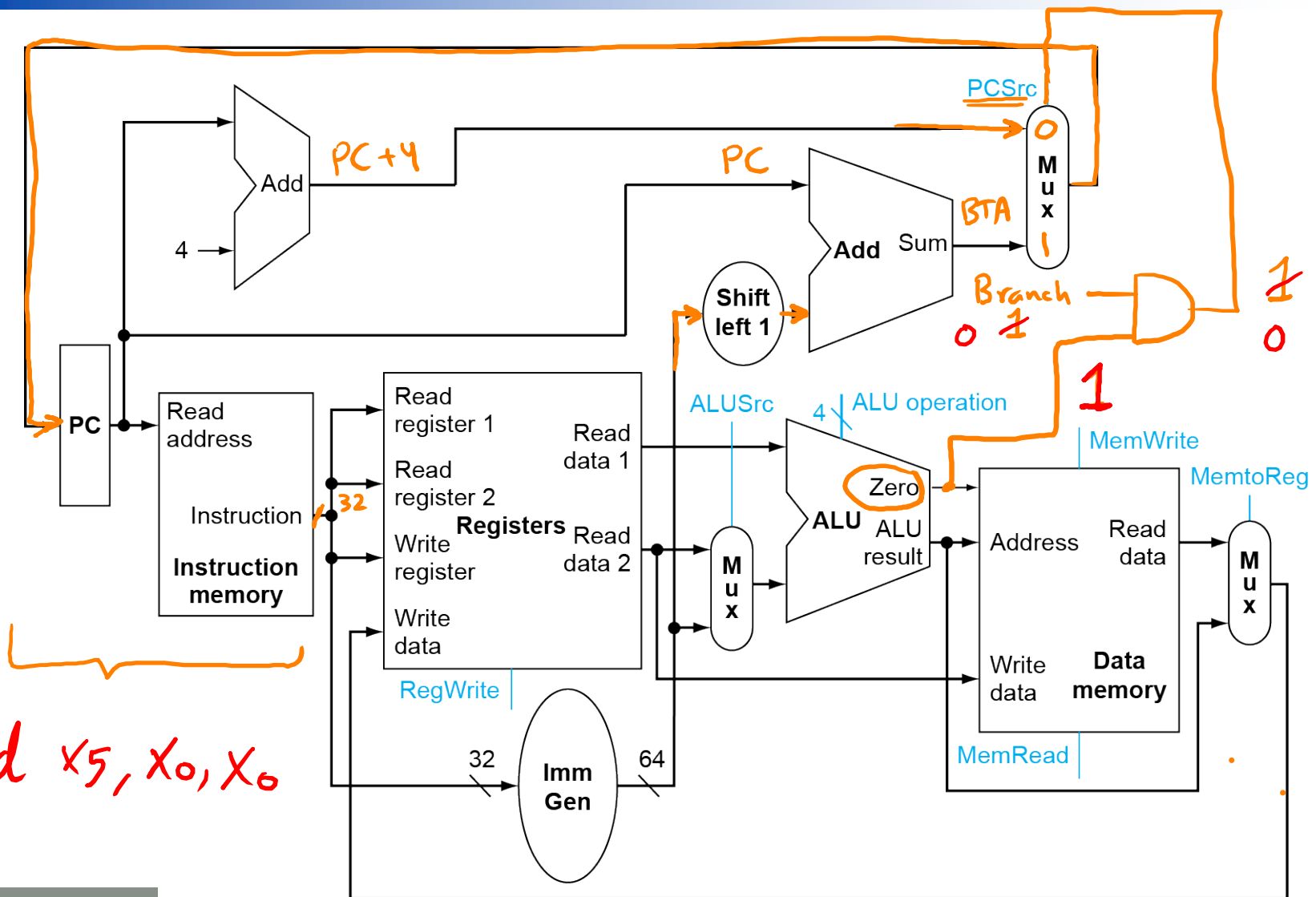
Composing the Elements

- 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 of Single-Cycle CPU



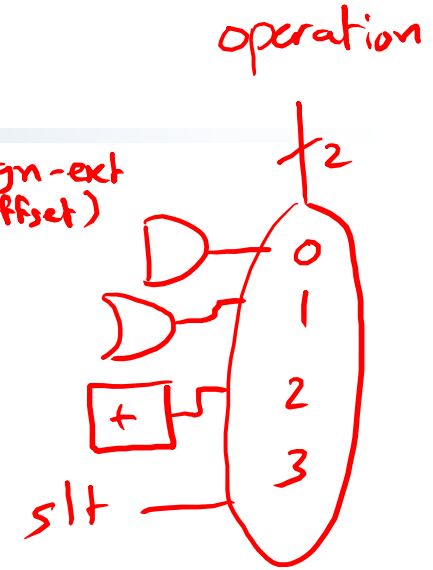
add x5, x0, x0

ALU Control

■ ALU used for

- Load/Store: $F = \text{add}$
- Branch: $F = \text{subtract}$ (rs_1) - (rs_2)
- R-type: F depends on opcode

memory = (rs₁) + sign-ext (offset)



A invert, B negate, Operation[1:0]

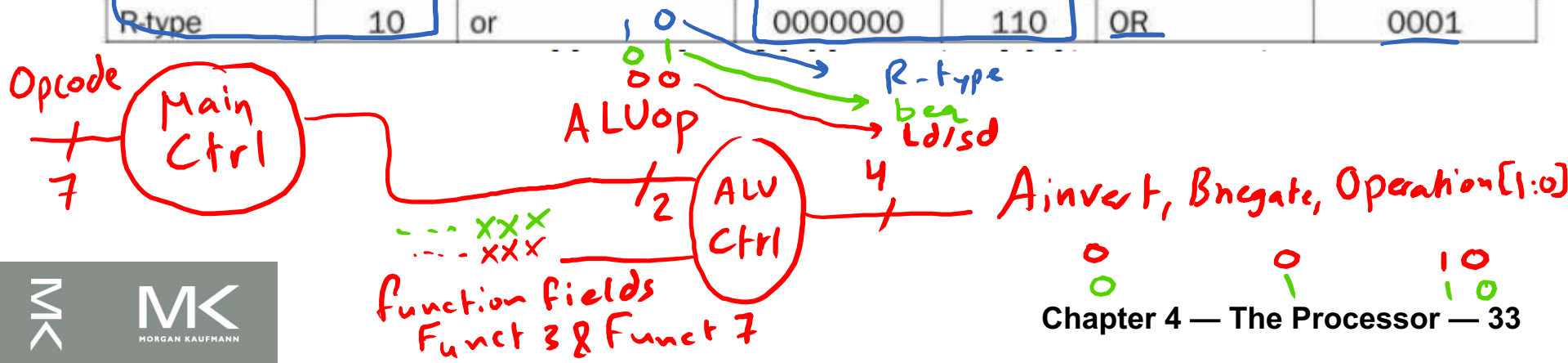
*and
or
add
sub
slt*

ALU control	Function
<u>0000</u>	<u>AND</u>
0001	<u>OR</u>
0010	<u>add</u>
0110	<u>subtract</u>
<i>0111</i>	<i>slt</i>

ALU Control

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

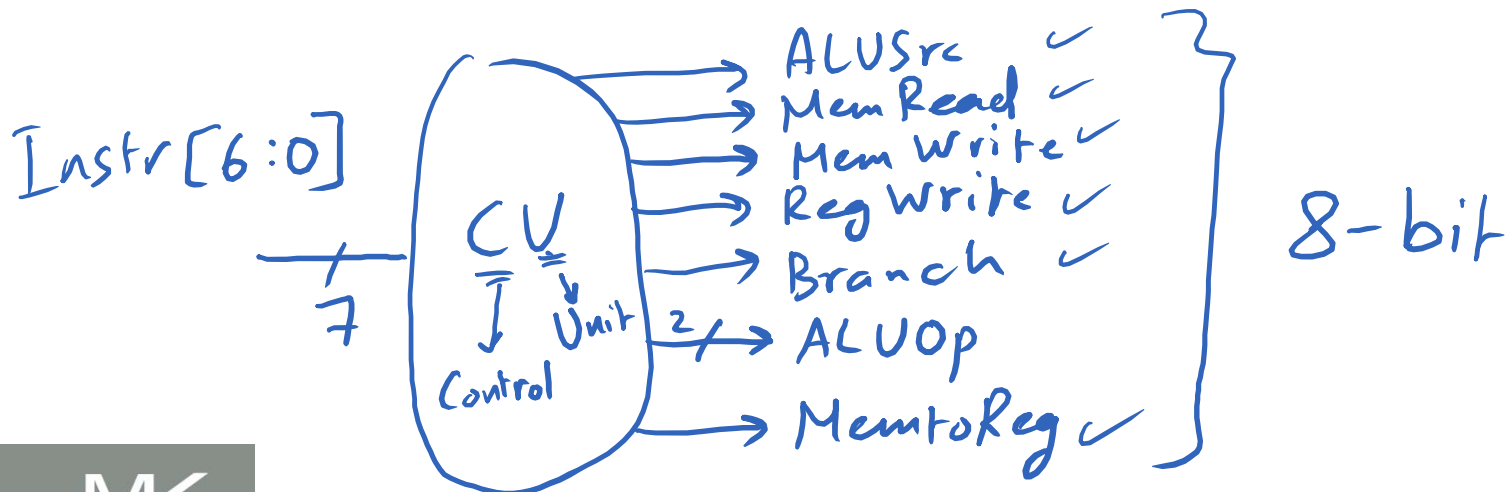
Instruction opcode	ALUOp	Operation	Funct7 field	Funct3 field	Desired ALU action	ALU control input
ld	00	load doubleword	XXXXXXX	XXX	add	0010
sd	00	store doubleword	XXXXXXX	XXX	add	0010
beq	01	branch if equal	XXXXXXX	XXX	subtract	0110
R-type	10	add	0000000	000	add	0010
R-type	10	sub	0100000	000	subtract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



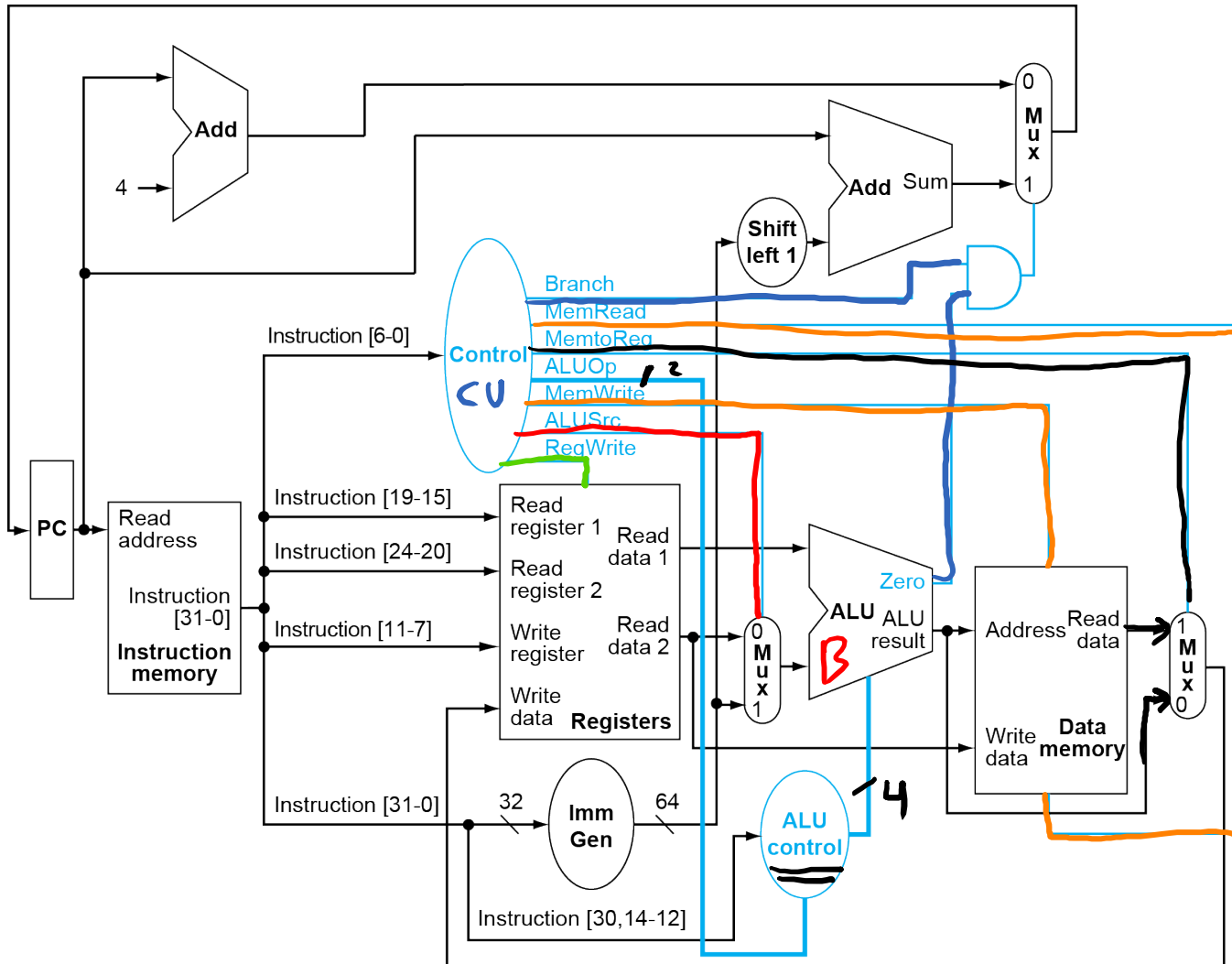
The Main Control Unit

- Control signals derived from instruction's opcode

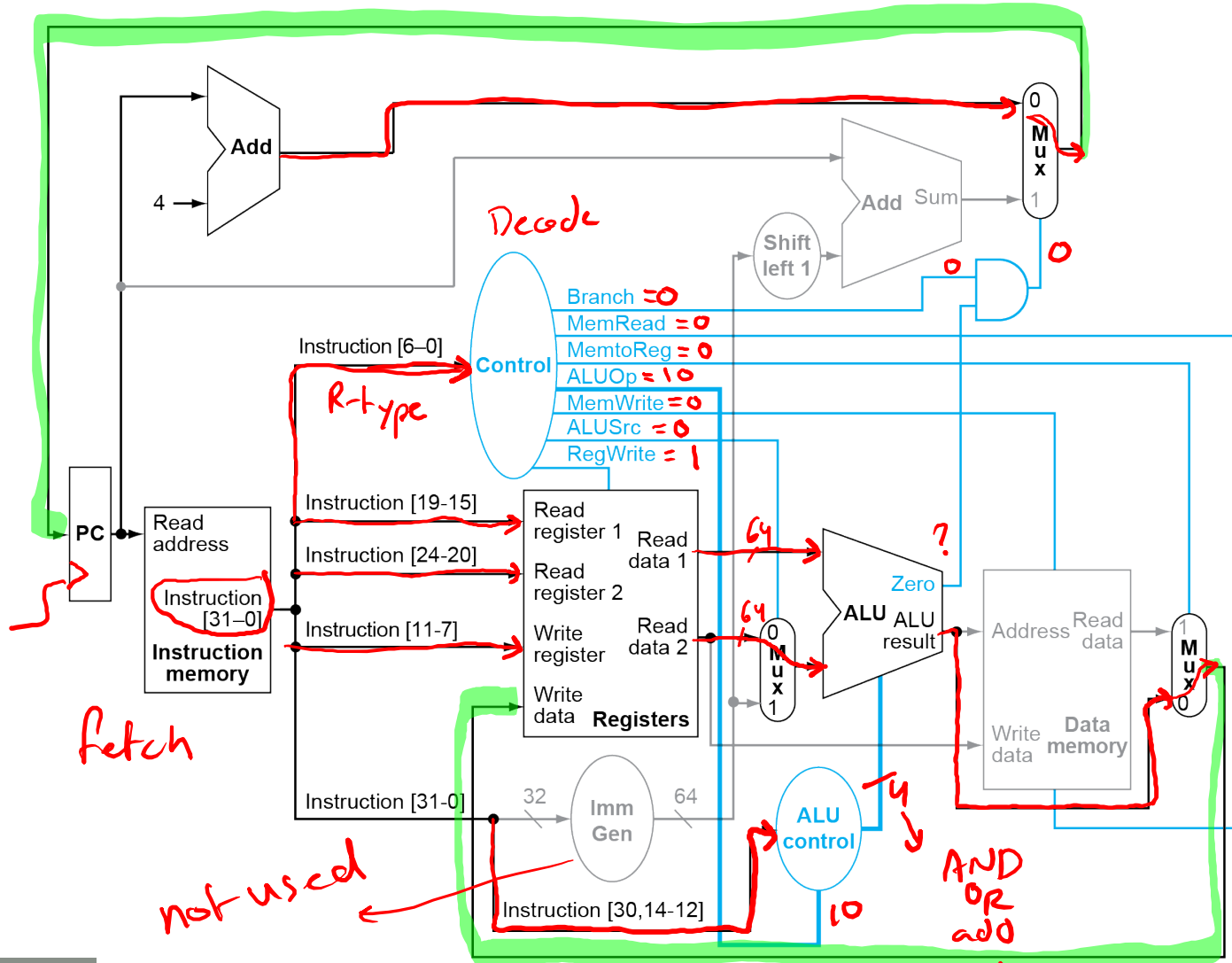
Name (Bit position)	31:25	24:20	19:15	14:12	11:7	6:0
(a) R-type	funct7	rs2	rs1	funct3	rd	opcode
(b) I-type	immediate[11:0]		rs1	funct3	rd	opcode
(c) S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode
(d) SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode



Datapath With Control

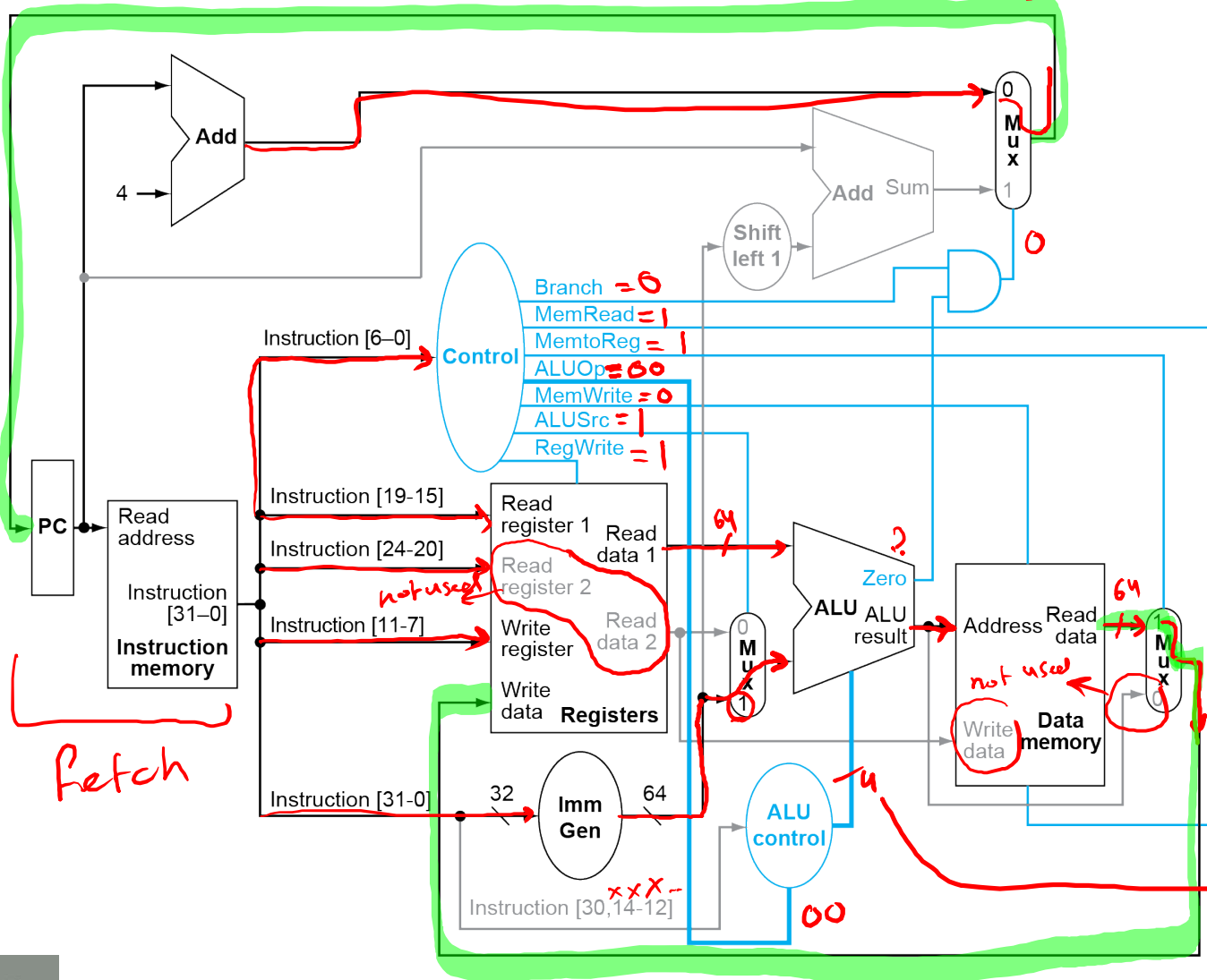


R-Type Instruction



Load Instruction

Ld rd, offset(rs1)
memory address = (rs1) + sign-ext(offset)

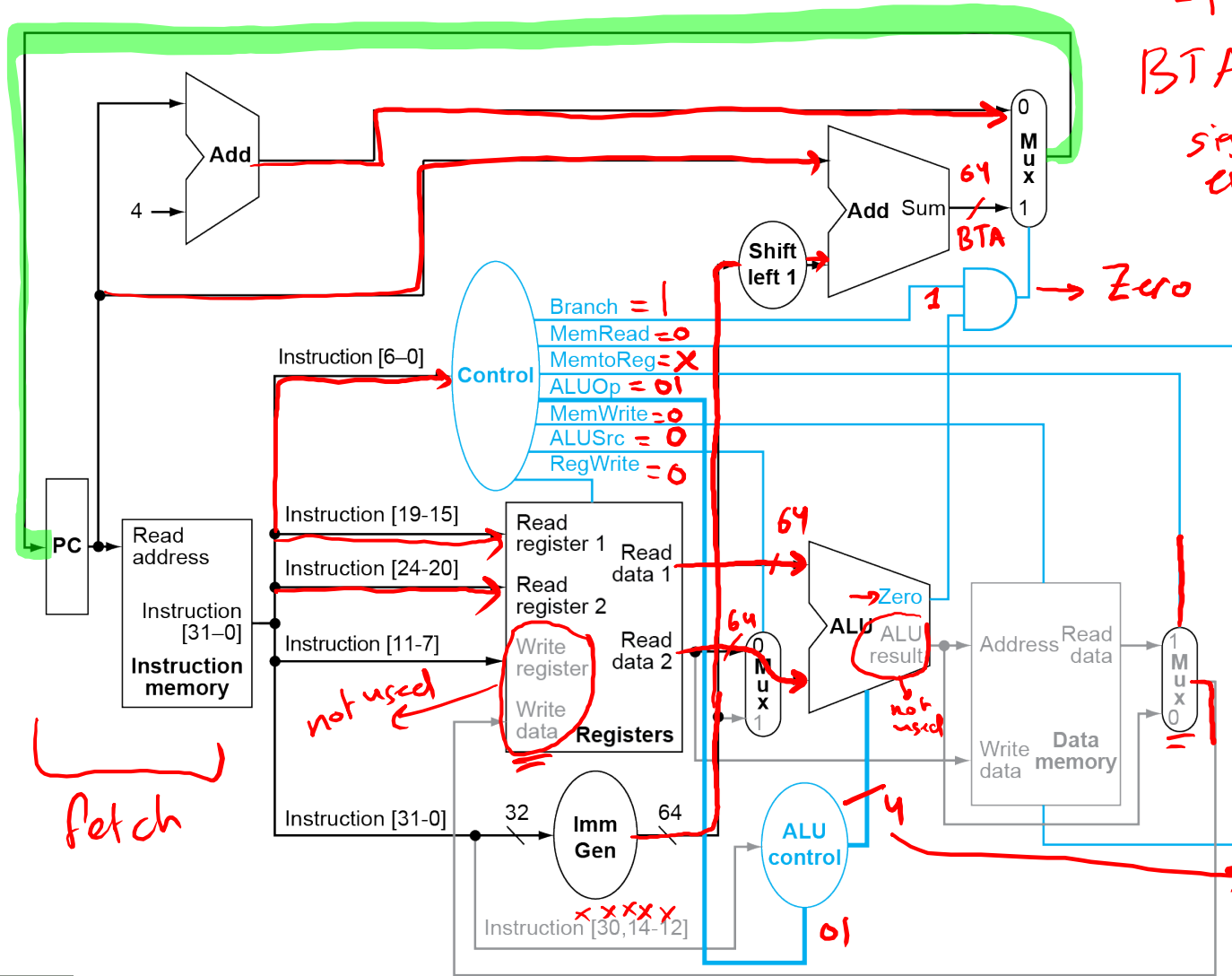


Fetch

*addition
0010*

BEQ Instruction

beq rs₁, rs₂, Label
rs₁ - rs₂
BTA = PC +
*sign(Label) * 2*



fetch

not used

not used

subtraction 0110

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.

Branch & Zero

of RF

Control Unit

Main

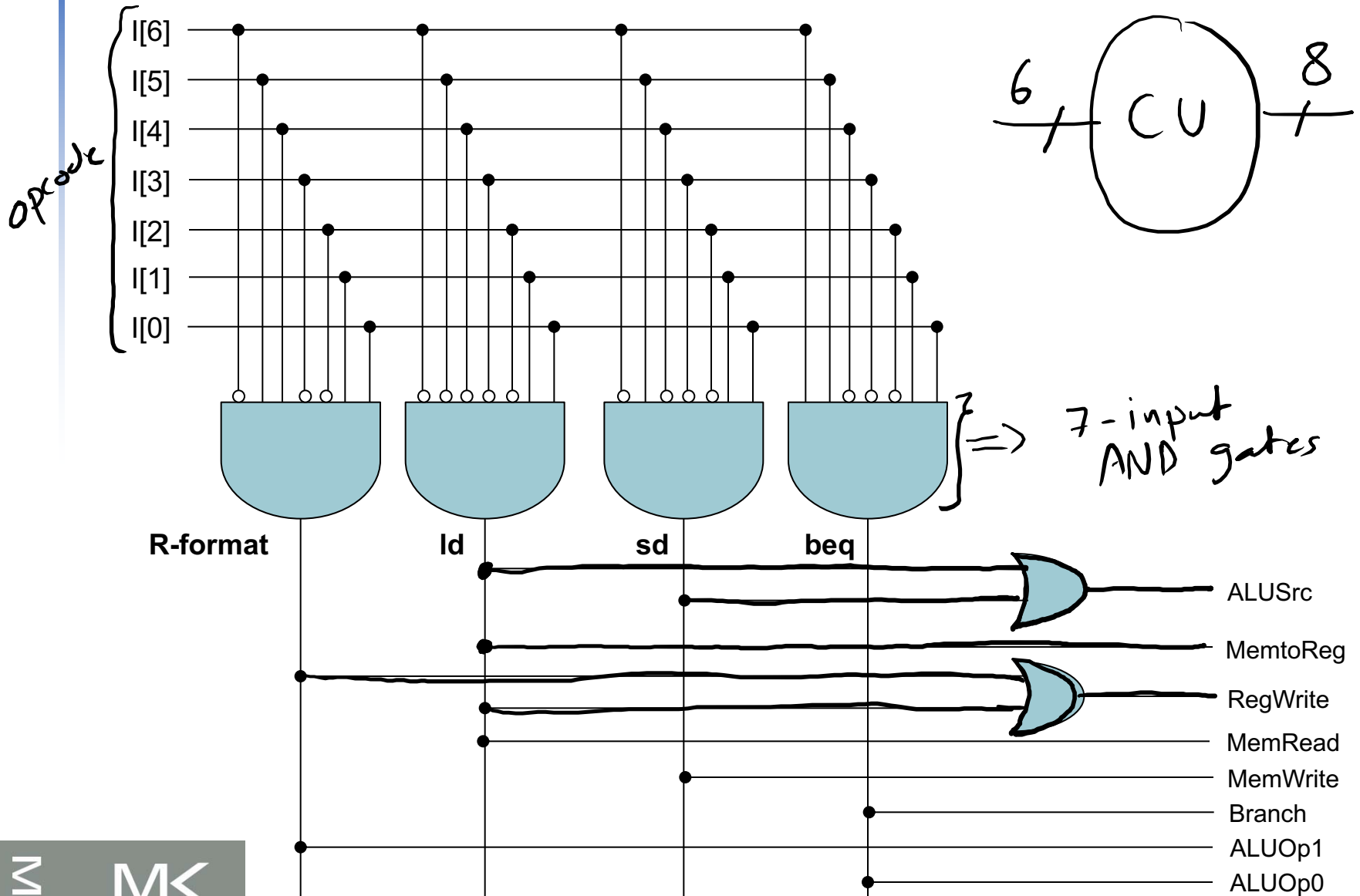


- The input is the Opcode field (7 bits) from the instruction register
- The output is 8 control signals
 - Control signals can be set solely based on the opcode field, except PCSrc (i.e. PCSrc = Branch AND Zero)

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*	

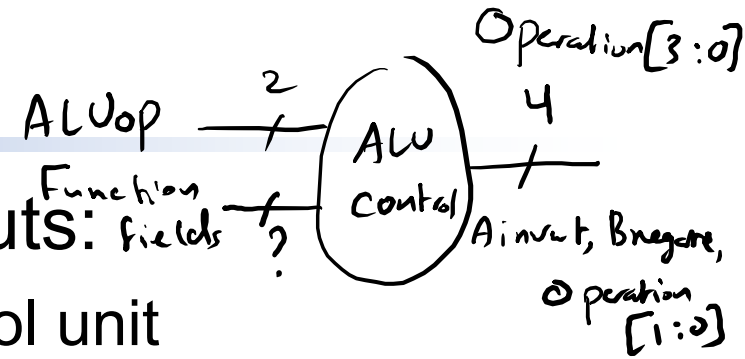
ALUOp1
 ↑
 00 (+)
 01 (-)
 10 Rtype

Control Unit Design



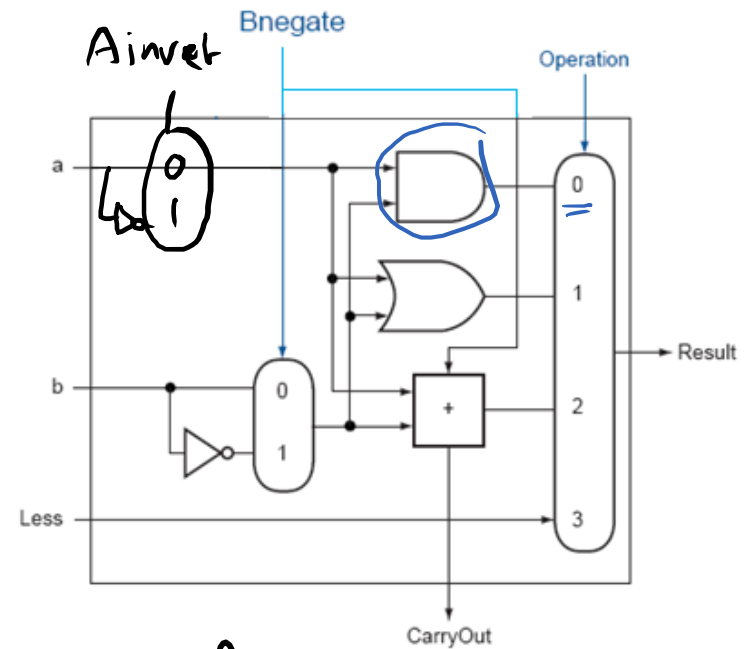
ALU Control

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



Function	<u>Ainvert</u> Operation [3]	<u>Bnegate</u> Operation [2]	<u>Operation</u> [1:0]
→ and	0	0	00
→ or	0	0	01
→ add	0	0	10
→ sub	0	1	10

slt 0 1 11
 nor 1 1 00

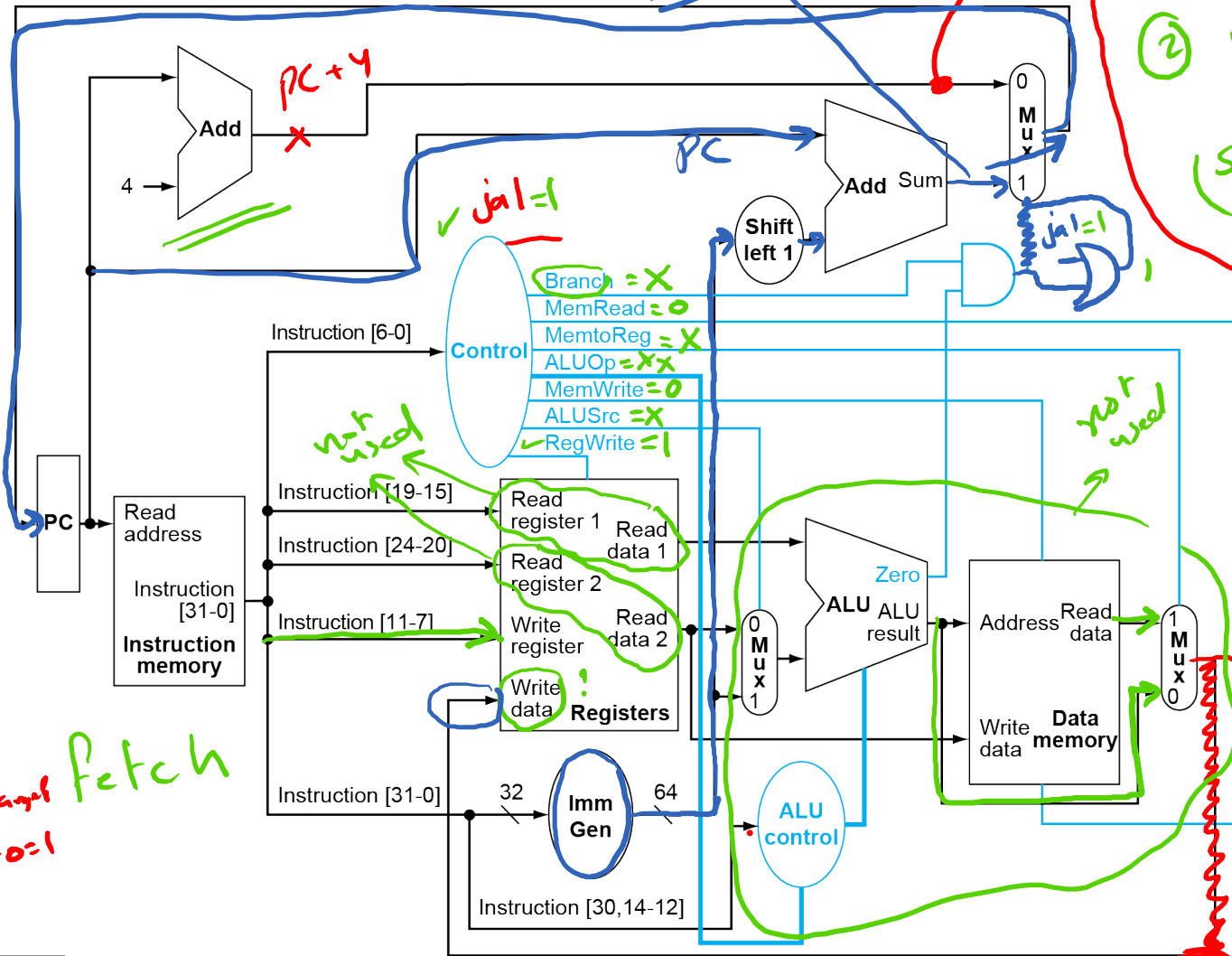


ALU Slice

Datapath With Control

BTA or "jalTA"

jal rd, label
 ① $(vd) = PC + 4$
 ② $PC = Target = PC + (sign) label \times 2$

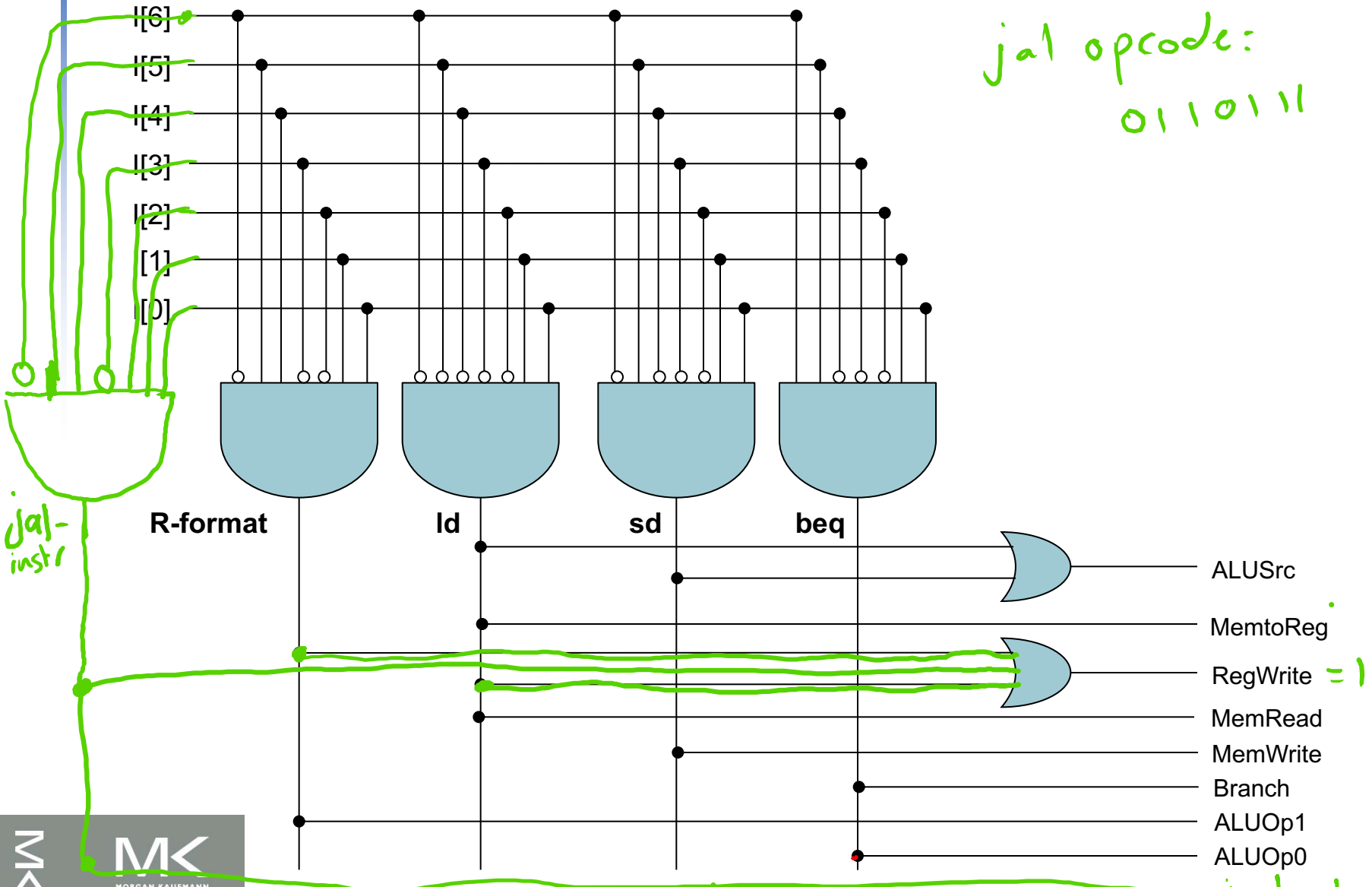


instr	jal
ld	0
sd	0
R-type	0
beq	0

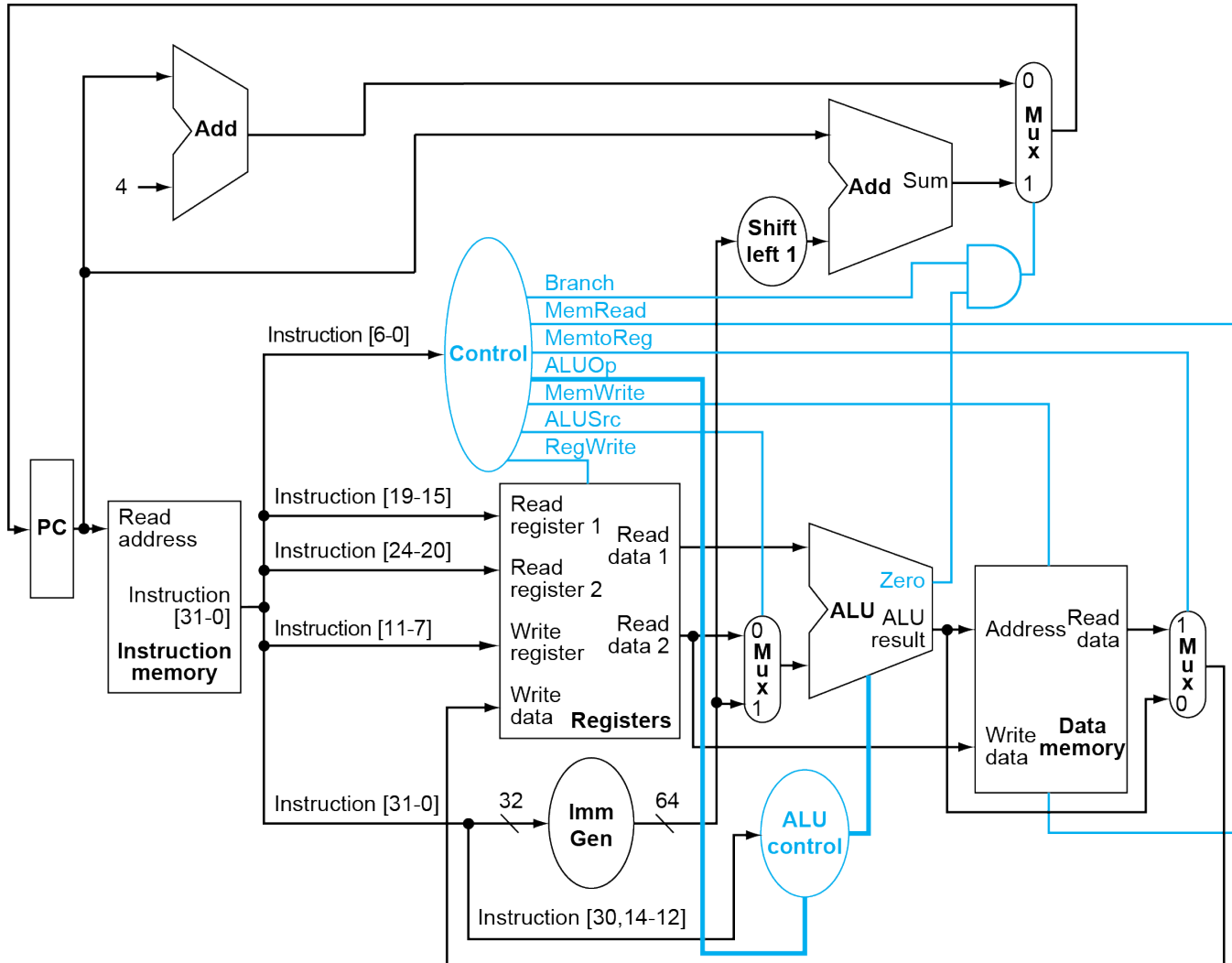
PC = PC + 4
 PC = Target
 zero = 0
 Fetch



Control Unit Design

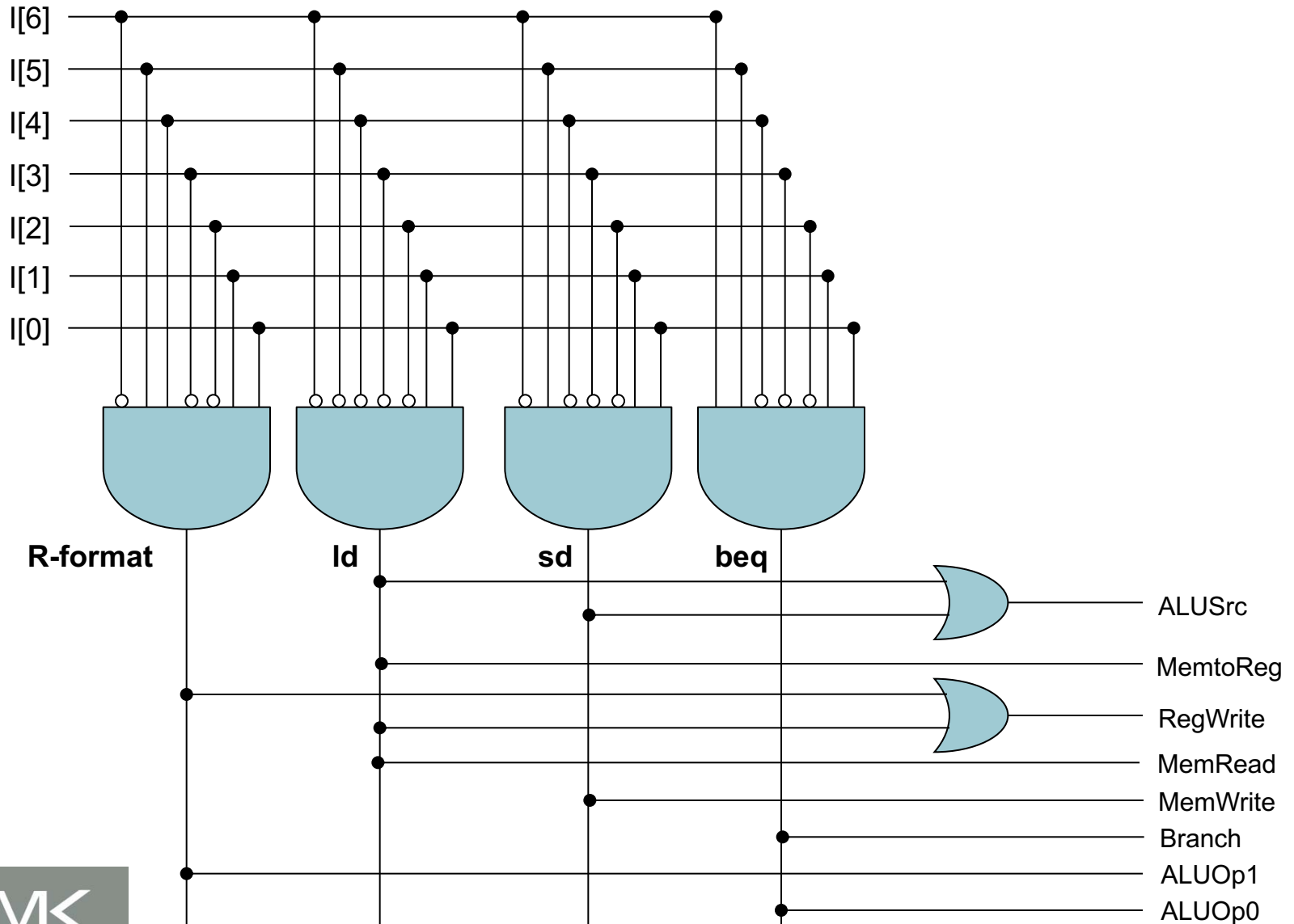


Datapath With Control *"addi"*



Control Unit Design

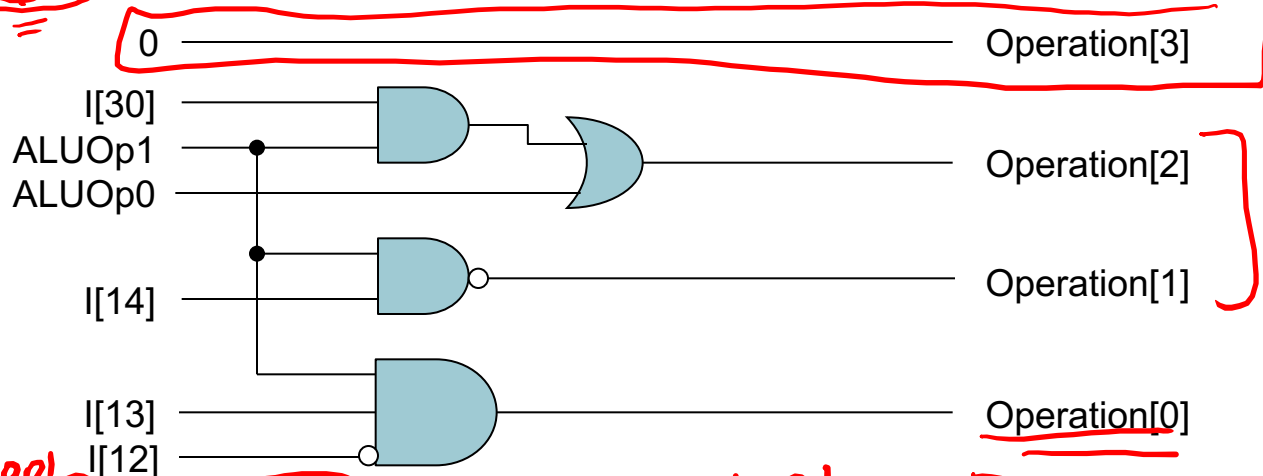
"addi"



ALU Control (New Instruction Example)

ALUOp		Funct7 field							Funct3 field			Operation
ALUOp1	ALUOp0	I[31]	I[30]	I[29]	I[28]	I[27]	I[26]	I[25]	I[14]	I[13]	I[12]	
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*

1 | 1 | X X - X - X - X X X | 0 1 1 1 | 0 0 1*



HW

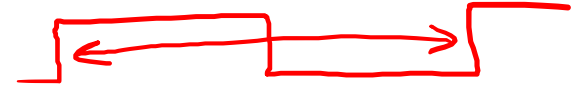
ALUOp1
ALUOp0

I[13]
I[12]

ALUOp1
I[13]
I[12]

operation[0]

Performance Issues



- Longest delay determines clock period

- Critical path: load instruction

- ^① Instruction memory → ^② register file → ^③ ALU → ^④ data memory → ^⑤ register file

sd: ①, ②, ③, ④
R-type: ①, ②, ③, ⑤

- Not feasible to vary period for different instructions

beq: ①, ②, ③

- Violates design principle

- Making the common case fast

- We will improve performance by pipelining

Pipelining

- An implementation technique in which multiple instructions are overlapped in execution



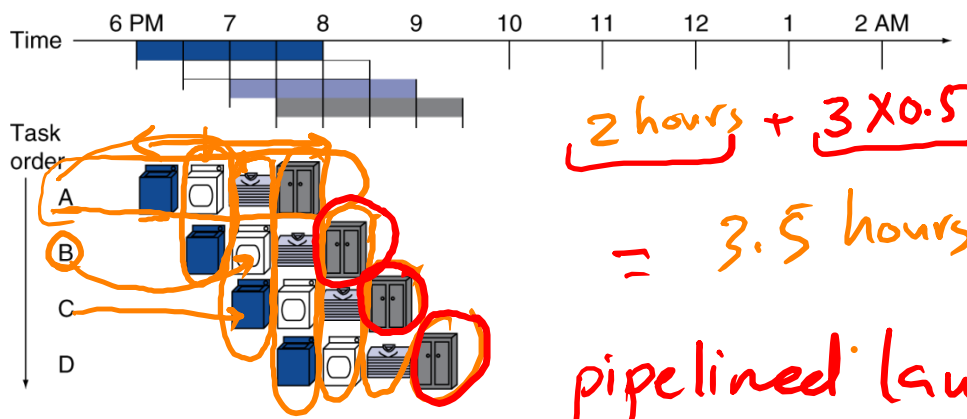
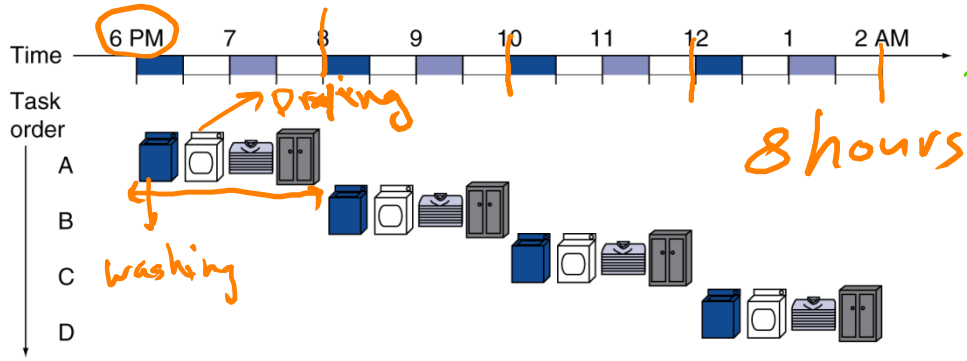
Compared with C.f. Single-Cycle and Multi-Cycle implementations in which only a single instruction is executing at any time

- *Today, pipelining is nearly universal*

Pipelining Analogy

$$\lim_{n \rightarrow \infty} \frac{2n}{0.5n + 1.5} = 4$$

- Pipelined laundry: overlapping execution
 - Parallelism improves performance



Four loads:

- Speedup
= $8 / 3.5 = 2.3$

Non-stop:

- Speedup
= $2n / (0.5n + 1.5) \approx 4$
= number of stages

pipelined laundry is 2.3 times faster than non-pipelined laundry

$$2 + (n-1) \times 0.5$$

RISC-V Pipeline

5-stage 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

$$CC_{SC} = 200 + 100 + 200 + 200 + 100 = 800 \text{ ps}$$

$$+ \text{Memory} + \text{RF} = \text{Fetch} + \text{RF} + \text{AW}$$

$$CC_{SC} = \text{delay of load}$$

$$CC_{MC} = 200 \text{ ps}$$

ALU, Memory

$$CC_{\text{pipeline}} = 200 \text{ ps}$$

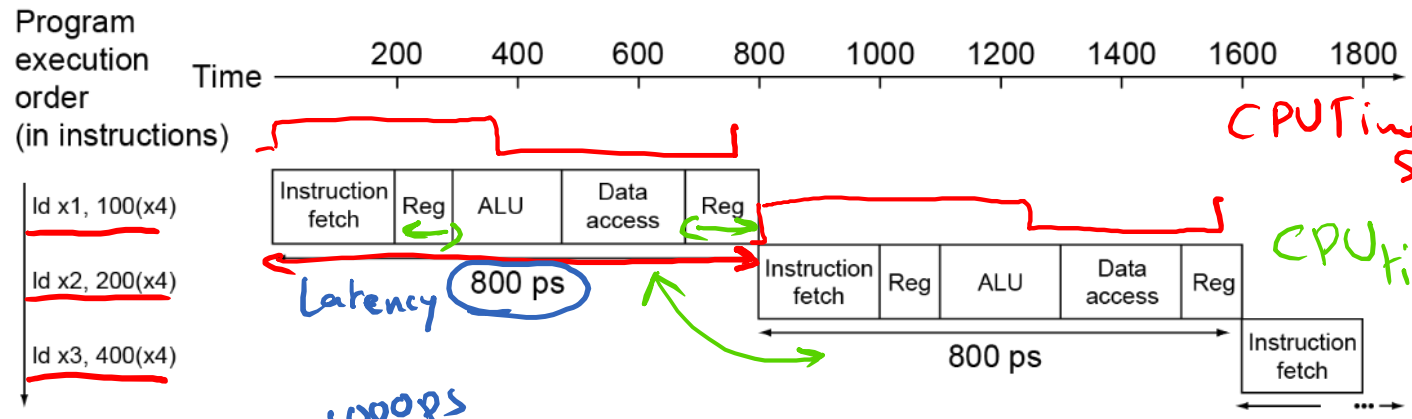
- Assume time for stages is
 - 100ps for register read or write
 - 200ps for other stages
- Compare pipelined datapath with single-cycle datapath

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	X	700ps
R-format	200ps	100 ps	200ps	X	100 ps	600ps
beq	200ps	100 ps	200ps	X	X	500ps

Pipeline Performance

$(1000) / (5) = 200$

Single-cycle ($T_c = 800\text{ps}$)



3 in 2400ps

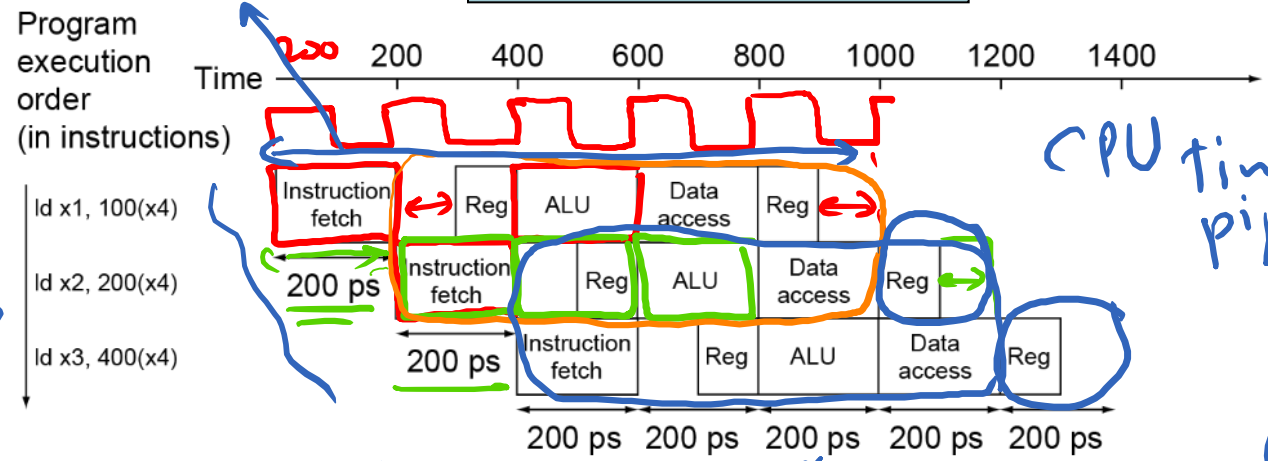
$CPU\ Time_{SC} = 2400\text{ps}$

$CPU\ Time_{SC} = 2000\text{ps}$

Latency = 1000ps

$\frac{2000}{1400} = \frac{15}{7} = 2.1$

Pipelined ($T_c = 200\text{ps}$)



3 in 1400ps

$CPU\ Time_{pipeline} = 1000 + 2 \times 200 = 1400\text{ps}$

pipeline is $\frac{2400}{1400} = \frac{12}{7} = 1.7...$ faster than SC

Non-stop load instructions $\Rightarrow n$ instructions

$$\text{CPU time}_{SC} = \frac{800 \times n}{1000 \times n} \Rightarrow \text{for balanced stages}$$

$$\begin{aligned} \text{CPU time}_{\text{pipeline}} &= \frac{5 \times 200}{1000} + (n-1) \times 200 \\ &= 1000 + (n-1) \times 200 \end{aligned}$$

$$= 800 + 200 \times n$$

$$\lim_{n \rightarrow \infty} = \frac{800 \times \cancel{n}}{800 + 200 \times \cancel{n}} = \frac{800}{200} = 4$$

pipeline is 4 times faster than SC

Pipeline Speedup

① If all stages are balanced

■ i.e., all take the same time

■ Time between instructions_{pipelined}
= $\frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of stages}}$

② Non stop instructions

Speedup = number of stages

③ No Stalls

■ If not balanced, speedup is less

■ Speedup due to increased throughput

number of instruction per unit of time

■ Latency (time for each instruction) **does not decrease**

→ Stays the same
or
→ increases

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

CISC



Hazards

- Situations that prevent starting the next instruction in the next cycle

① Structure hazards

- A required resource is busy

② Data hazard

- Need to wait for previous instruction to complete its data read/write

③ Control hazard

- Deciding on control action depends on previous instruction

Structure Hazards: structural hazards

Conflict for use of a resource

multiple instructions trying to use the same resource at the same time

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”

IF
MEM

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

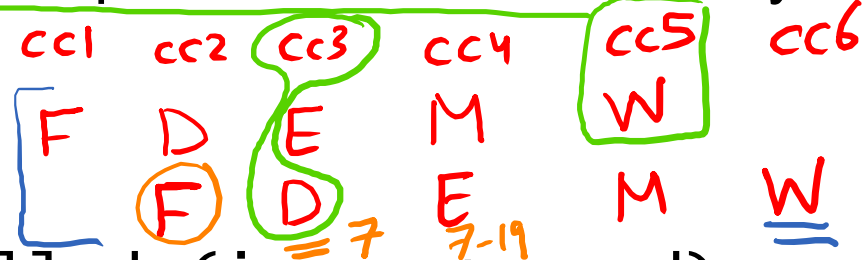
write in cc5
read in cc3

read before
write
there is a hazard

- An instruction depends on completion of data access by a previous instruction

```

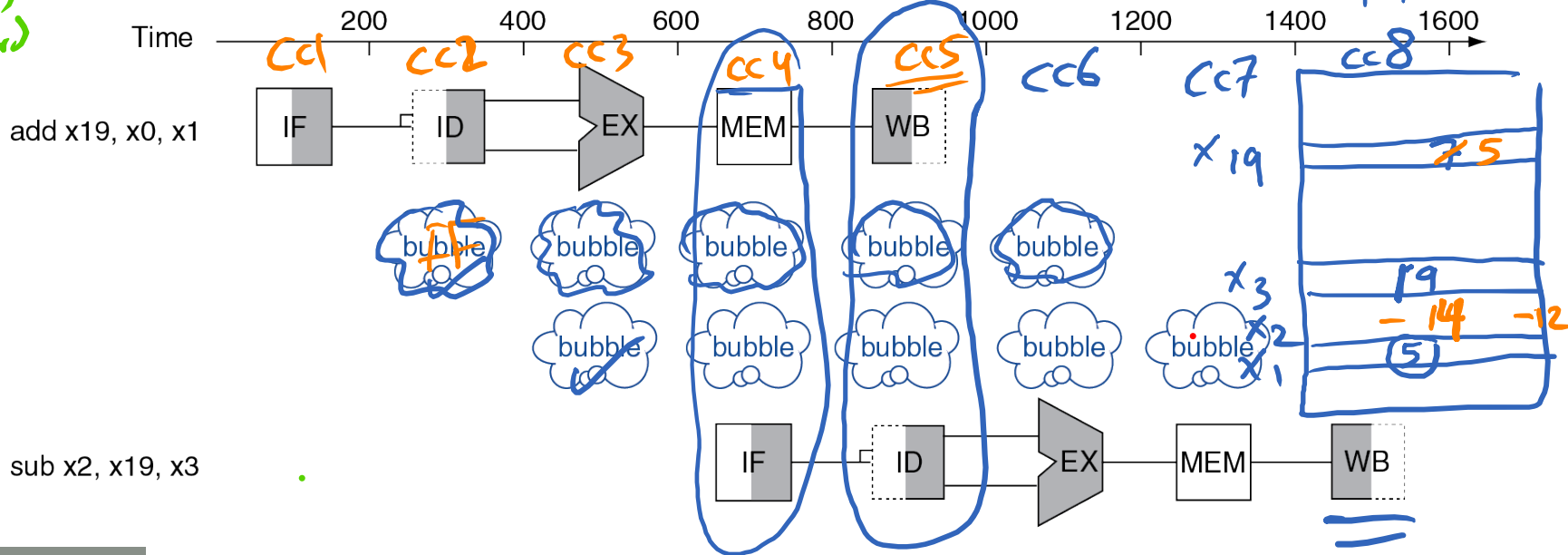
add rd, x19, x0, x1
sub x2, x19, x3
    
```

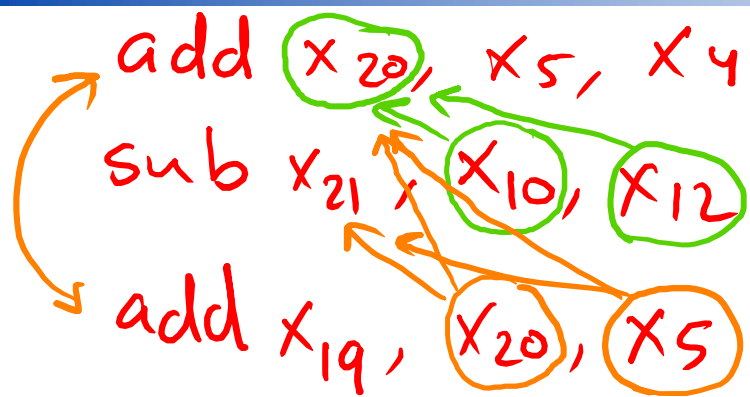


Pipeline must be stalled (i.e. stopped)

- Pipeline stall \equiv bubble

read after write hazard
(RAW) hazard

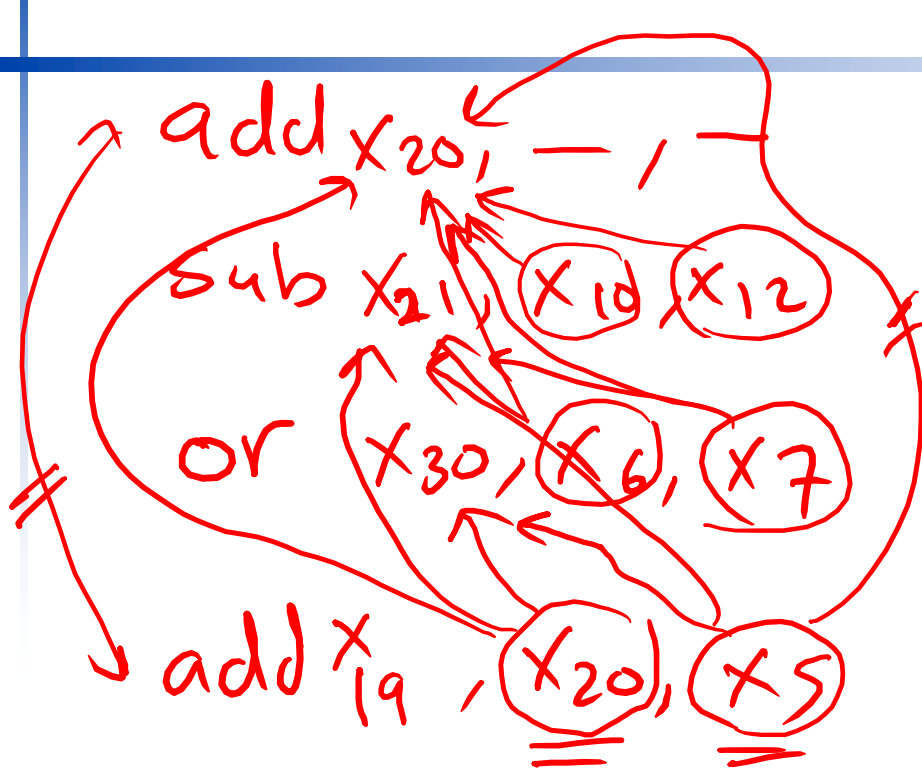




cc1	cc2	cc3	cc4	cc5	cc6	cc7
F	D	E	M	W		
	F	D	E	M	W	
		F	D	E	M	W



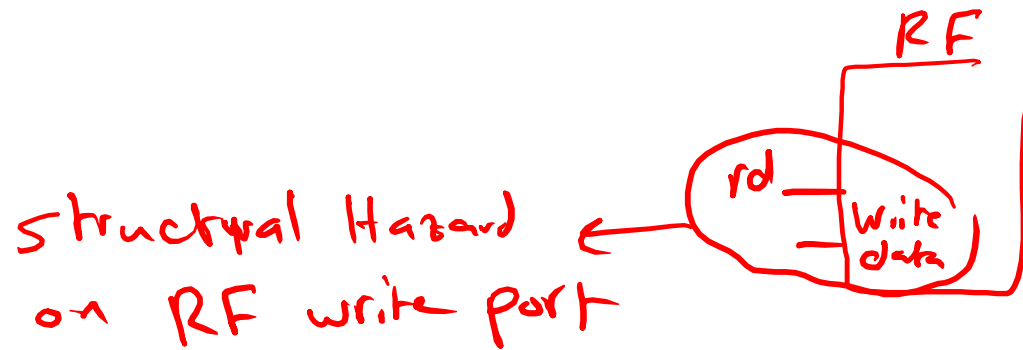
add	F	D	E	M	W	
sub		F	D	E	M	W
add				F	D	

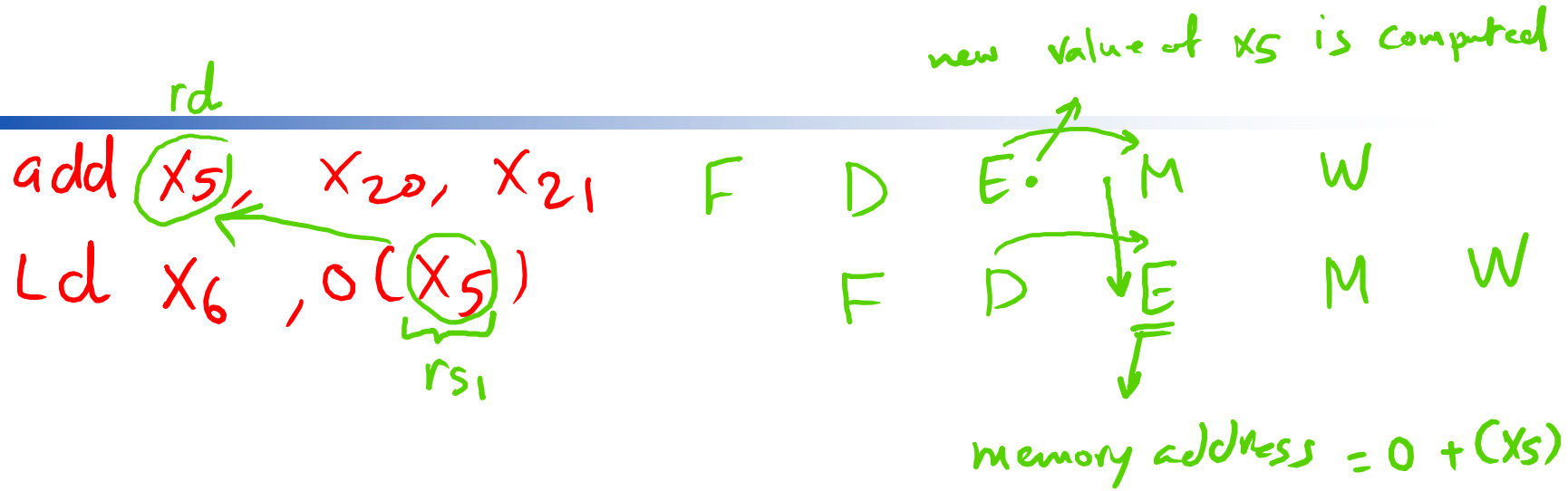


					cc5
F	D	E	M	(W)	
F	D	E	M	W	
F	D	E	M	W	
F	(D)	E	M	W	

WAW : Write After Write } do NOT
 WAR : " " Read } happen
 in the
 5 stage
 RISC-V pipeline

In pipelining, all instructions go through the five stages to avoid structural Hazards.





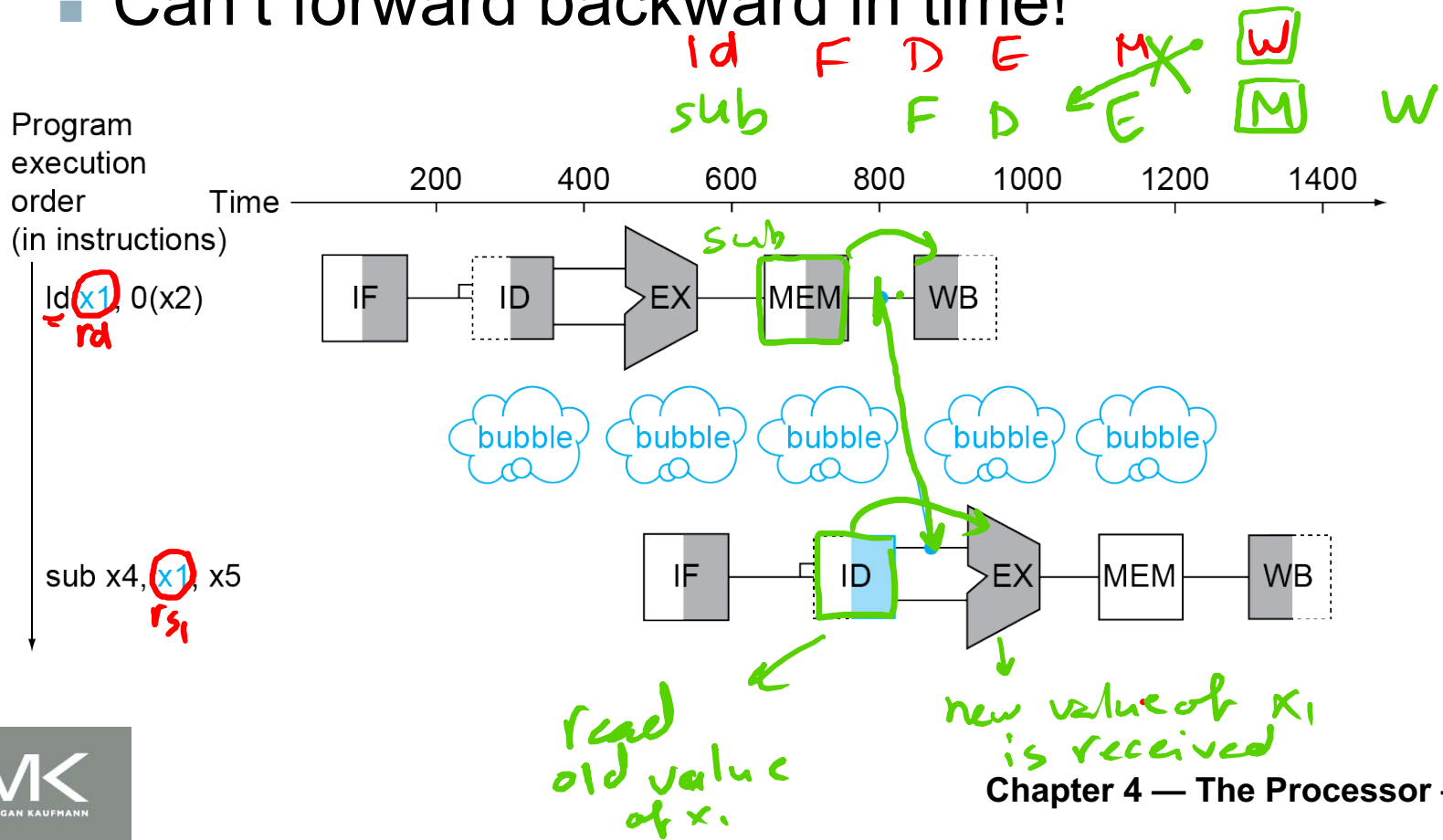
Which instructions can forward data & From which stages?

- ① R-type can forward from MEM or WB stages
- ② Load can forward from WB.

* Destination stage in forwarding is always the EX stage

Load-Use Data Hazard

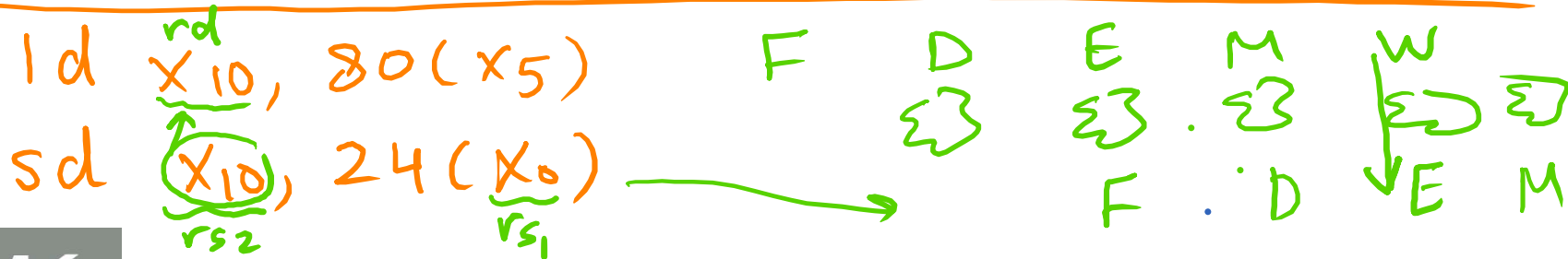
- Can't always avoid stalls by forwarding
 - If value not computed when needed
 - Can't forward backward in time!





old values of x₆ & x₇

- Fwd - W - to - EX - rs₁
- Fwd - M - to - E - rs₂



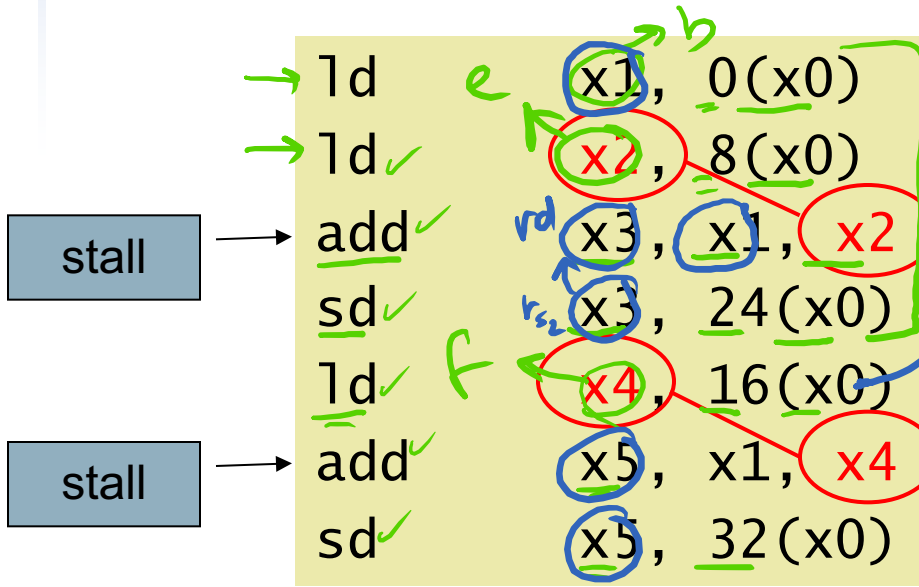
Code Scheduling to Avoid Stalls

- Load
- Arithmetic
- Store

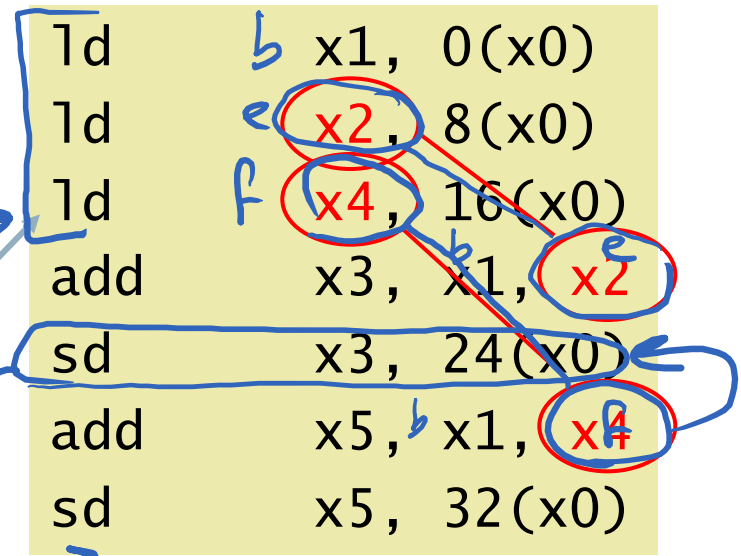
- Reorder code to avoid use of load result in the next instruction

$a = b + e;$
 $c = b + f;$

- C code for $a = b + e;$ $c = b + f;$
- Assume that a, b, c, e, f are in memory with offsets $(24, 0, 32, 8, 16)$ from $x0$



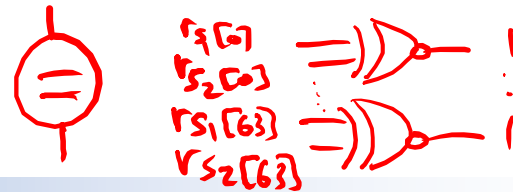
13 cycles



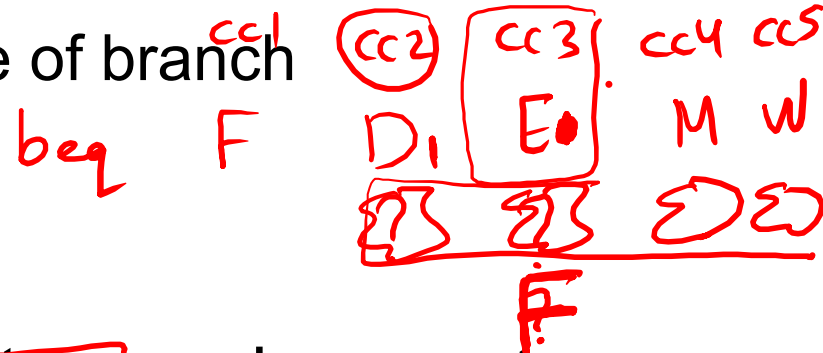
11 cycles

first instruction → 5 cycles
6 cycles

Control Hazards

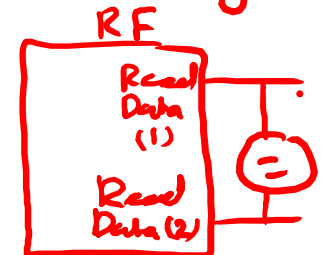


- Branch determines flow of control
 - Fetching next instruction depends on branch outcome
 - Taken → Fetch Target
 - NOT Taken → Fetch PC+4
 - Pipeline can't always fetch correct instruction
 - Still working on ID stage of branch



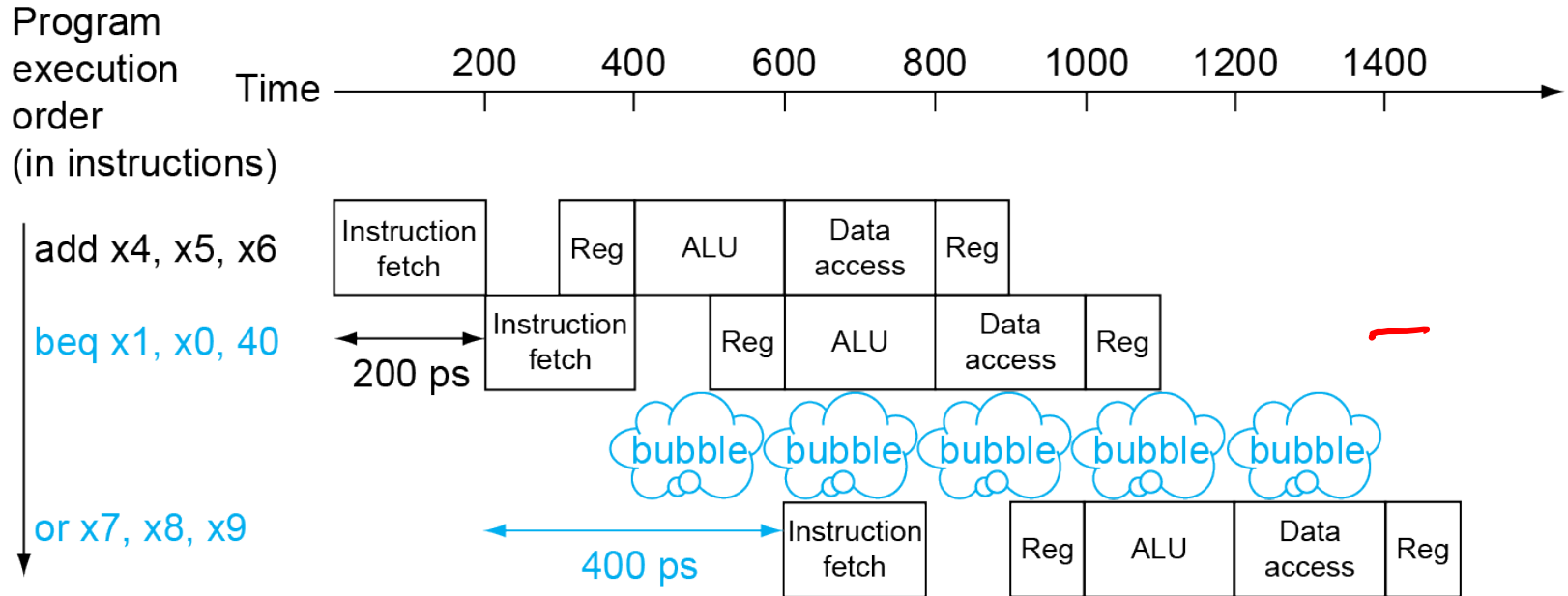
- In RISC-V pipeline
 - Need to compare registers and compute target early in the pipeline "Decode stage"
 - Add hardware to do it in ID stage

$$BTA = PC + imm \times 2$$



Stall on Branch

- Wait until branch outcome determined before fetching next instruction



100 branch instructions → 100 stalls

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 ^{always} not taken

- Fetch instruction after branch, with no delay

Branch predictor → with 50% accuracy
100 branches → 50 cycles as lost

Taken → Fetch Target after branch
Not Taken → Fetch PC+4 after branch

Predictor Types

① Always Taken Predictor
↳ Not Taken Predictor

② Static Predictor

③ Dynamic Predictor

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

predict
Taken ←

beq - -
add
sub
ld

beq
↓
predict Not Taken

predict as "Not Taken"

will be executed

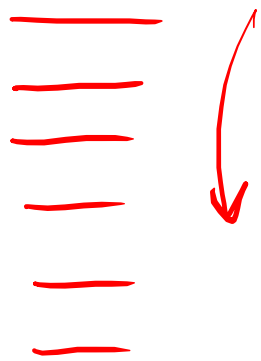
n+1 times

n-times "not Taken" one-time Taken

n-times

"predict Taken"
100%

Loop: beq rs1, rs2, Exit



beq x0, x0, loop

Exit:

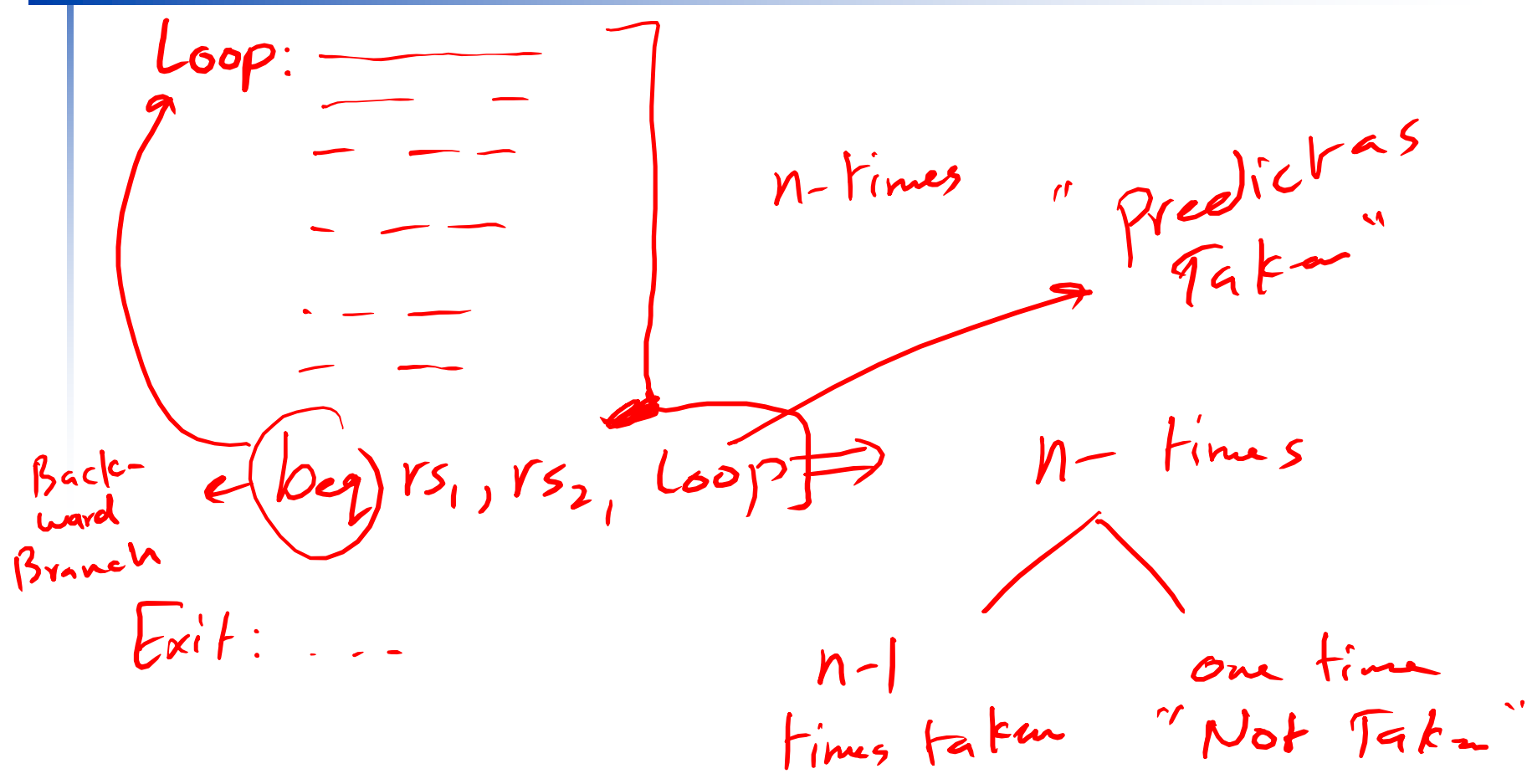
Forward Branch

n-times =>

Prediction is correct

one-time =>

prediction is wrong and one cycle is wasted



RISC-V Pipelined Datapath

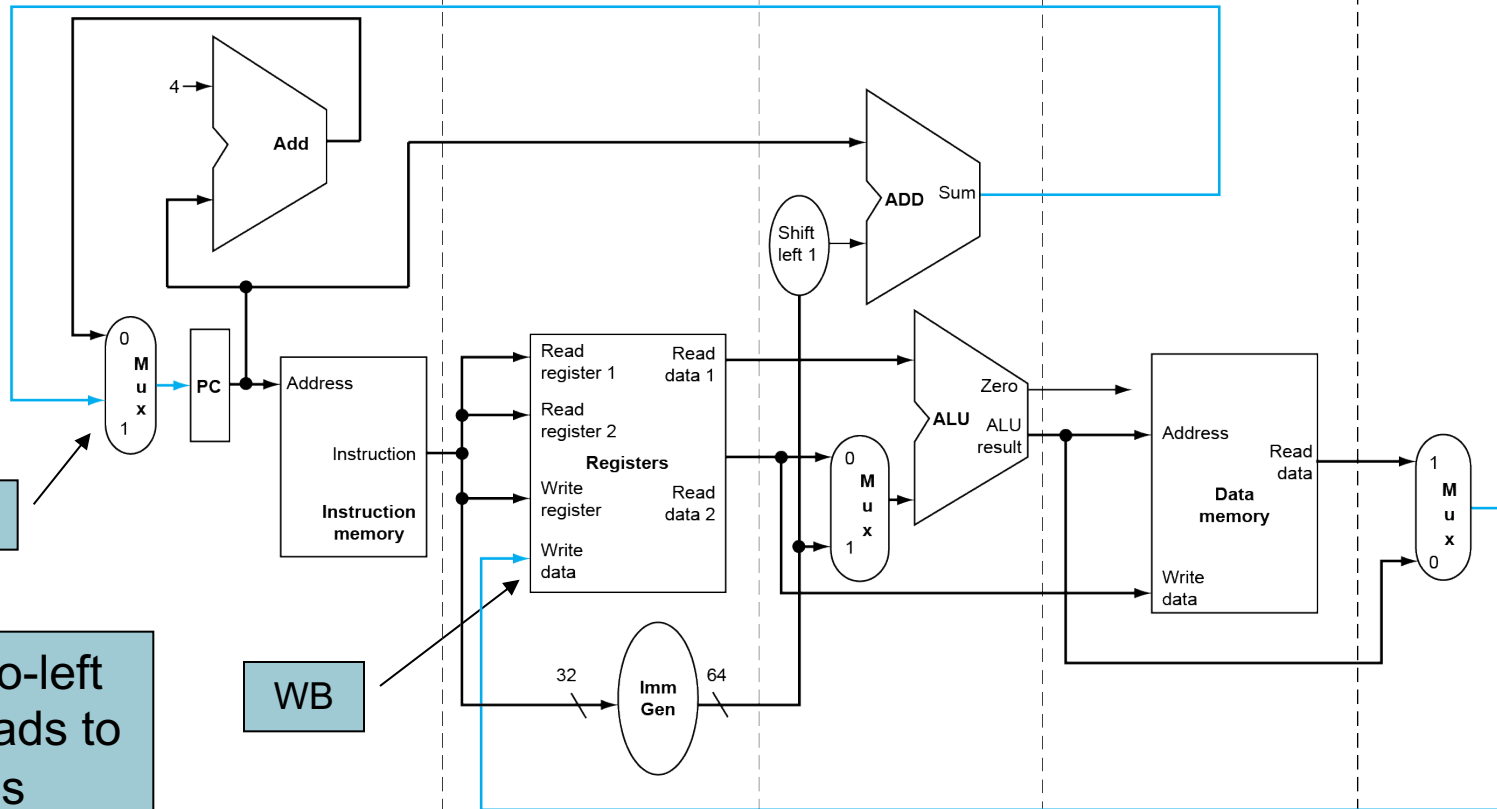
IF: Instruction fetch

ID: Instruction decode/
register file read

EX: Execute/
address calculation

MEM: Memory access

WB: Write back



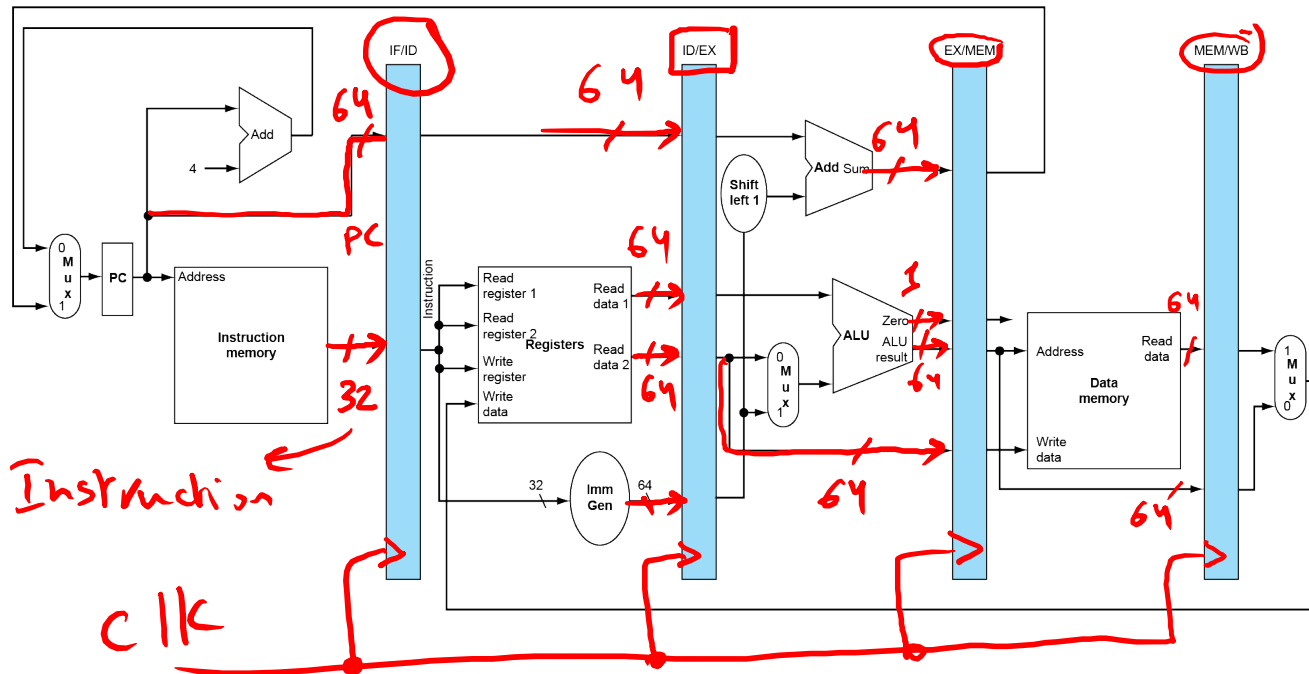
MEM

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

① “Single-clock-cycle” pipeline diagram

- Shows pipeline usage in a single cycle
- Highlight resources used

Do not confuse this with “Single-Cycle CPU”

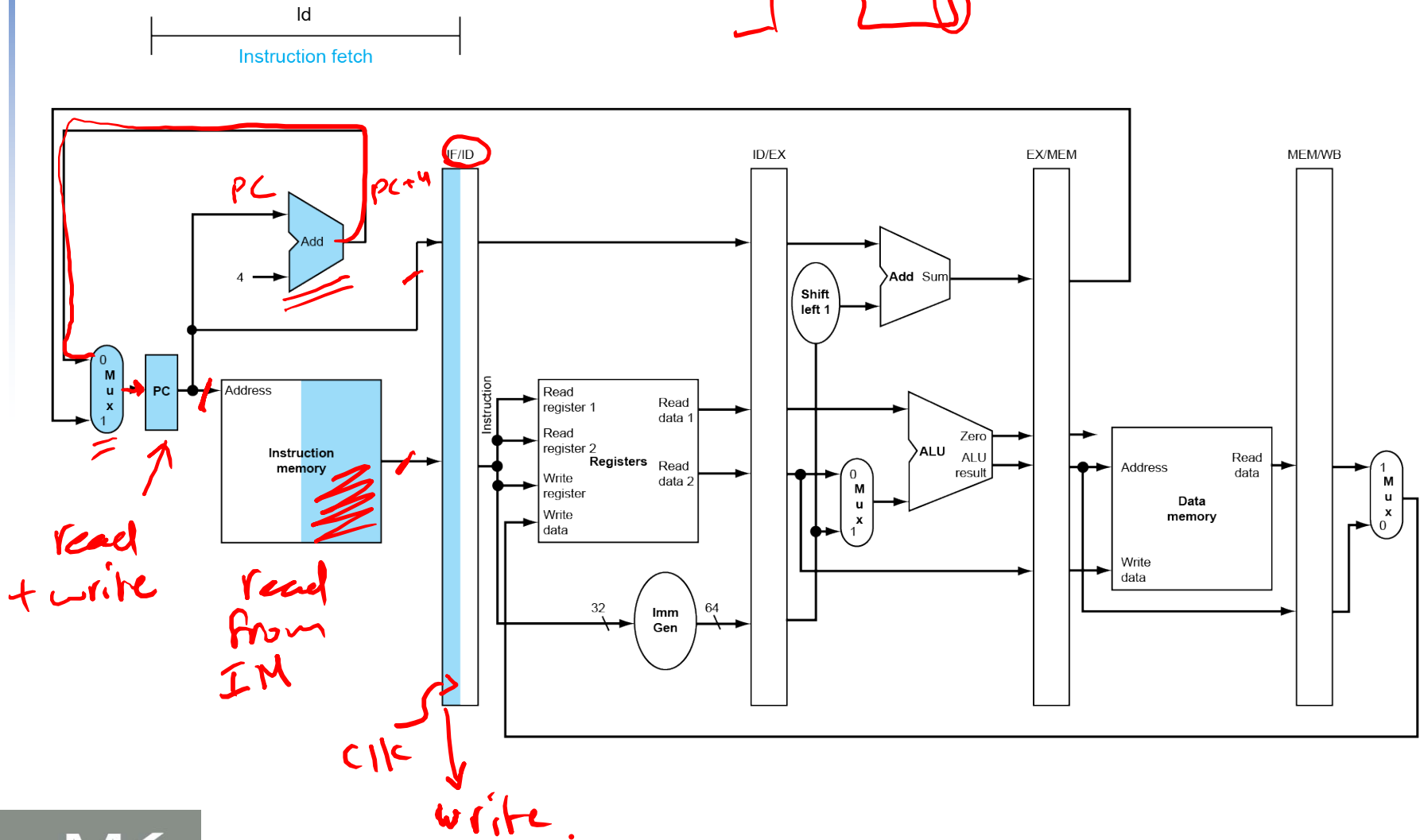
② c.f. “multi-clock-cycle” diagram

- Graph of operation over time

Do not confuse this with “Multi-Cycle CPU”

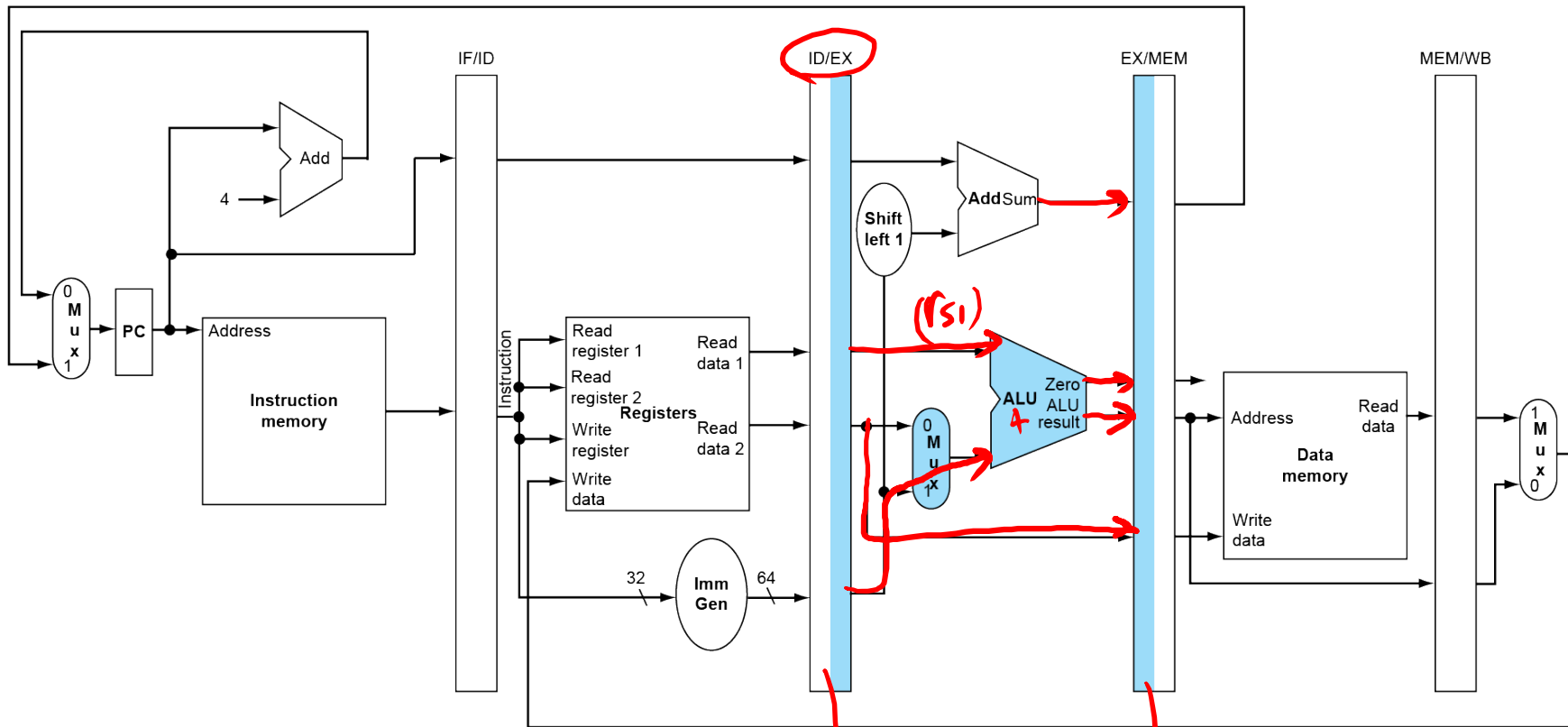
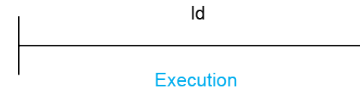
- We’ll look at “single-clock-cycle” diagrams for load & store

IF for Load, Store, ...



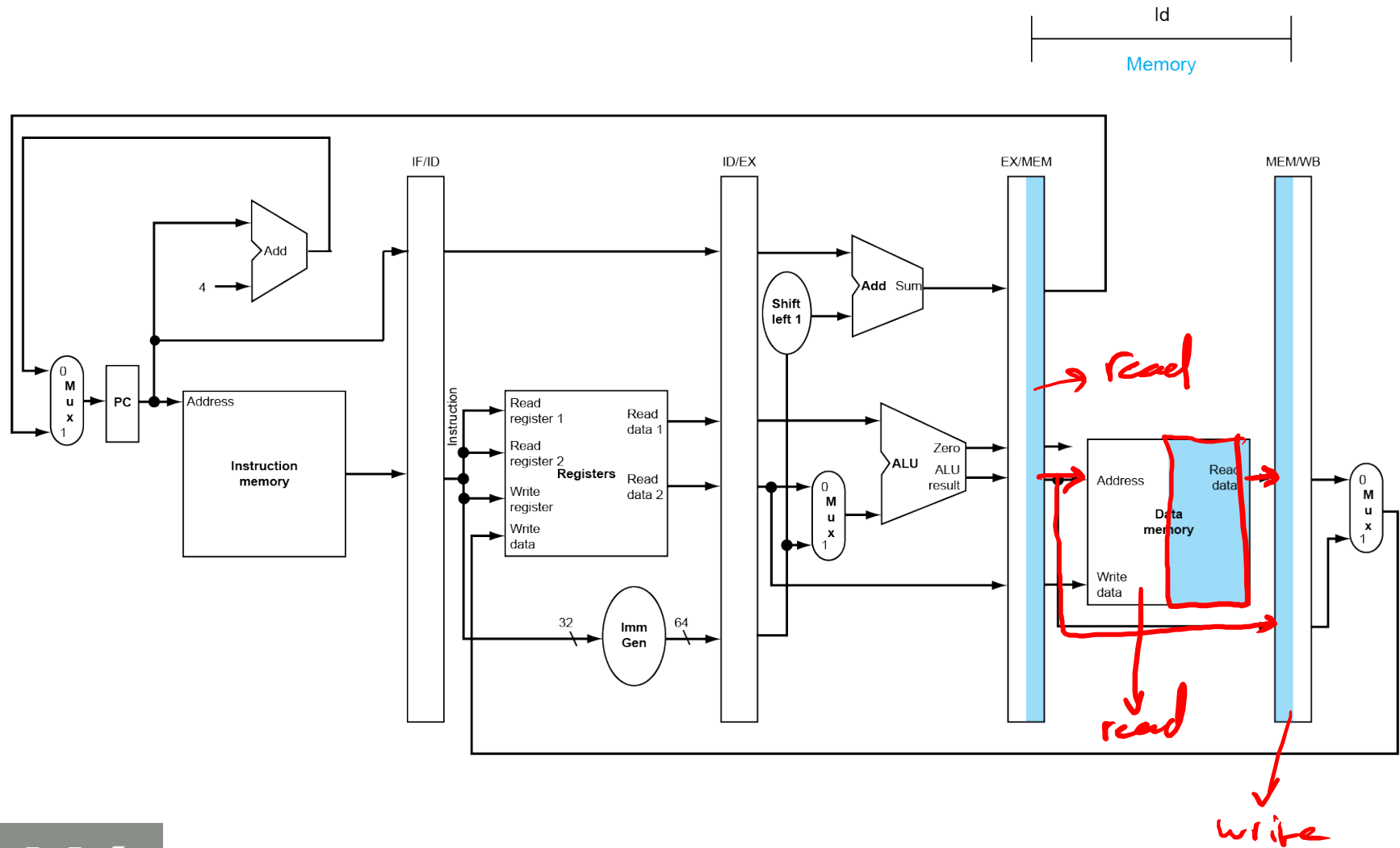
EX for Load

memory address = $(r_{s1}) + \text{Sign-extend}(\text{Imm})$



read write

MEM for Load



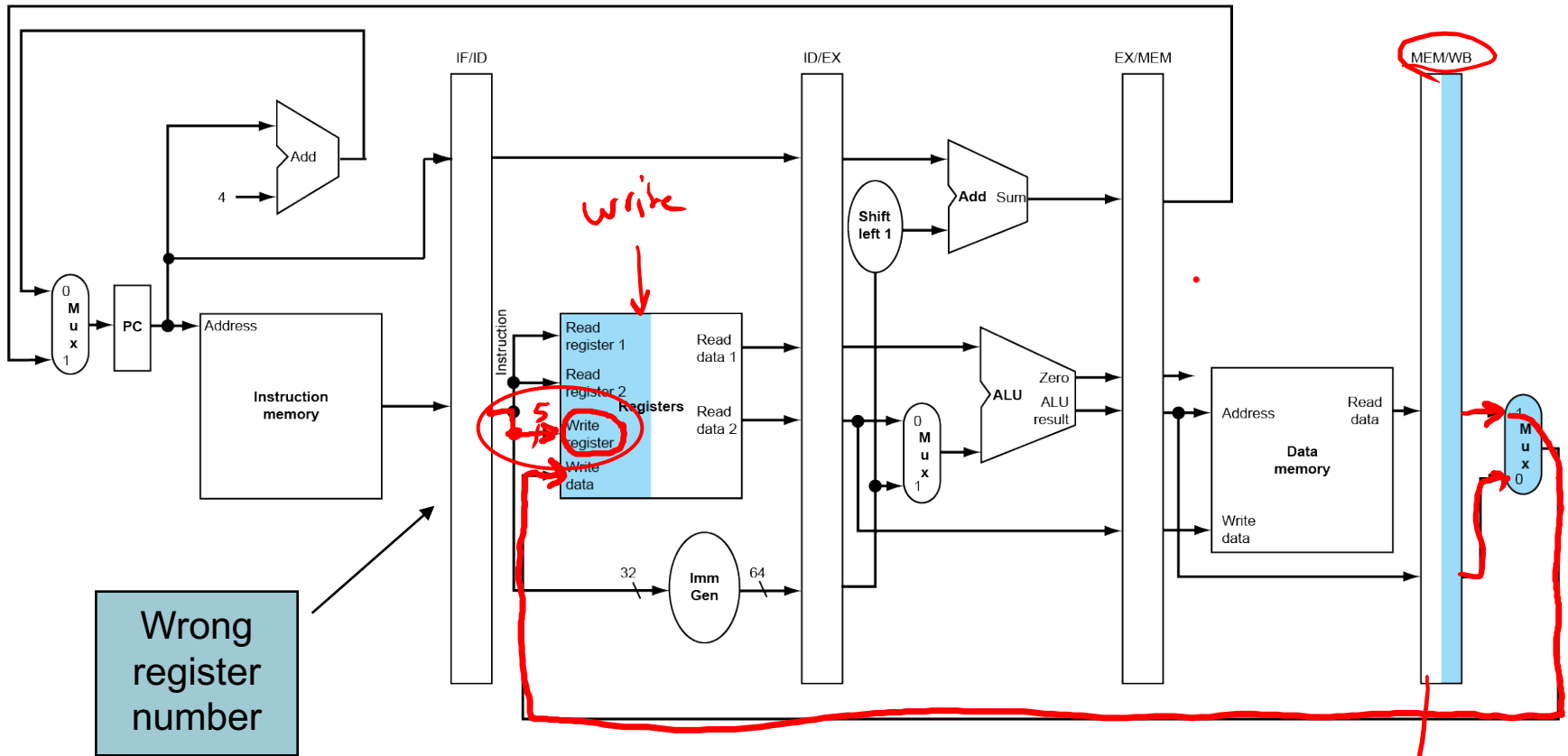
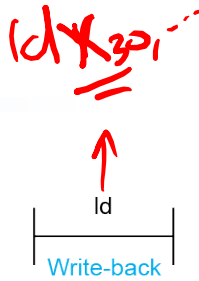
WB for Load *add x10*

instr 4

instr 3

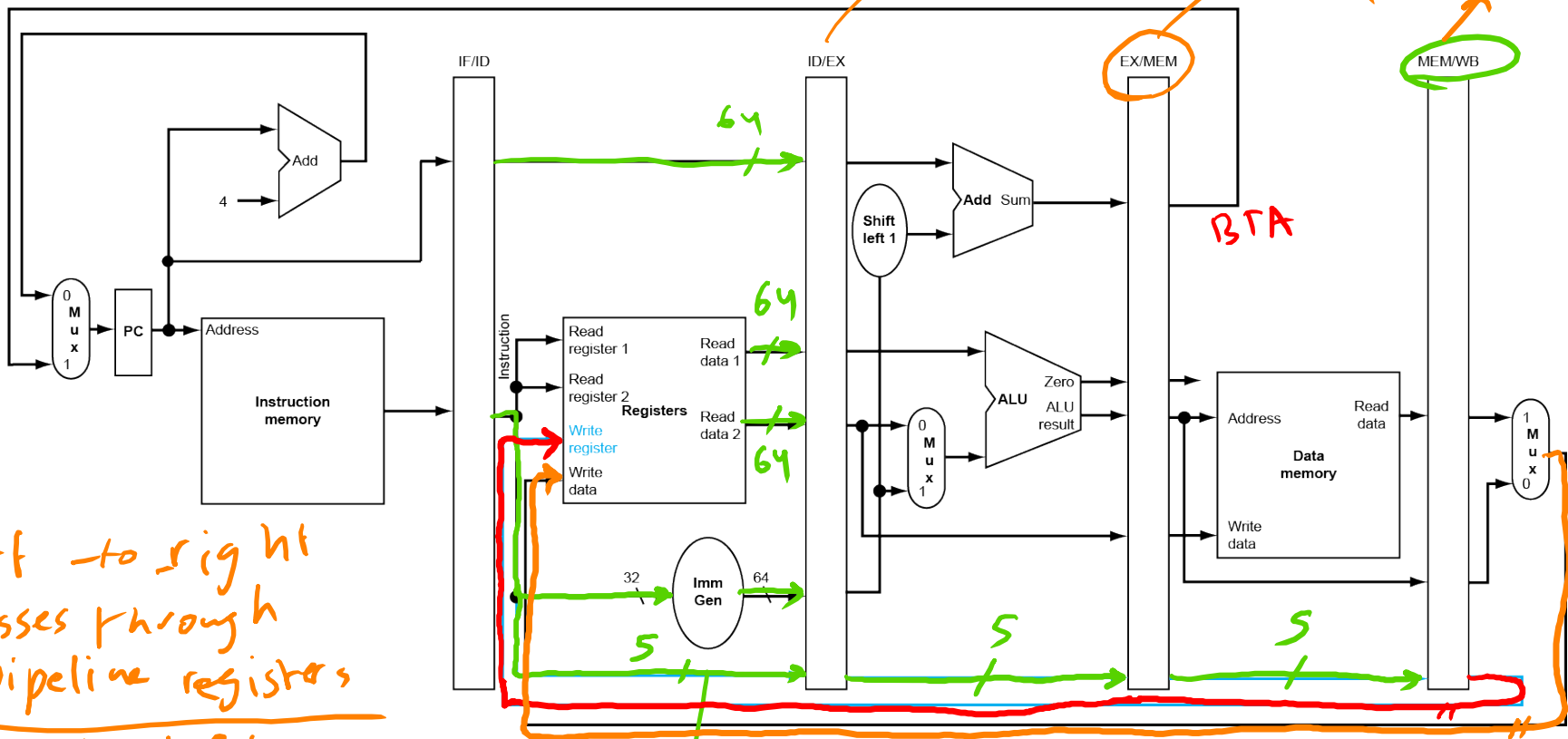
instr 2

instr 1



Wrong register number

Corrected Datapath for Load

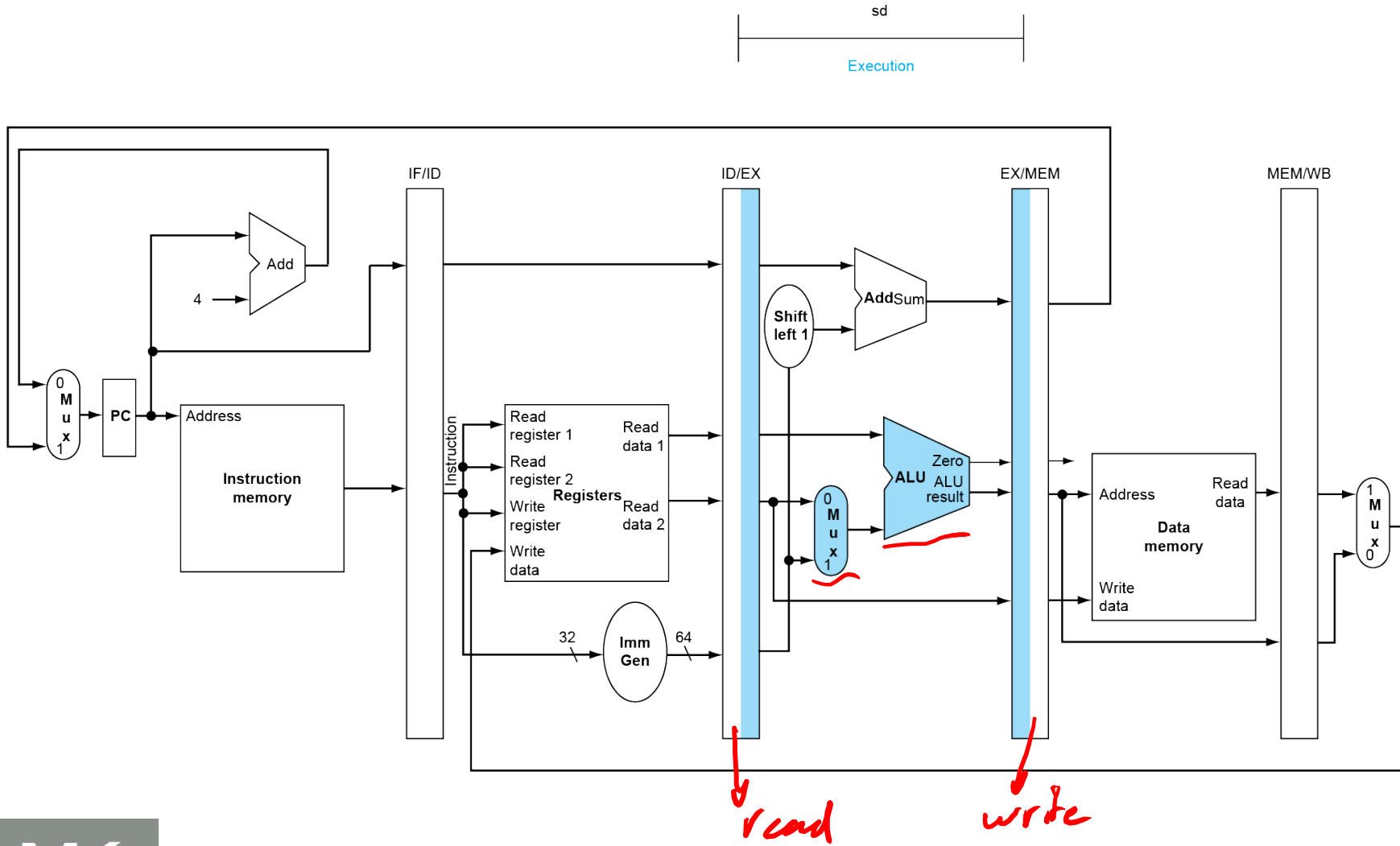


left to right passes through pipeline registers

right to left does NOT pass through pipeline registers

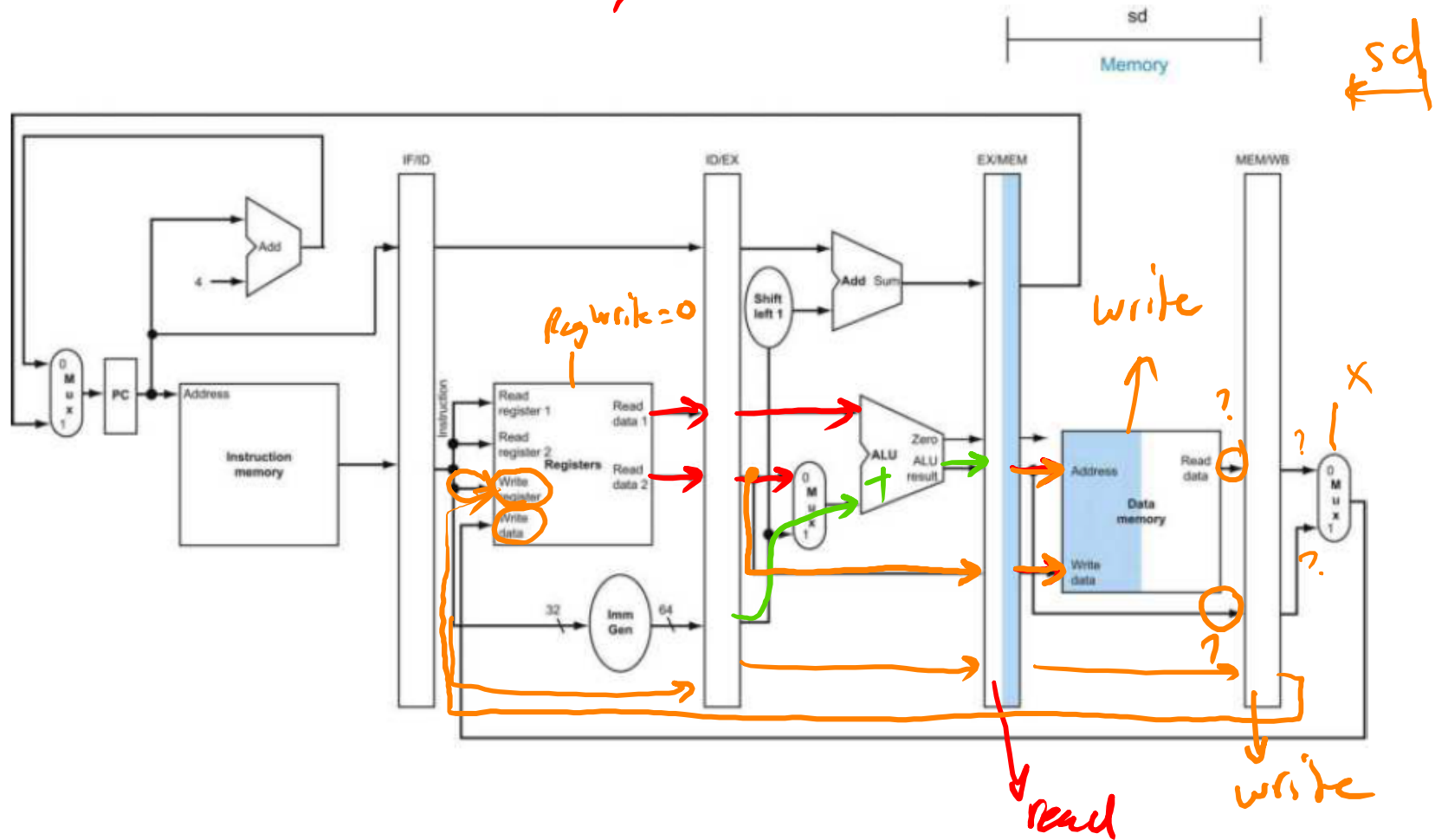


EX for Store

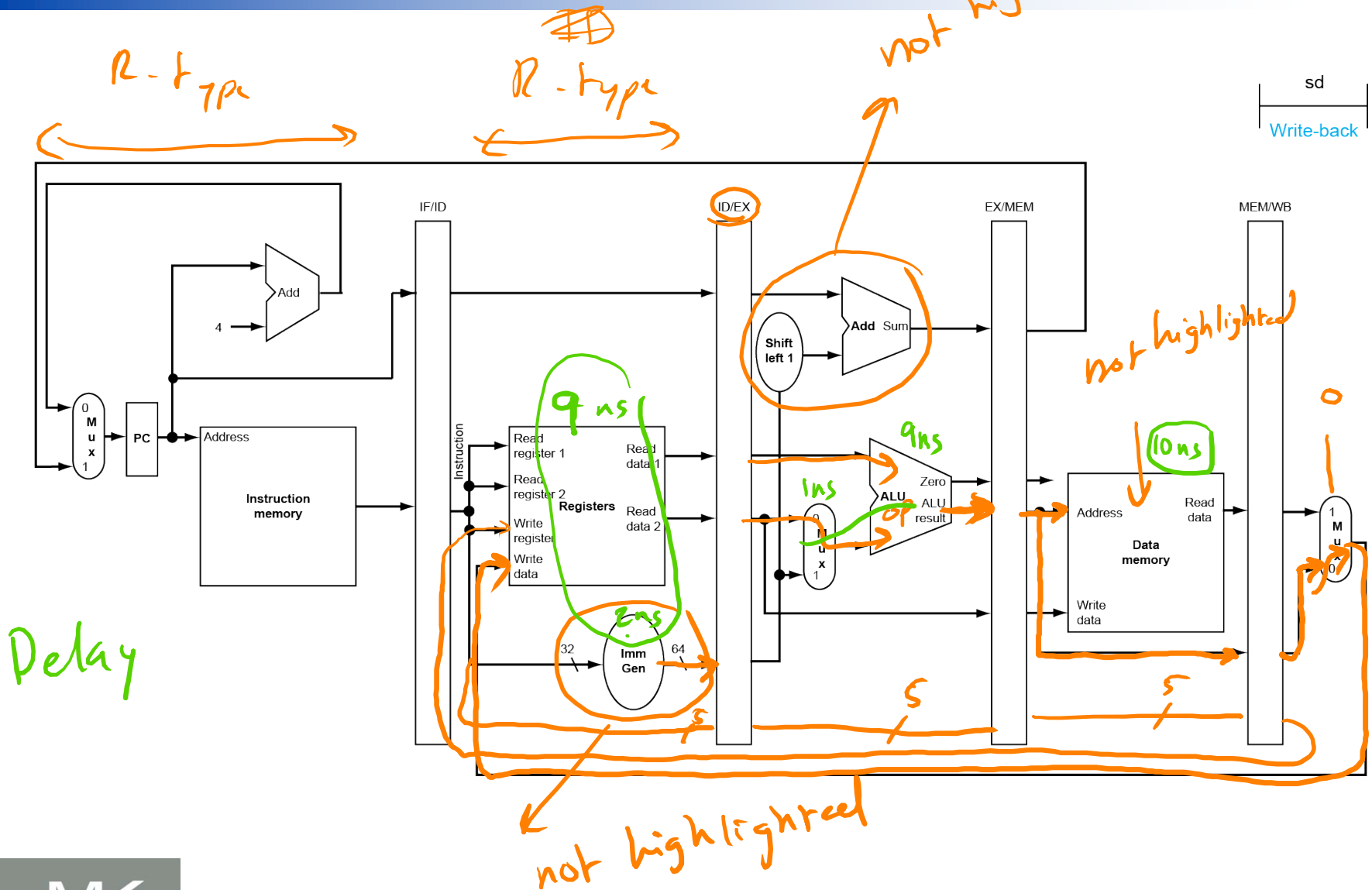


MEM for Store

sd r32, offset(r31)

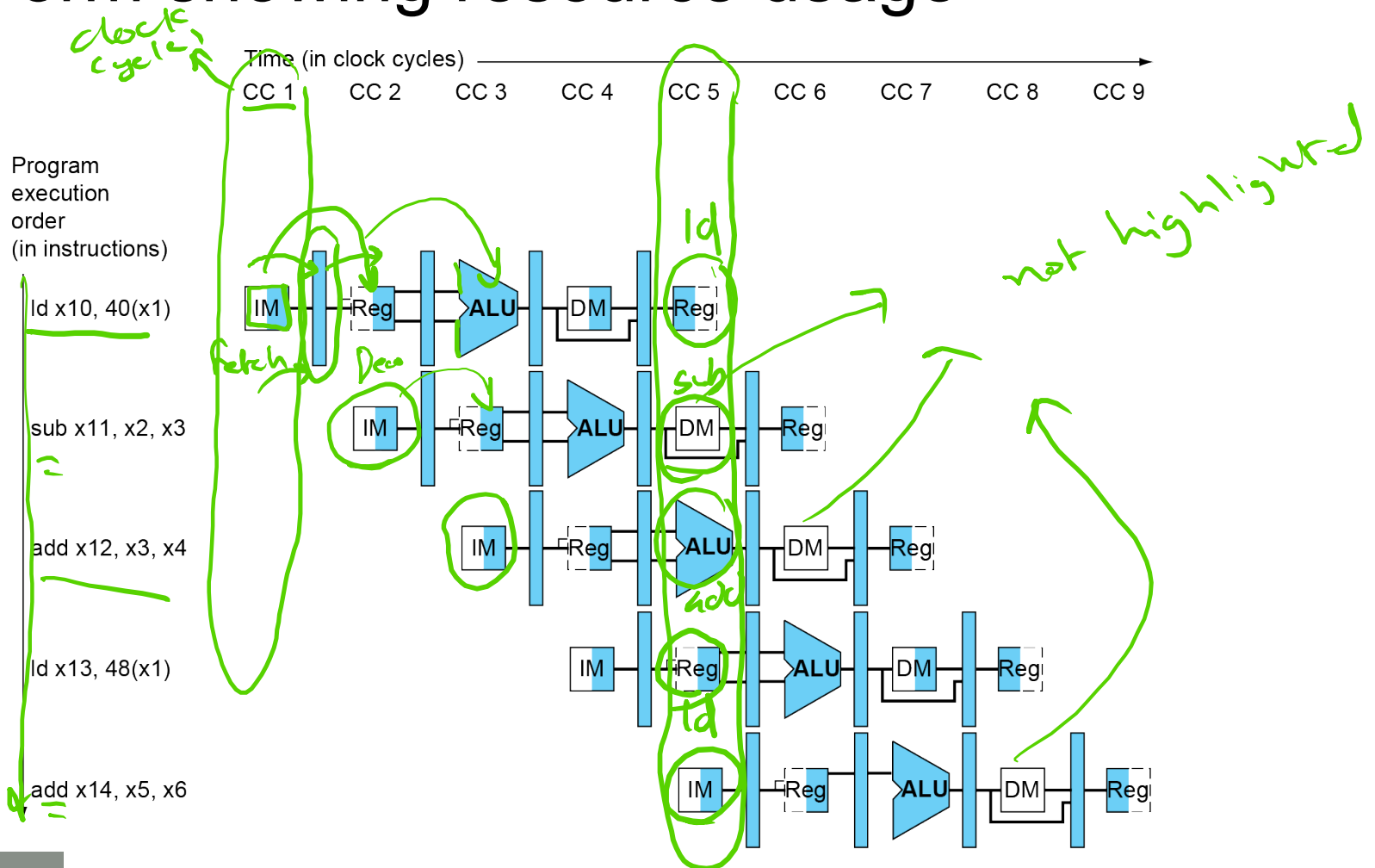


WB for Store



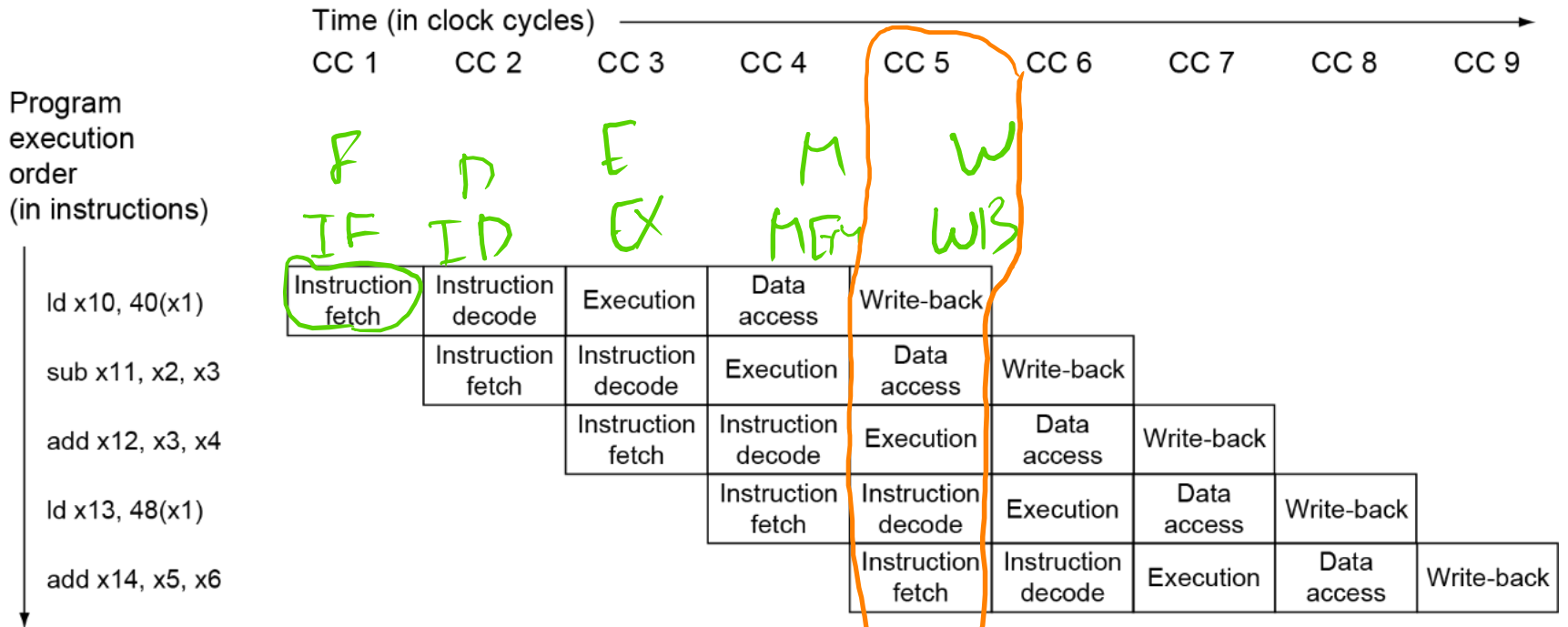
Multi-Cycle Pipeline Diagram

- Form showing resource usage



Multi-Cycle Pipeline Diagram

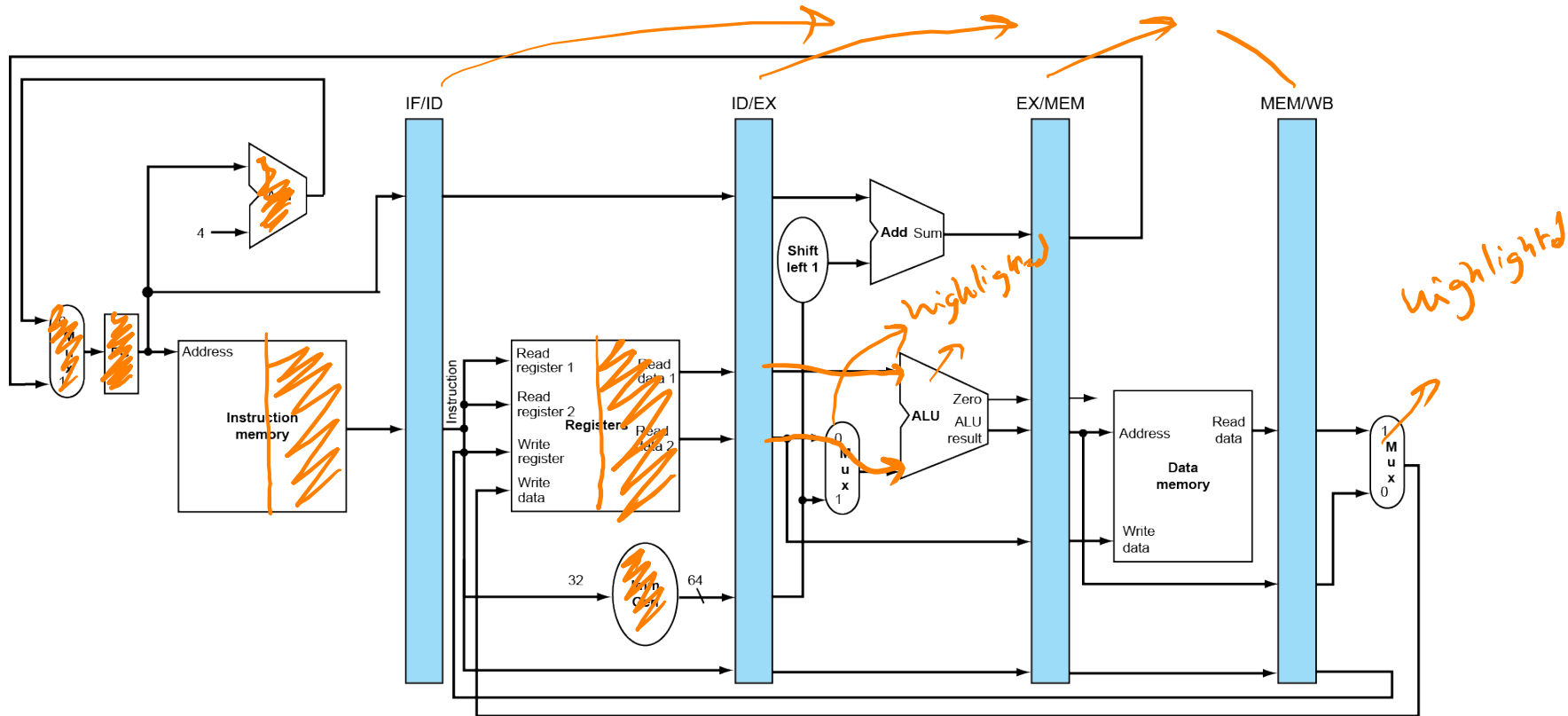
Traditional form



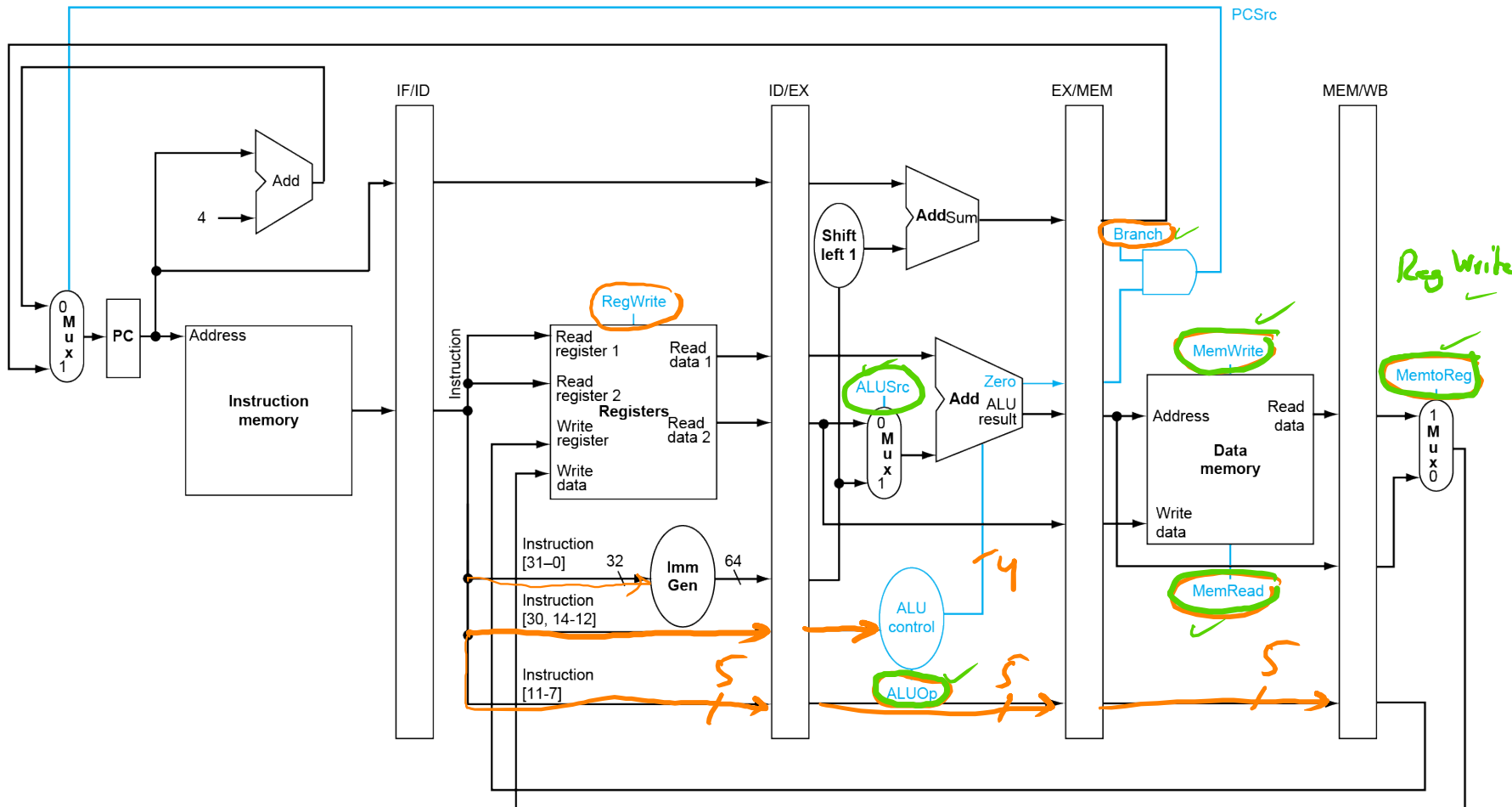
Single-clock-cycle Diagram

Single-Cycle Pipeline Diagram

- State of pipeline in a given cycle (cc5)

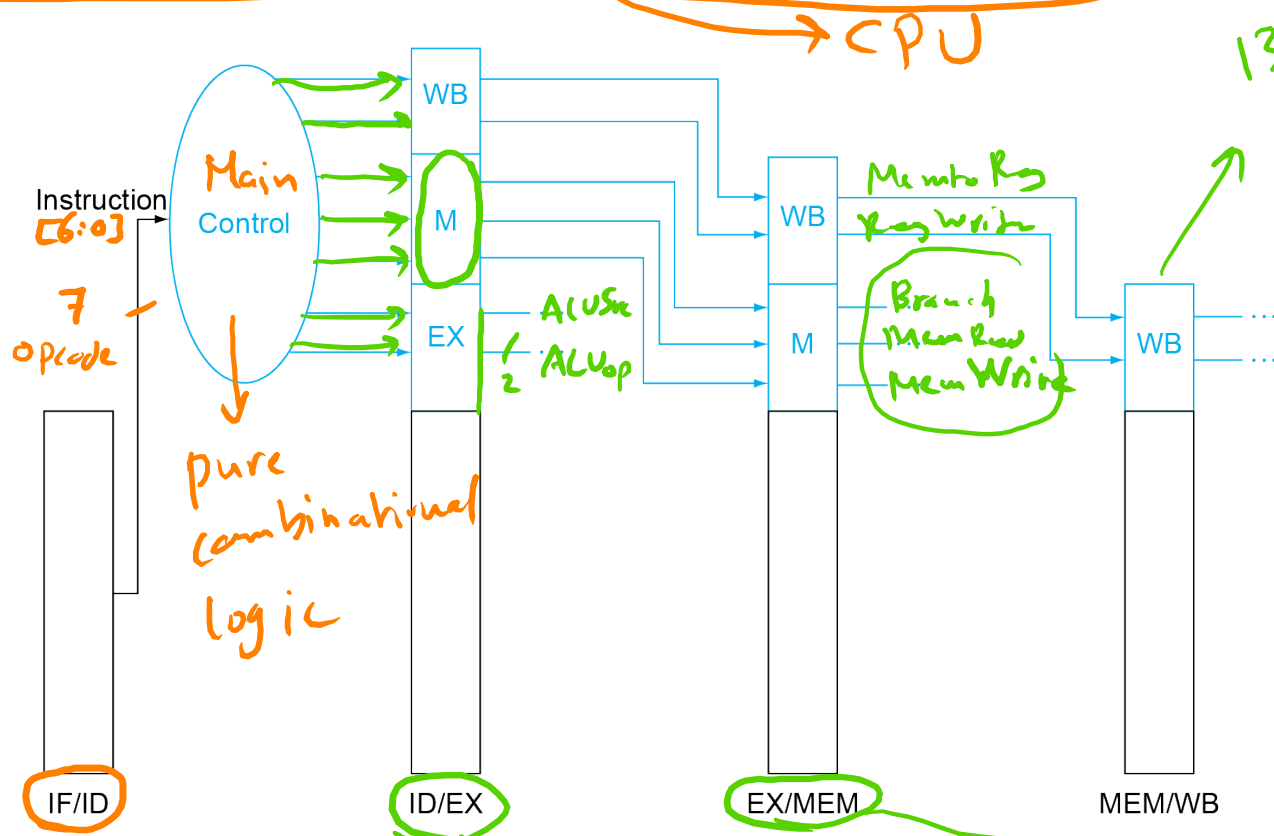


Pipelined Control (Simplified)



Pipelined Control

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

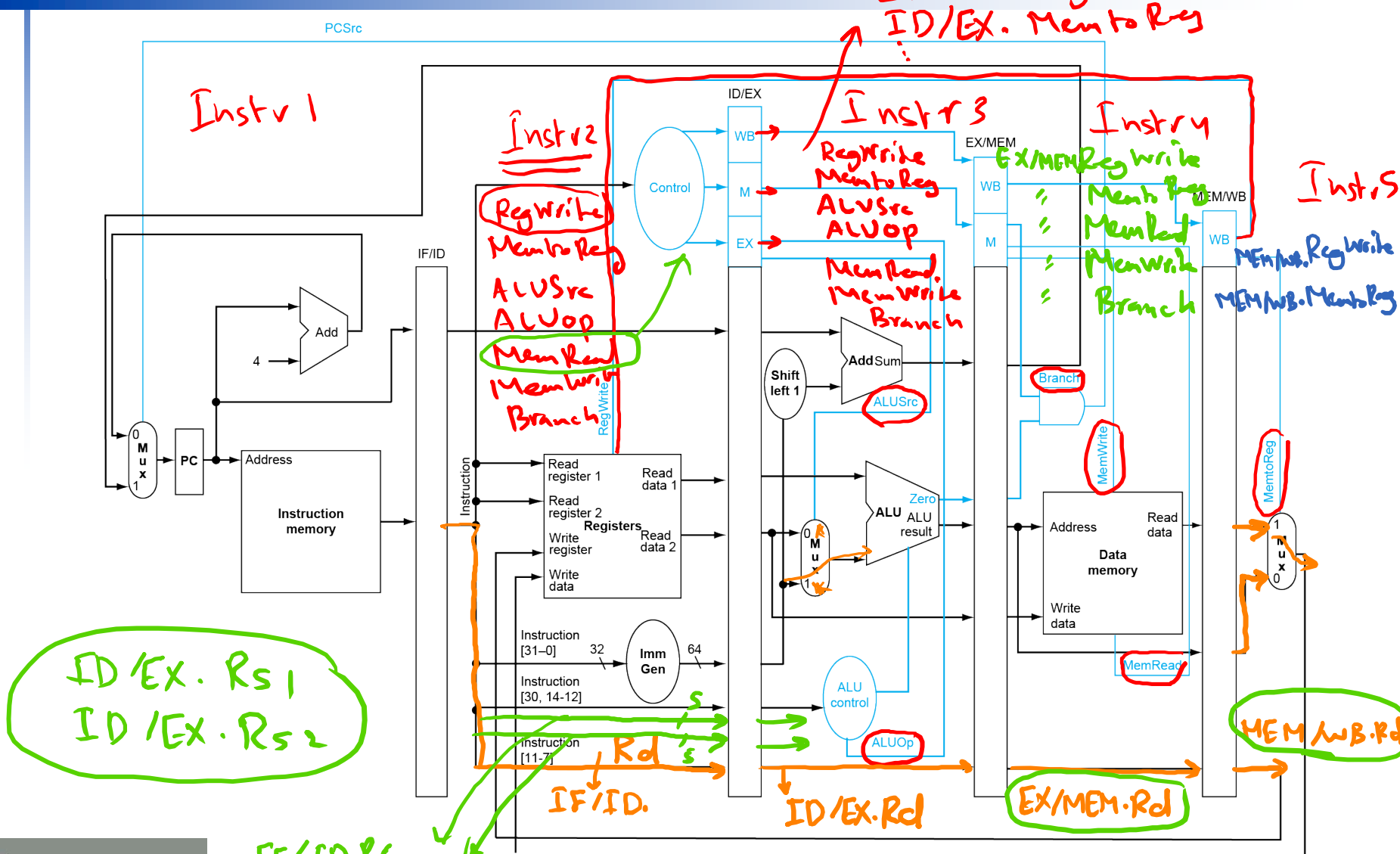


$$261 + 8 = 269 \text{ bit}$$

$$133 + 2 = 135 \text{ bit}$$

$$198 + 5 = 203 \text{ bit}$$

Pipelined Control



ID/EX. Rs1
ID/EX. Rs2

IF/ID. Rs1
IF/ID. Rs2 = Instr [24:20]



	cc1	cc2	cc3	cc4	cc5	cc6	cc7	cc8
* Ld	F	D	E	M	W			
add		F	D	E	M	W		
<u>sd</u> *			F	<u>D</u>	E	M	W	
sub				F	D	E	M	W

In cc5,
MEM/WB.MemRead = 1

In cc4 what is the value of EX/MEM.MemRead?

answer = 1

In cc4 what is the value of ID/EX.MemRead?

answer = 0

In cc4 what is the value of MemRead?

answer = 0

In cc5, ID/EX.ALUSrc = 1
In cc5, EX/MEM.RegWrite = 1

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

Value of register x2:	CC 1	CC 2	CC 3	CC 4	CC 5	CC 6	CC 7	CC 8	CC 9
	10	10	10	10	10/-20	-20	-20	-20	-20

Program execution order (in instructions)

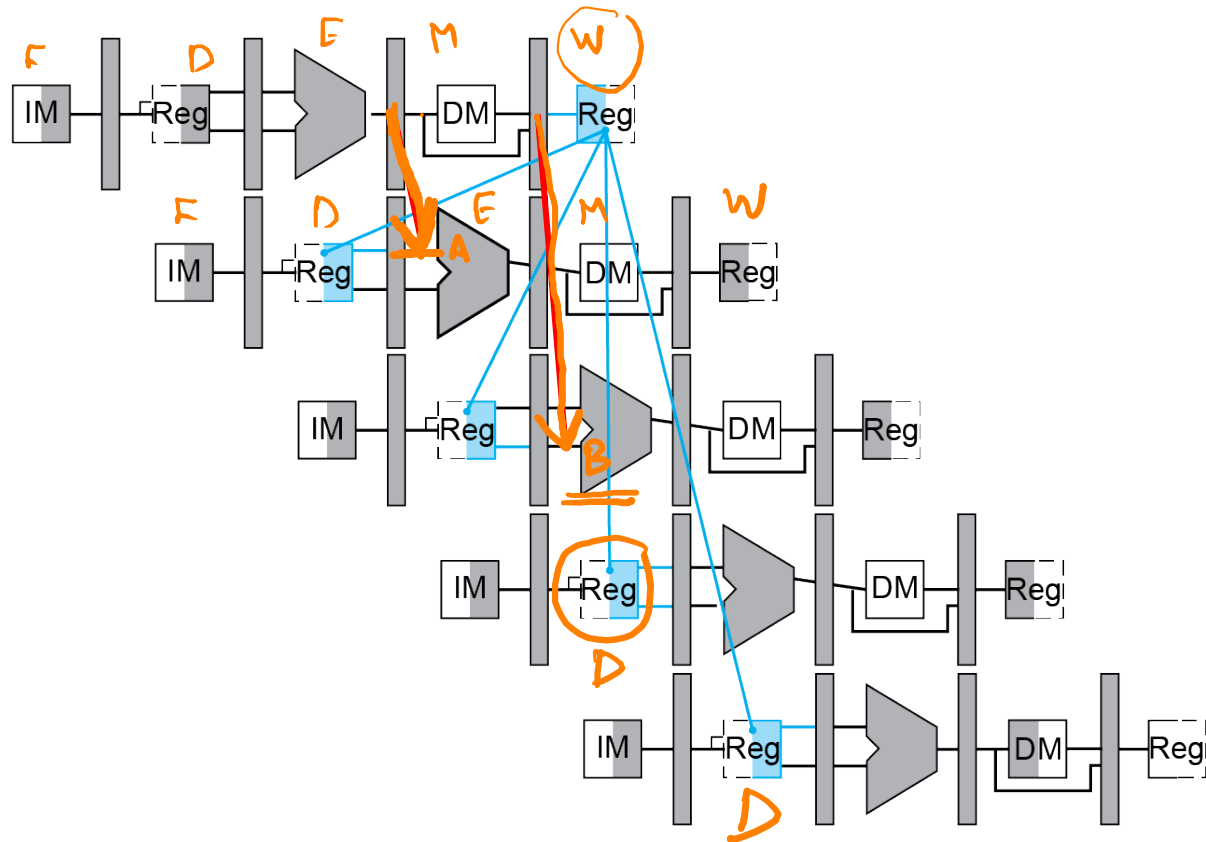
sub x2, x1, x3

and x12, x2, x5

or x13, x6, x2

add x14, x2, x2

sd x15, 100(x2)



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

the destination register number of the instruction in Memory stage

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

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

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

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

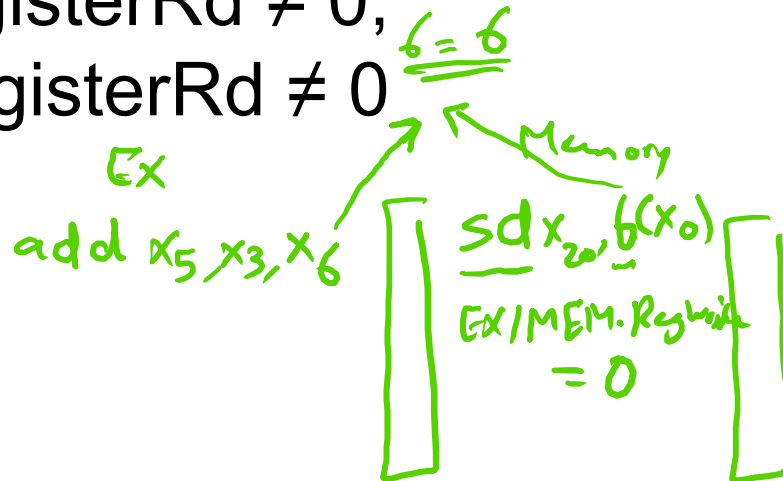
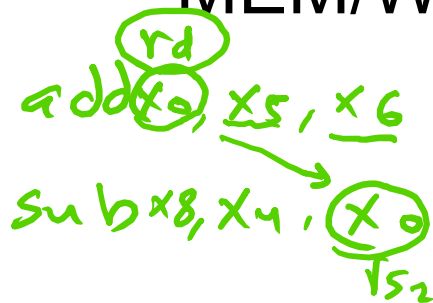
Fwd from EX/MEM pipeline reg

Fwd from MEM/WB pipeline reg

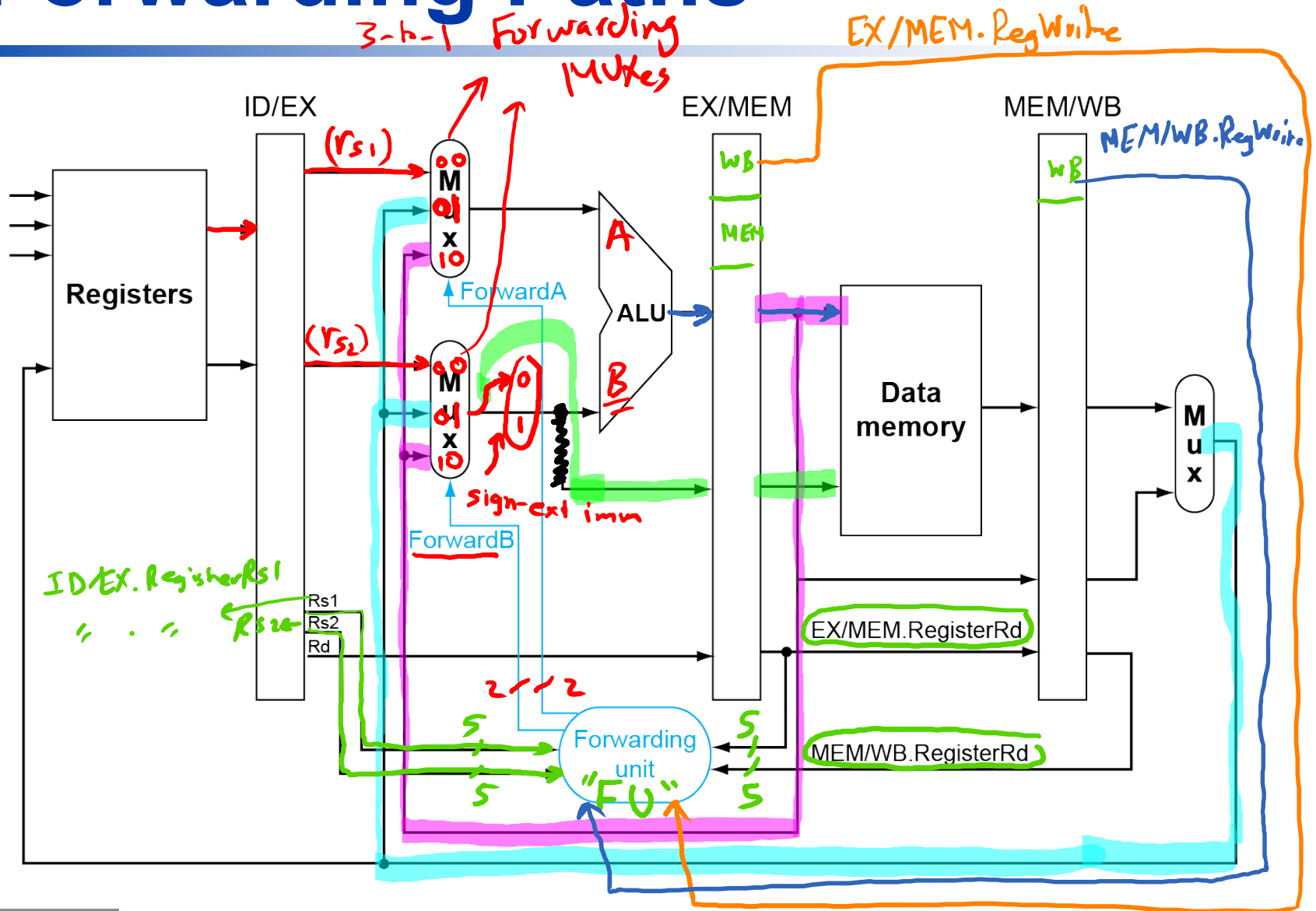
Detecting the Need to Forward

- 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 $\neq 0$,
MEM/WB.RegisterRd $\neq 0$

Nop \equiv no operation \equiv add x0, x5, x6



Forwarding Paths



Forwarding Conditions

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

Forwarding Conditions

EX hazard

EX/MEM to

or
rs1
rs2

Mem < rs1
rs2

1. if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0 and (EX/MEM.RegisterRd = ID/EX.RegisterRs1)))

ForwardA = 10 ✓

2. if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0 and (EX/MEM.RegisterRd = ID/EX.RegisterRs2)))

ForwardB = 10 ✓

MEM hazard

MEM/WB to or rs1
rs2

WB < rs1
rs2

3. if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0 and (MEM/WB.RegisterRd = ID/EX.RegisterRs1)))

ForwardA = 01

4. if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0 and (MEM/WB.RegisterRd = ID/EX.RegisterRs2)))

ForwardB = 01

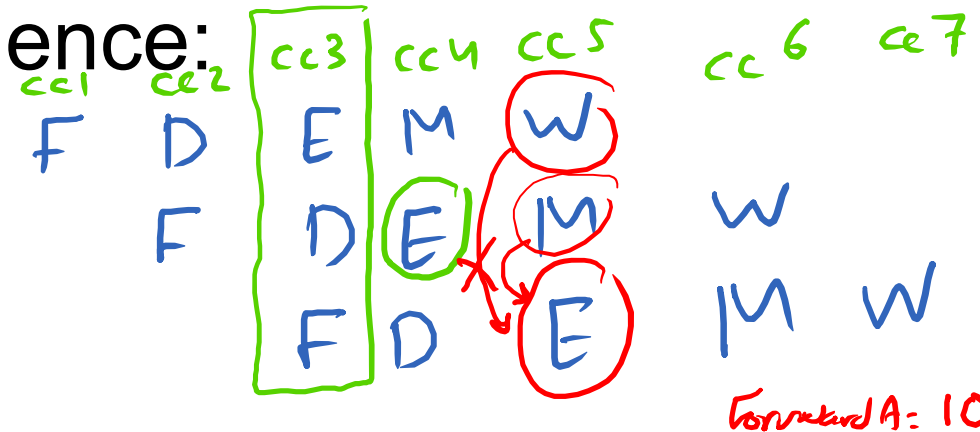
① & ③ are False ⇒ ForwardA = 00
② & ④ " " ⇒ ForwardB = 00

Double Data Hazard

- Consider the sequence:

```

1 add x1, x1, x2
2 add x1, x1, x3
3 add x1, x1, x4
    
```



- Both hazards occur

- Want to use the most recent

In cc5
 Forward A = 10
 Forward B = 00

- Revise MEM hazard condition

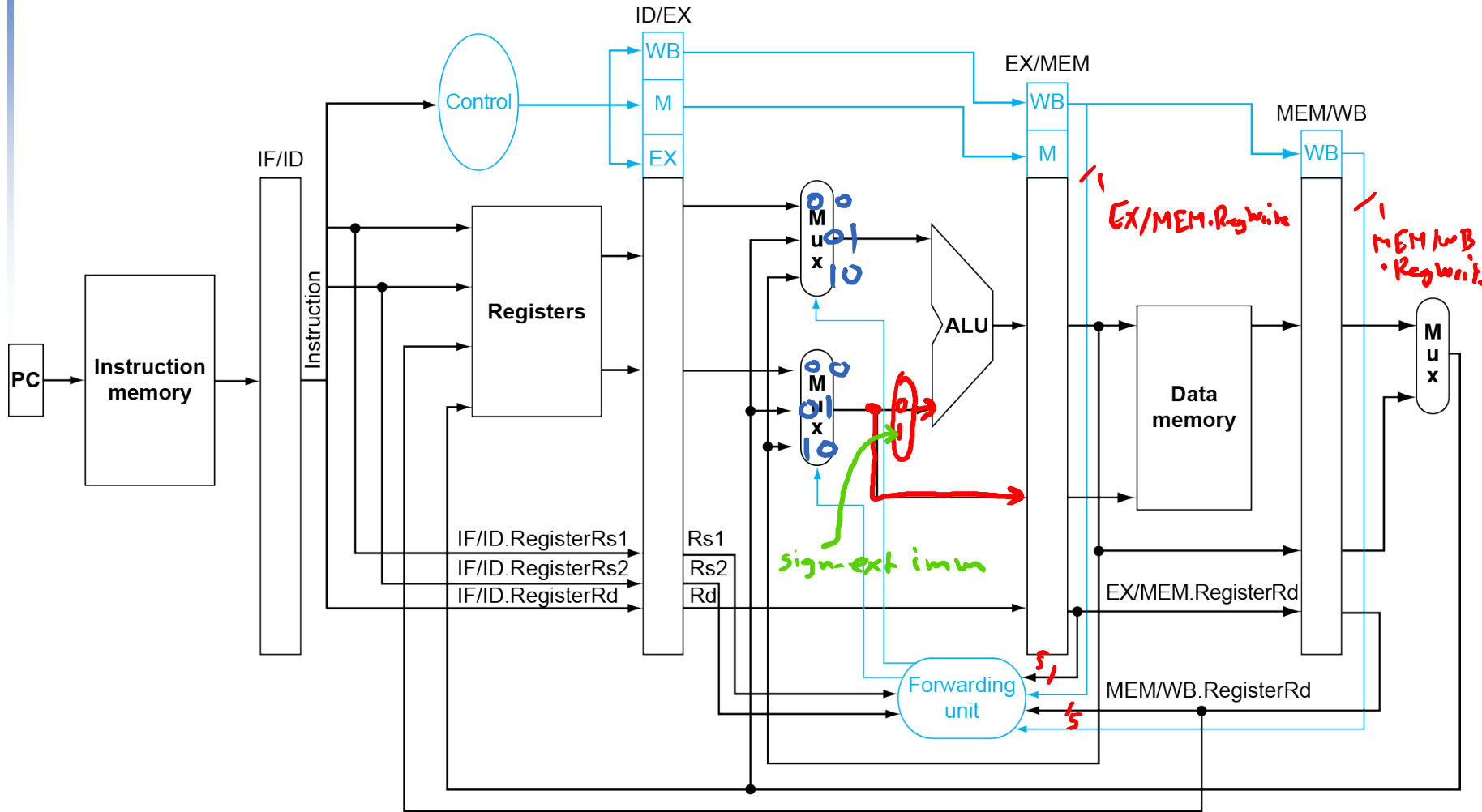
- Only fwd if EX hazard condition isn't true

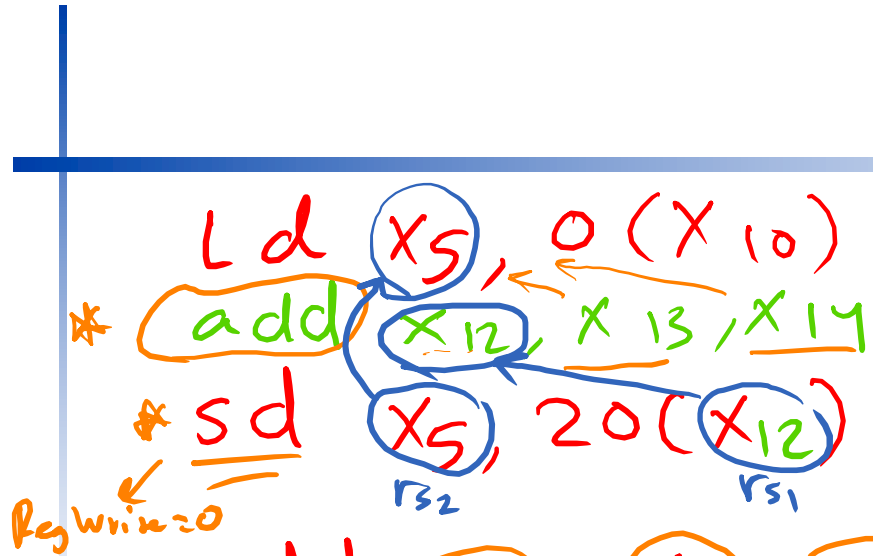
in cc3 < Forward A = 00
 Forward B = 00
 in cc4 < Forward A = 10
 Forward B = 00

Revised Forwarding Condition

- MEM hazard MEM / WB ← ~~RS1~~ ~~RS2~~ RS2
- 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



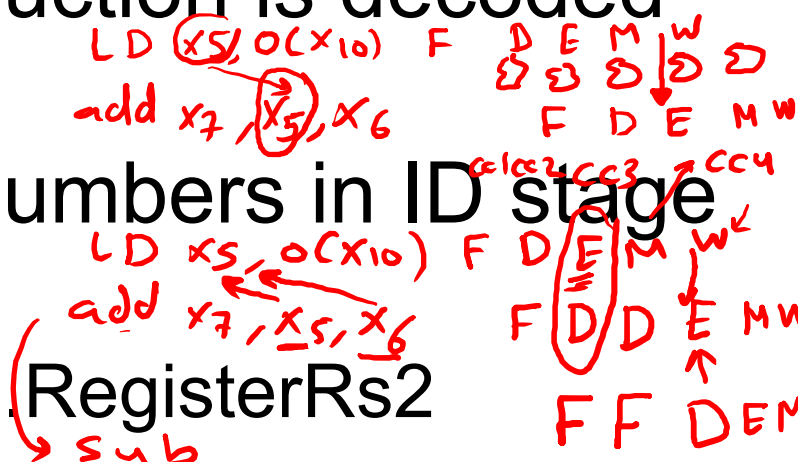


cc1	cc2	cc3	cc4	cc5	cc6	cc7	cc8	cc9
F	D	E	M	W				
	F	D	B	M	W			
		F	D	F	M	W		
			F	D	E	M	W	
				F	D	E	M	W

- in cc3: No forwarding
- in cc4: " "
- in cc5: Fwd - from - WB - to - r_{32} , Fwd - from - Mem - to - r_{31}
- in cc6: No forwarding
- in cc7: Fwd - from Mem to r_{31}

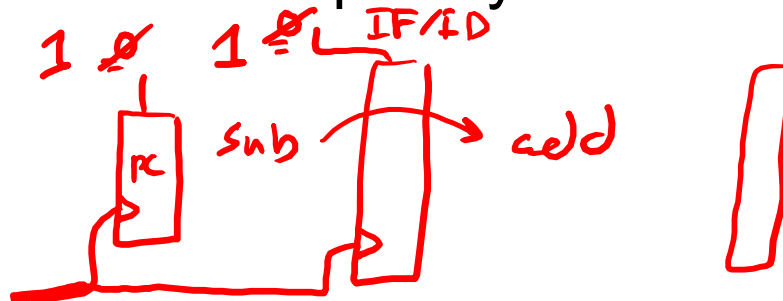
Load-Use Hazard Detection

- 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
 - ① ID/EX.MemRead⁼¹ and (ID/EX.RegisterRd ≠ 0) and ③ (ID/EX.RegisterRd = IF/ID.RegisterRs1) or (ID/EX.RegisterRd = IF/ID.RegisterRs2)
- If detected, stall and insert bubble

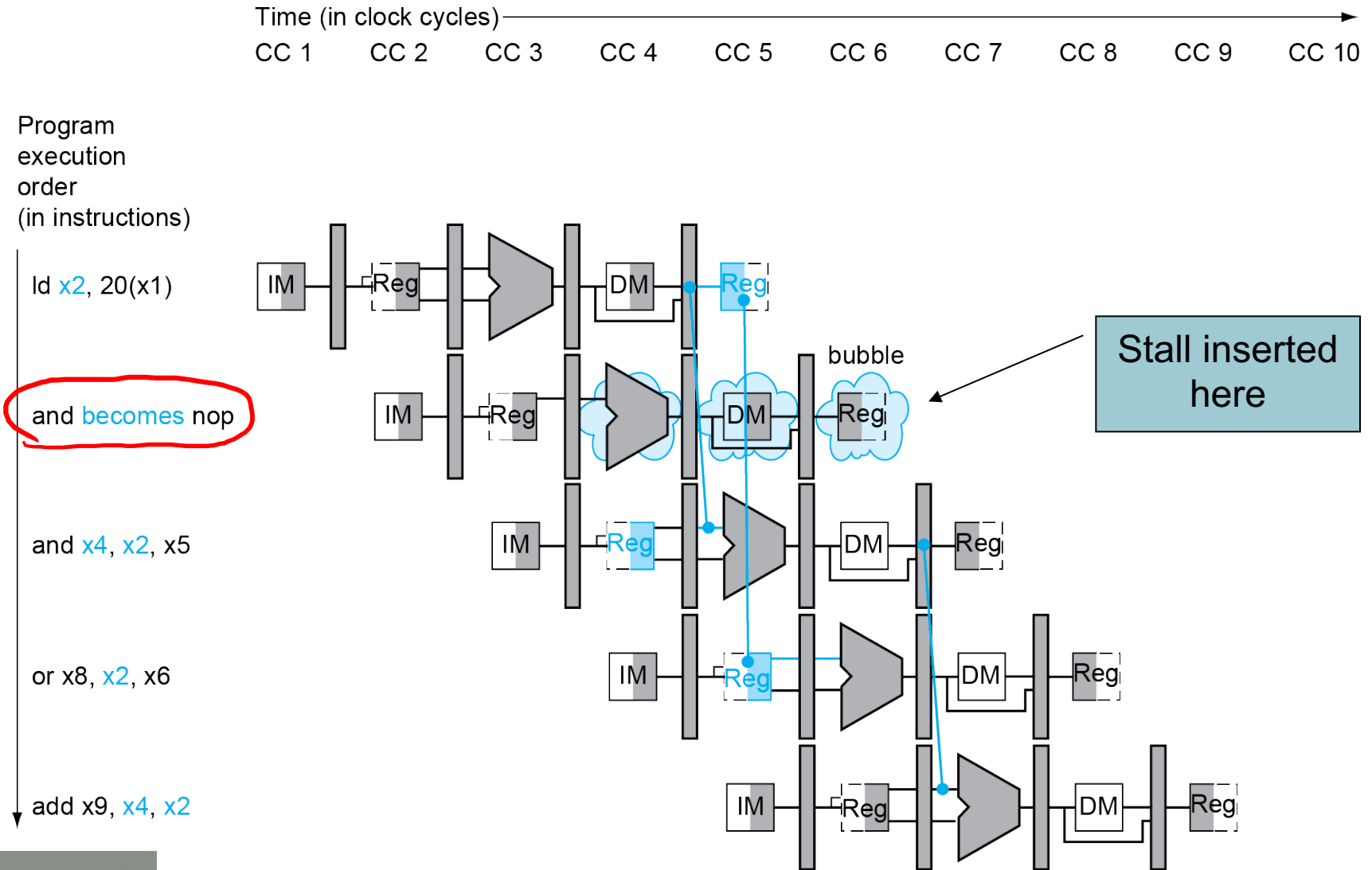


How to Stall the Pipeline

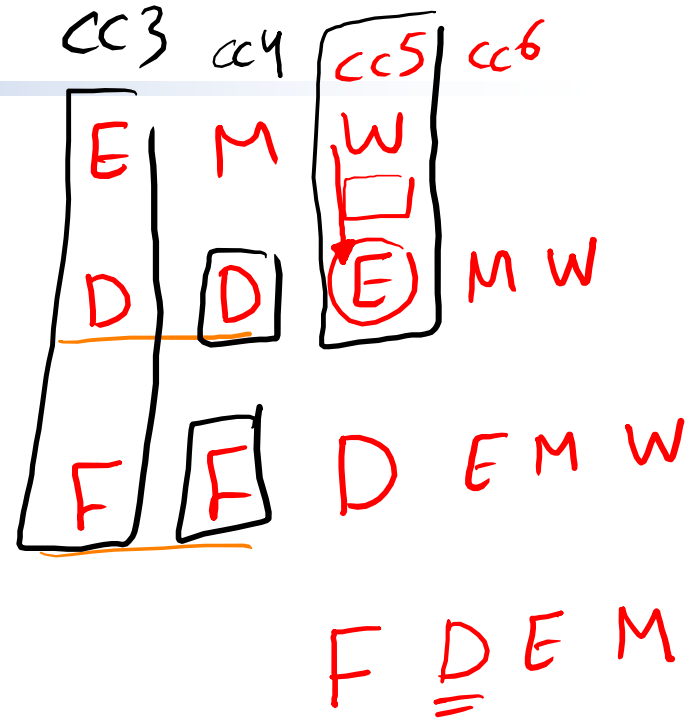
- Force control values in ID/EX register to 0
 - EX, MEM and WB do nop (no-operation)
- Prevent update of PC and IF/ID register
 - Using instruction is decoded again
 - Following instruction is fetched again
 - 1-cycle stall allows MEM to read data for 1d
 - Can subsequently forward to EX stage



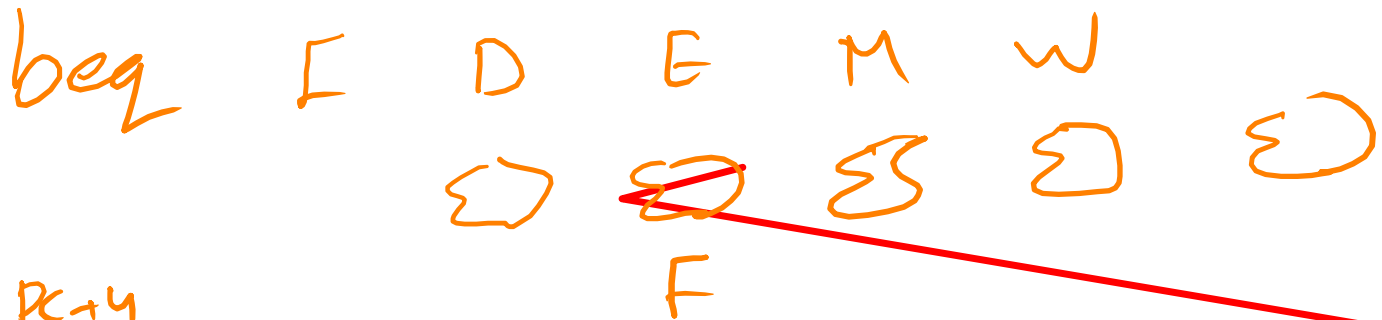
Load-Use Data Hazard



ld x2, z0(x1) F D
 and x4, (x2), x5 F
 or x8, (x2), x6
 add x9, x4, (x2)



stall on branch



PC+4
or BTA

In CC5 \Rightarrow EX/MEM. MemWrite = 0

In CC5 \Rightarrow Forward A = 01
Forward B = 00

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]

RISC-V Number Representations

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

0000 0000 0000 0000 0000 0000 0000 0000	$_{two}$	=	0_{ten}
0000 0000 0000 0000 0000 0000 0000 0001	$_{two}$	=	$+ 1_{ten}$
...			
0111 1111 1111 1111 1111 1111 1111 1110	$_{two}$	=	$+ 9,223,372,036,854,775,806_{ten}$
0111 1111 1111 1111 1111 1111 1111 1111	$_{two}$	=	$+ 9,223,372,036,854,775,807_{ten}$
1000 0000 0000 0000 0000 0000 0000 0000	$_{two}$	=	$- 9,223,372,036,854,775,808_{ten}$
1000 0000 0000 0000 0000 0000 0000 0001	$_{two}$	=	$- 9,223,372,036,854,775,807_{ten}$
...			
1111 1111 1111 1111 1111 1111 1111 1110	$_{two}$	=	$- 2_{ten}$
1111 1111 1111 1111 1111 1111 1111 1111	$_{two}$	=	$- 1_{ten}$

maxint (points to 9,223,372,036,854,775,807)

minint (points to -9,223,372,036,854,775,807)

MSB (points to the leftmost bit)

LSB (points to the rightmost bit)

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)

↓
 sign extension ↓
 zero extension

RISC-V Arithmetic Logic Unit (ALU)

- Must support the Arithmetic/Logic operations of the ISA

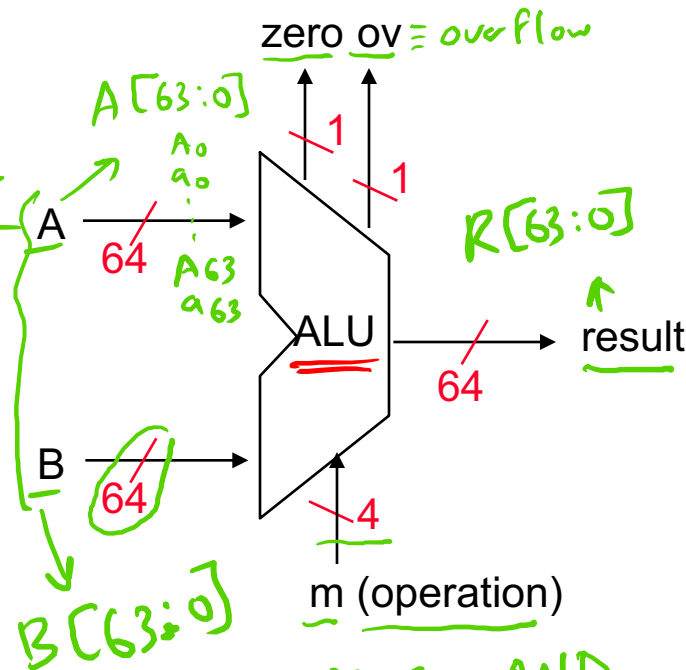
add, addi

sub

and, andi, or, ori, xor, xori

beq, bne, slt, slti, sltiu, sltu

RF
RF, immediate



- 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

0000 AND
0001 OR
:
:
"64-bit ALU"

Review: 2's Complement Binary Representation

□ Negate

$$-2^3 =$$

$$-(2^3 - 1) =$$

2'sc binary	decimal
1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1
<u>0000</u>	0
0001	+ 1
0010	+ 2
0011	+ 3
0100	+ 4
<u>0101</u>	+ 5
0110	+ 6
<u>0111</u>	+ 7

1011

and add a 1

1010

complement all the bits

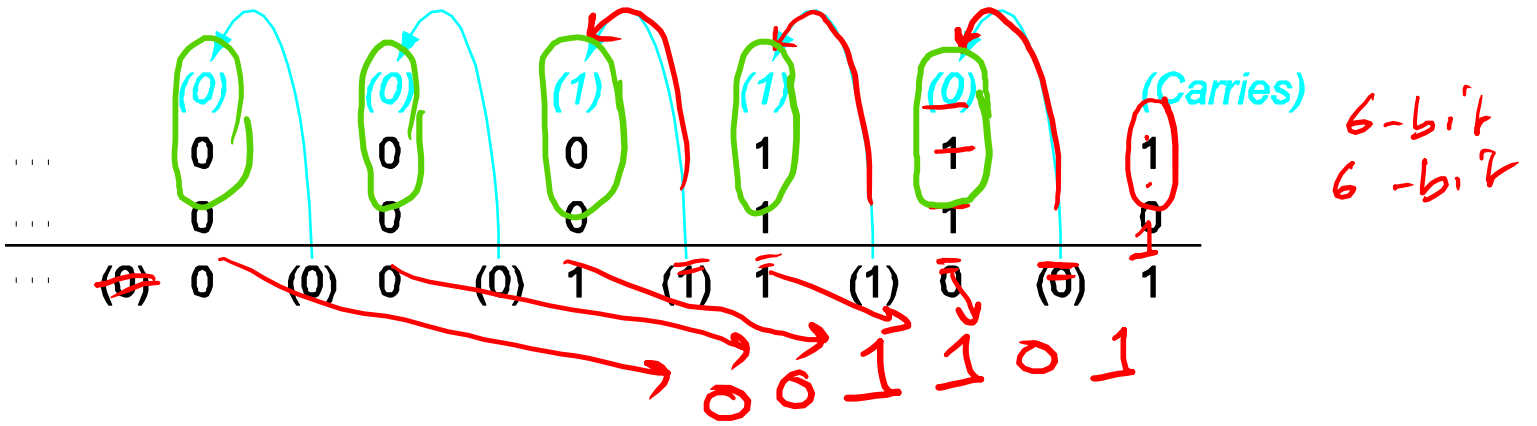
N: \bar{N} invert
 $-N$ negation

□ Note: negate and invert are different!

$$2^3 - 1 =$$

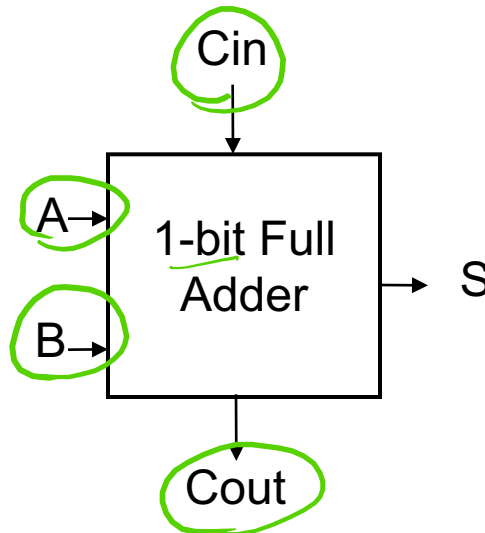
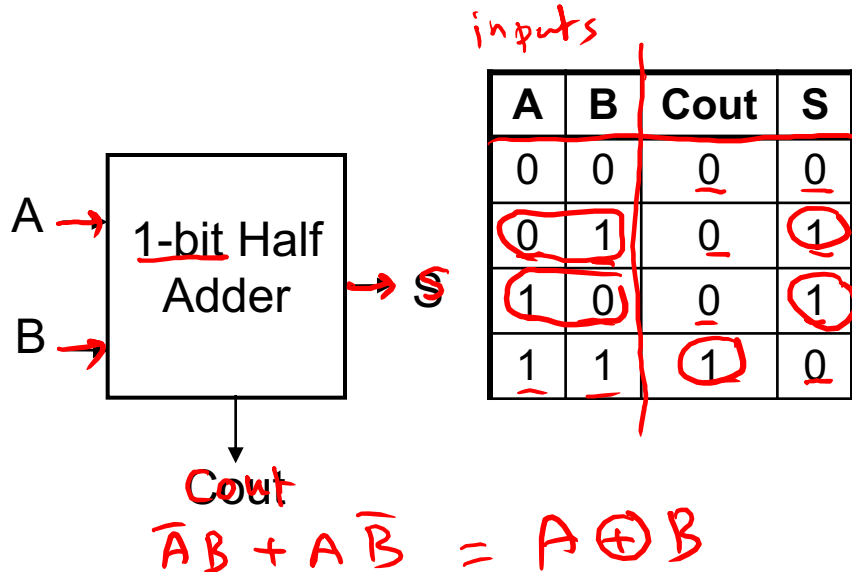
negation
 $N \Rightarrow -N$
 $-N = \bar{N} + 1$

Binary Addition



add x_5, x_6, x_7

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



Truth Table for 1-bit Full Adder:

inputs			output	
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

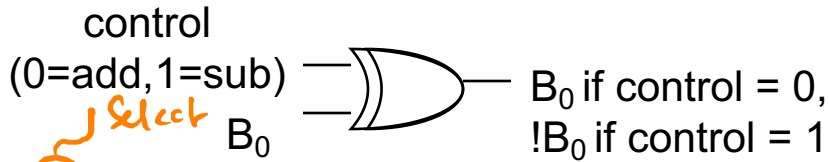
$S = A \oplus B$ (odd parity function)
 \hookrightarrow XOR
 $Cout = A \& B$ (majority function)

$S = A \oplus B \oplus Cin$ (odd parity function)
 $Cout = A \& B \mid A \& Cin \mid B \& Cin$ (majority function)

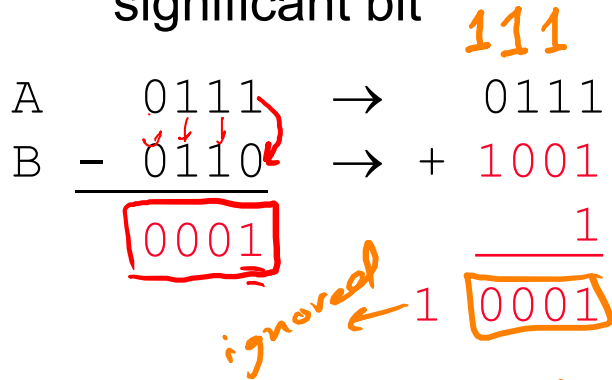
- How can we use it to build a 64-bit adder?
- How can we modify it easily to build an adder/subtractor?

A 64-bit Ripple Carry Adder/Subtractor

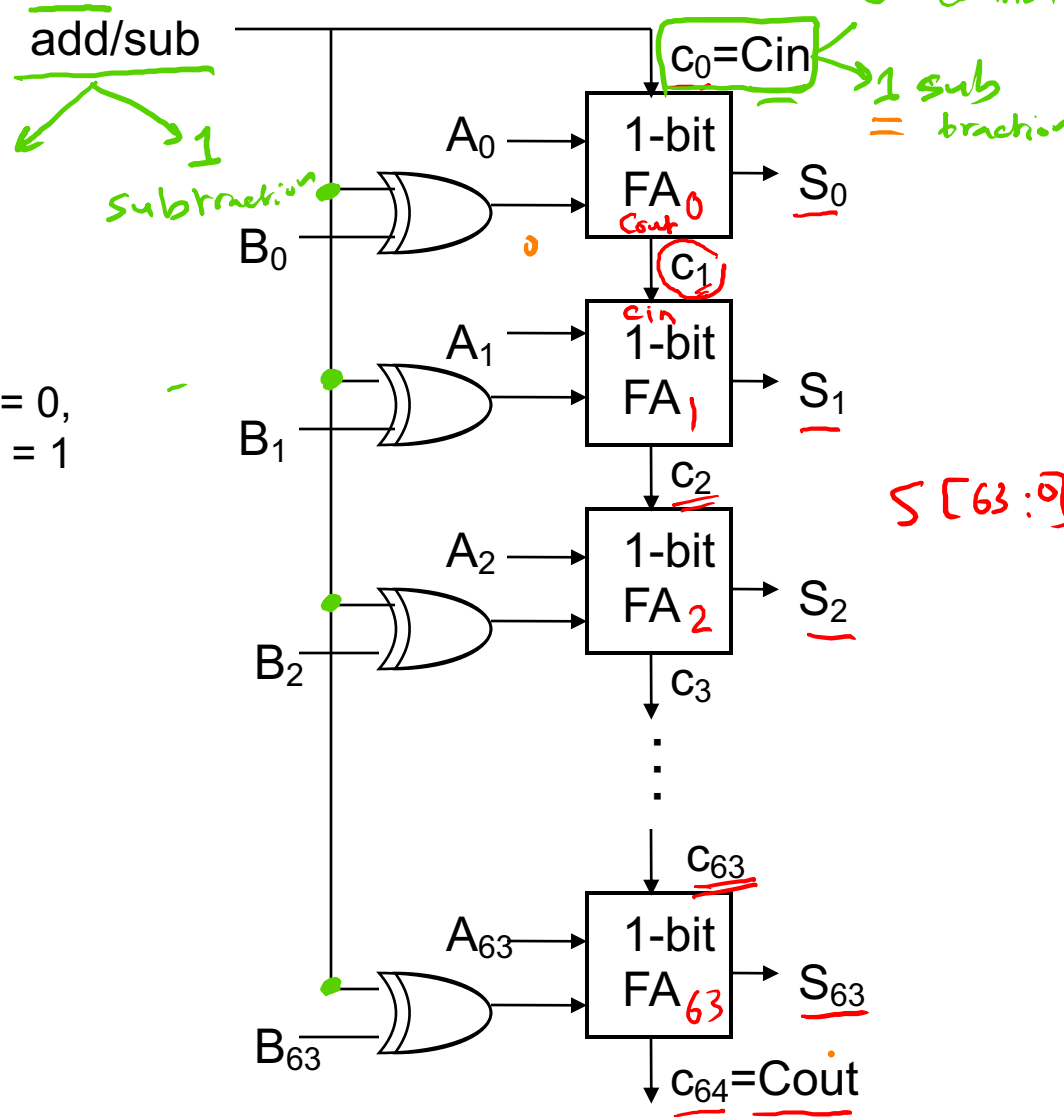
- Remember 2's complement is just
 - addition
 - complement all the bits



- add a 1 in the least significant bit

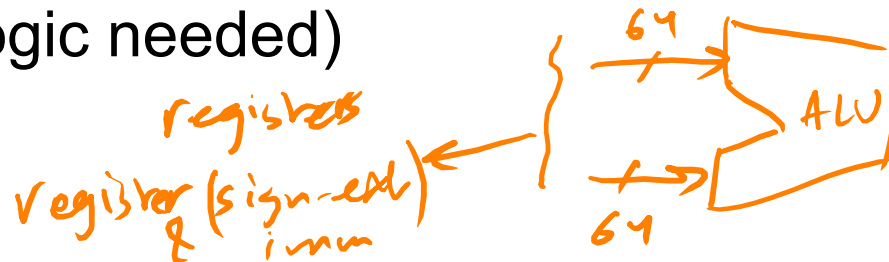


$$A - B = A + (\overline{B} + 1)$$



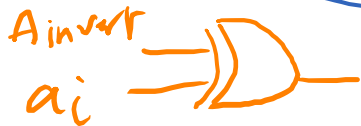
Tailoring the ALU to the RISC-V ISA

- ❑ 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: $(a - b) = \underline{0}$ 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)



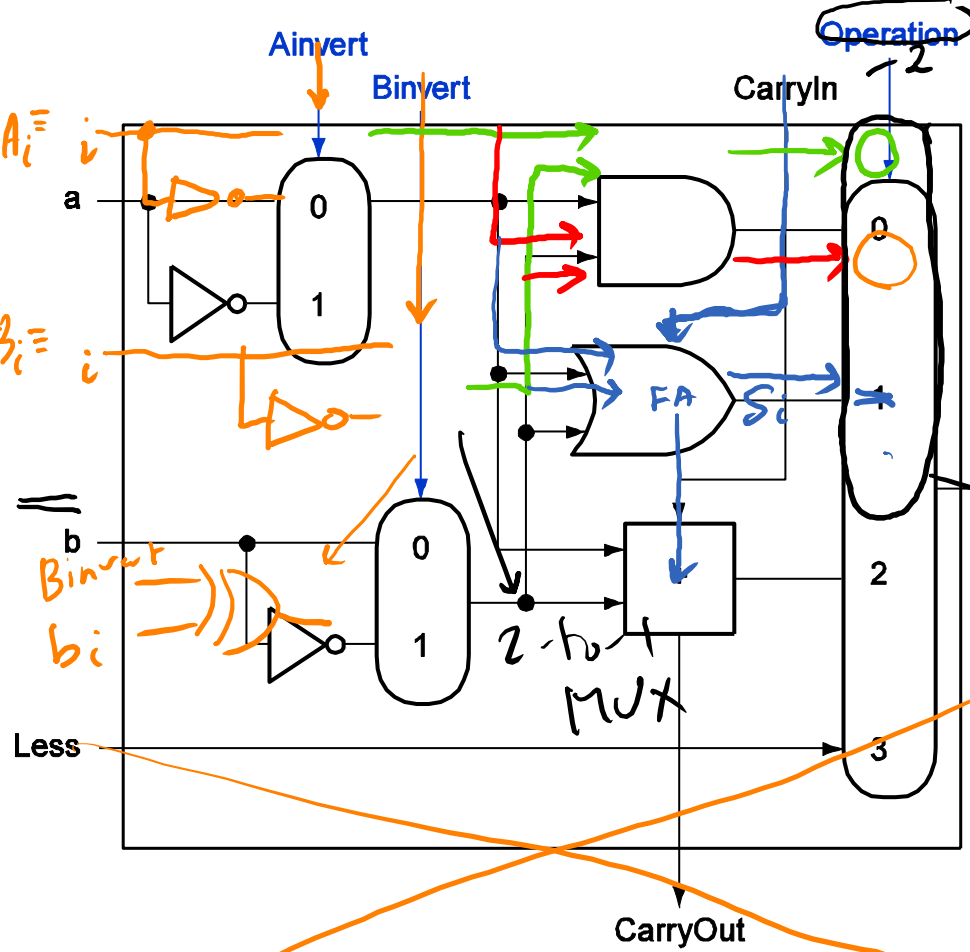
RISC-V ALU

Least-significant bits



~~and~~ or X_{10}, X_{20}, X_{30} A_{invert}

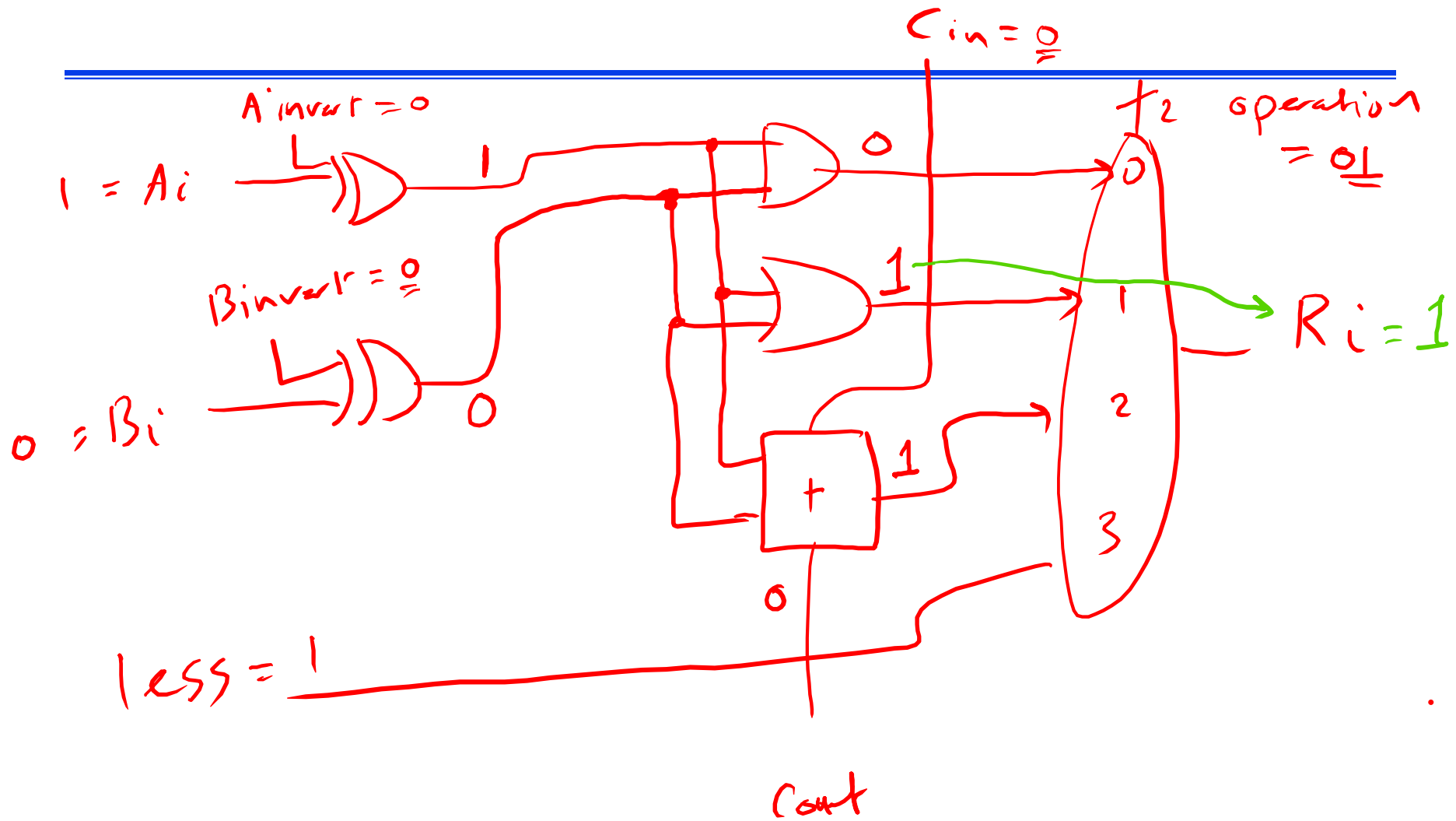
Function	Binvert	Operation	
✓ <u>and</u>	<u>0</u>	<u>00</u>	0
✓ <u>or</u>	0	<u>01</u>	6
✓ add	<u>0</u>	<u>10</u>	0
✓ sub	<u>1</u>	<u>10</u>	0
✓ <u>slt</u>	1	<u>11</u>	0
<u>nor</u>	1	00	1

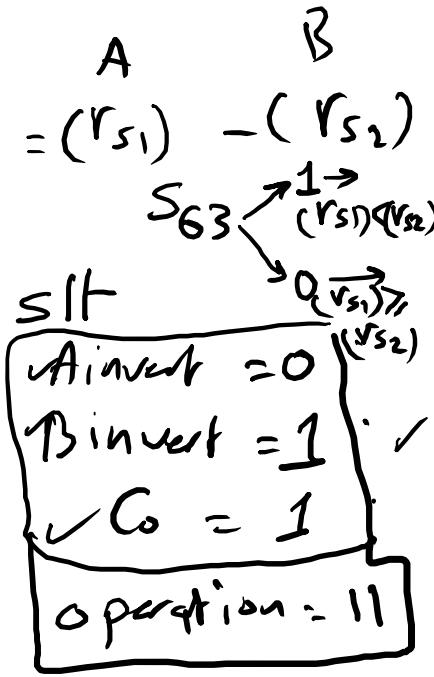
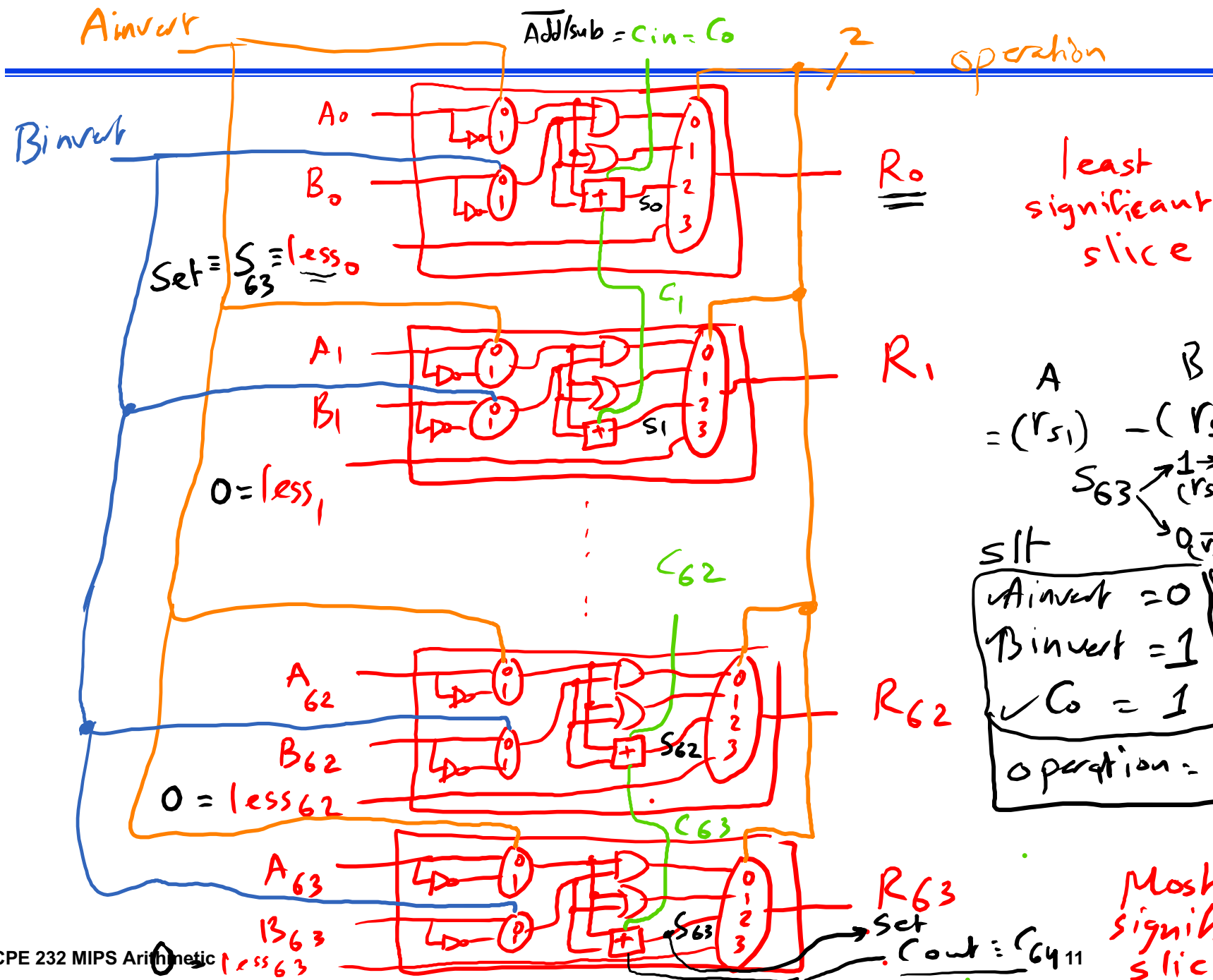


4-to-1 MUX

$$A - B = A + \bar{B} + 1$$

nor rd, rs1, rs2
 not(rs1) or(rs2)
 not(rs1) and⁹ not(rs2)





Set if less than (slt)

❑ slt rd, rs1, rs2

- If $(rs1) < (rs2) \rightarrow rd = (00000000 \dots 00001)_{two}$
- If $(rs1) \geq (rs2) \rightarrow rd = (00000000 \dots 00000)_{two}$

bit A, B, Label
 RISC-V
 if (A < B)
 ↓ MIPS
 1 ←
 { slt C, A, B
 base C, zero, label

❑ For ALU₁ to ALU₆₃, connect the **Less** input to ground or zero

↓ slice₁ ↓ most significant slice

❑ Use subtraction to determine if rs1 is less than rs2

- rs1 - rs2 is negative → **adder/subtractor output in ALU₆₃ is 1**
 - The adder/subtractor output of ALU₆₃ is called **Set** and is connected to the **Less** input of ALU₀
- (S63) → 1
 less = 1

❑ 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

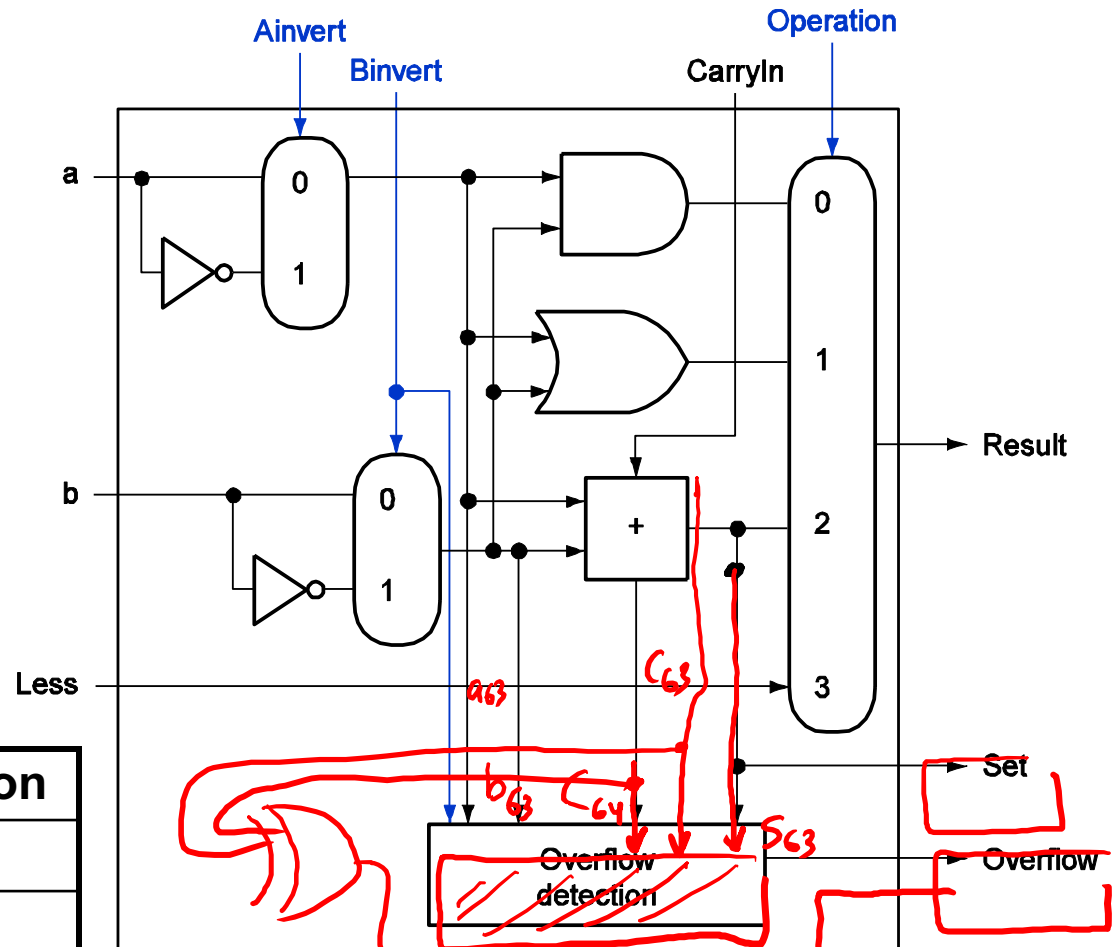
slt u
 slti rd, rs1, imm
 sltiu rd, rs1, imm

RISC-V ALU

Most-significant bit

might cause an overflow

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



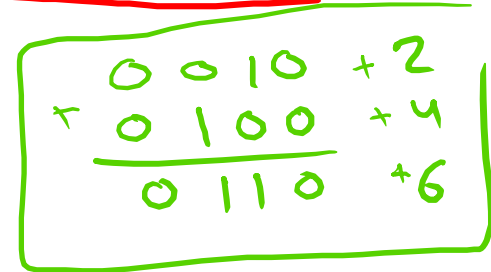
Overflow
 $A_{64} + B_{64} \rightarrow AW_{64} \rightarrow R$
 $B_{64} \rightarrow AW_{64}$
 $A_{64}, B_{64}, Operation [1:0]$

Overflow Detection

❑ Overflow: the result is too large to represent in 64 bits

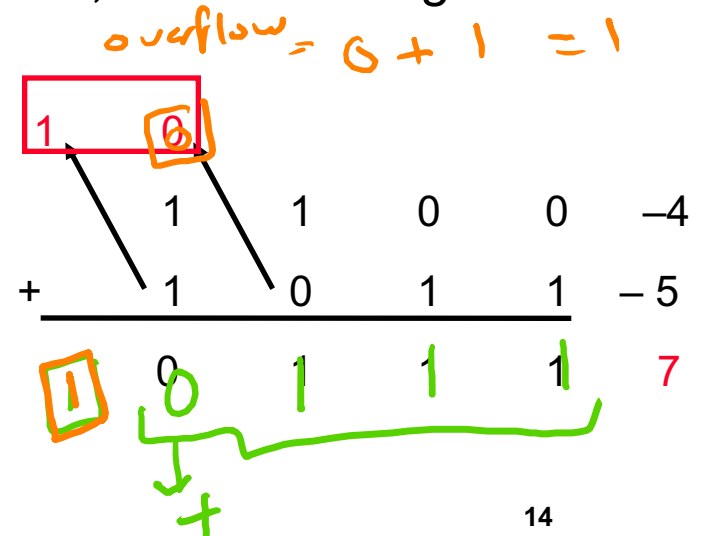
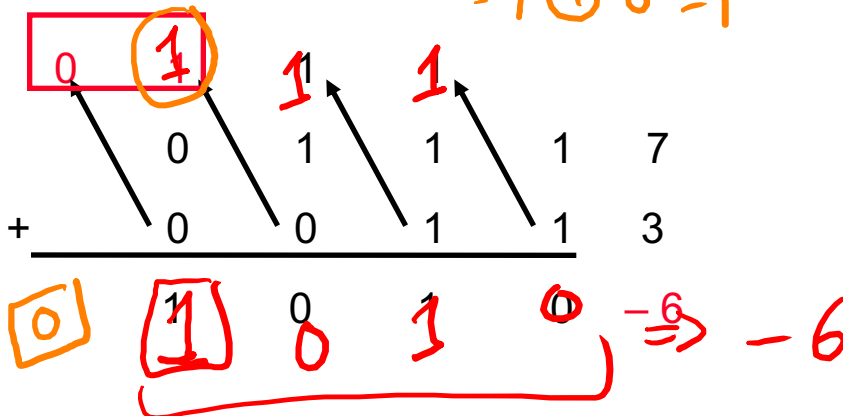
❑ Overflow occurs when

- 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
- $(+) - (-)$
 $(+) + (+)$
 $(-) - (+) \Rightarrow (-) + (-)$



❑ On your own: **Prove** you can detect overflow by:

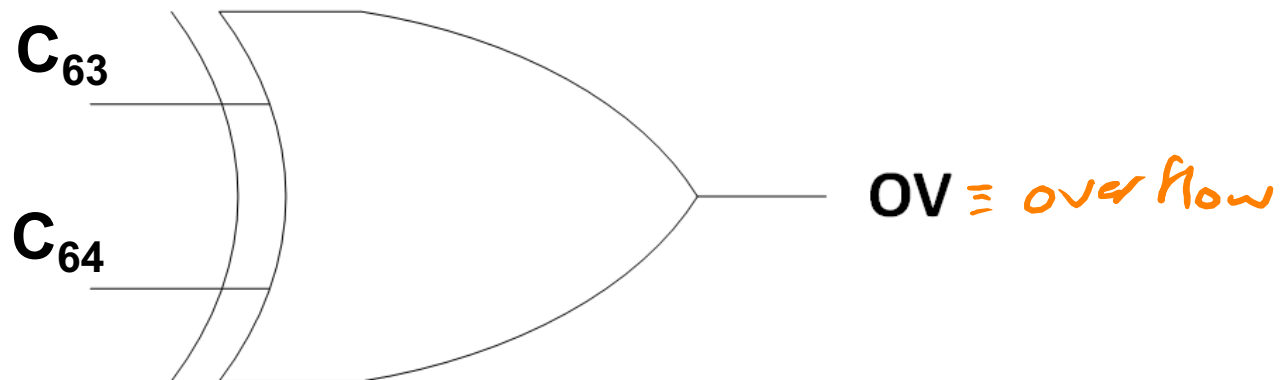
- Carry into MSB **XOR** Carry out of MSB, ex for 4 bit signed numbers



Overflow Detection

inputs *outputs*

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: Relation between Binvert & C₀

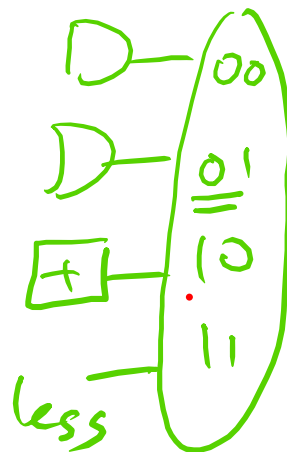
- For sub, slt, and branch: Binvert = 1 and C₀ = 1

$= A - B$
 $= A + \overline{B} + 1$
- For add: Binvert = 0 and C₀ = 0
- For logical and, or except NOR: Binvert = 0 and C₀ = X = 0

→ don't care
- For NOR: Binvert = 1 and C₀ = X = 1

→ don't care
 $a \text{ nor } b = \overline{a} \text{ and } \overline{b}$
- Based on above: Binvert and C₀ are merged together into once signal called **Bnegate**

Ainvert	Bnegate	Operation	
<u>0</u>	<u>0</u>	<u>00</u>	<u>AND</u>
<u>0</u>	<u>0</u>	<u>01</u>	<u>OR</u>
<u>0</u>	<u>0</u>	<u>10</u>	<u>ADD</u>
<u>0</u>	<u>1</u>	<u>10</u>	<u>SUB/Branch</u>
<u>0</u>	<u>1</u>	<u>11</u>	<u>SLT</u> ($(rs_1) < (rs_2)$)
<u>1</u>	<u>1</u>	<u>00</u>	<u>NOR</u>



Ainvert

Bnegate

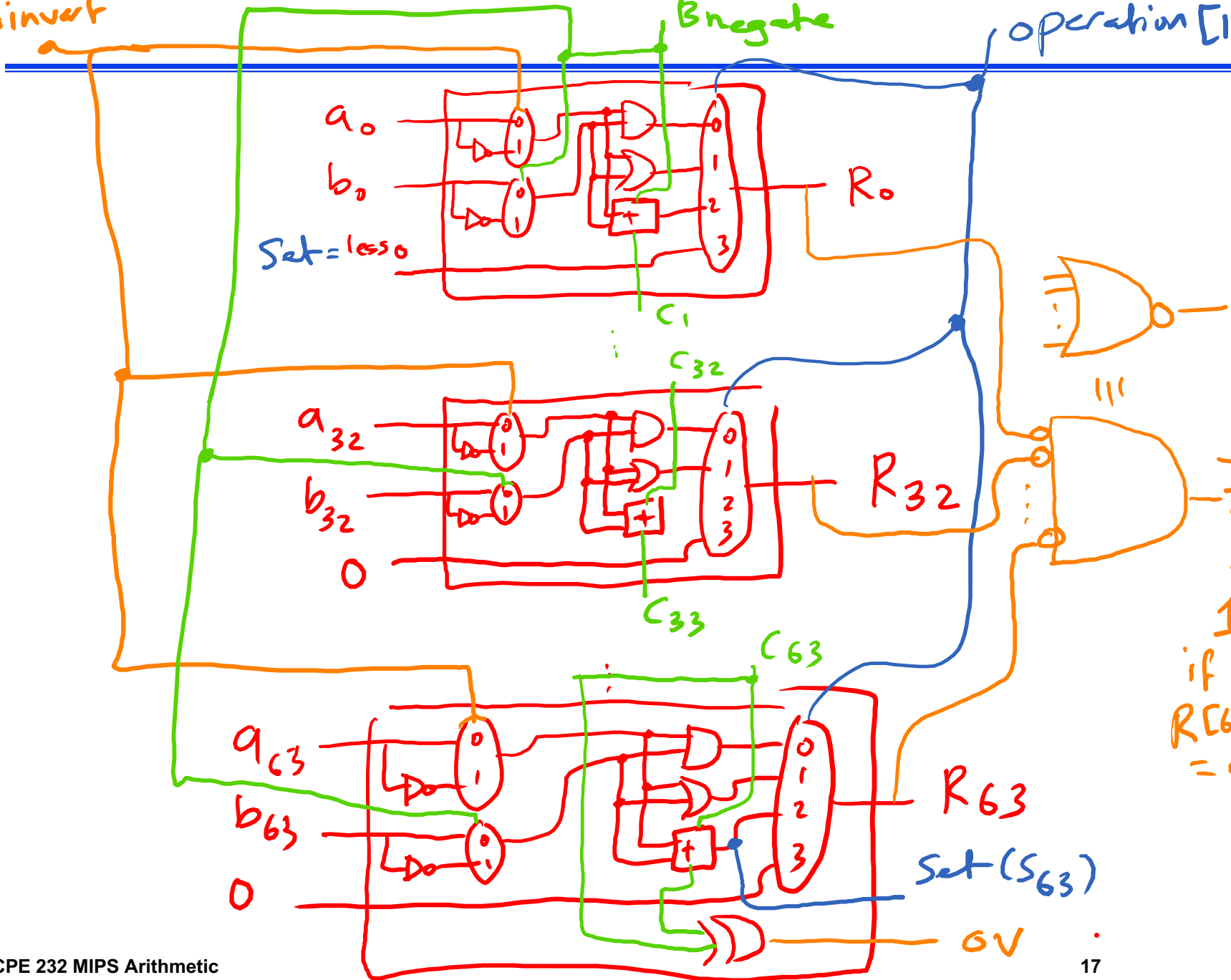
operation [1:0]

Set = less0

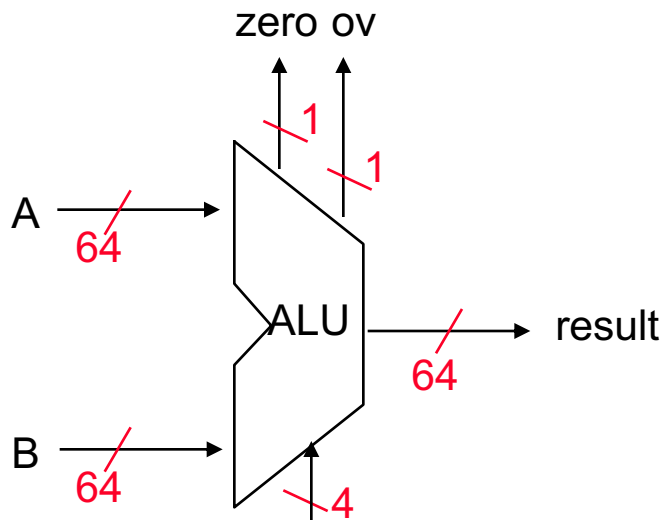
Set (S₆₃)

OV

Z
↓
1
if
R[63:0]
= 0

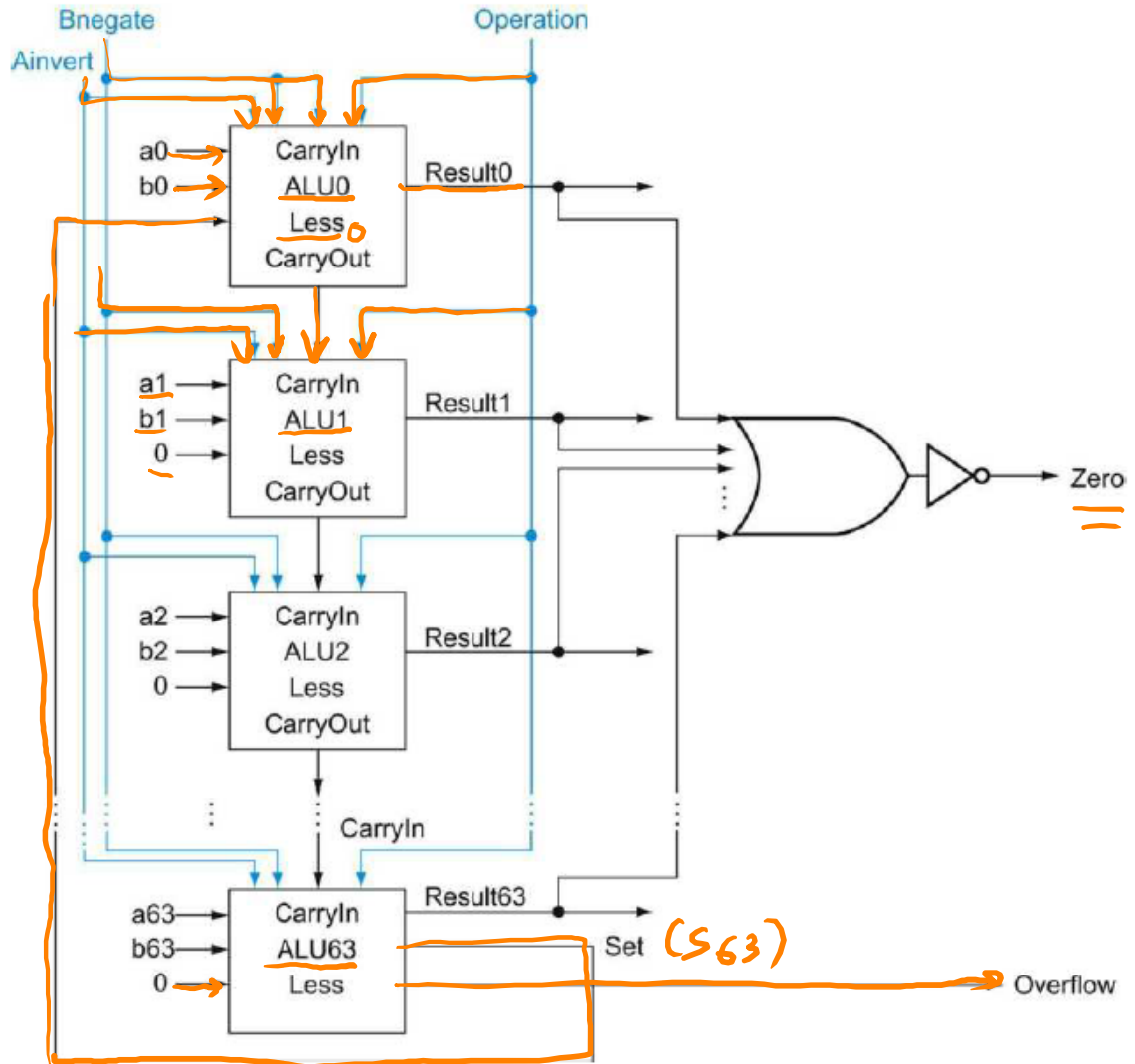


RISC-V ALU



“Ainvert, Bnegate, Operation”

[1..0]

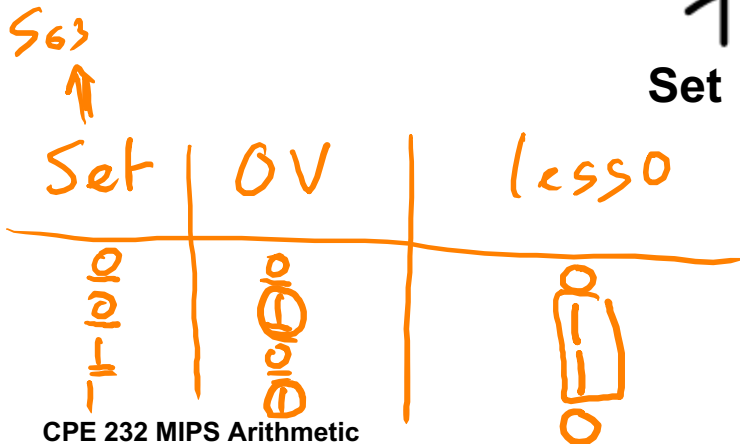
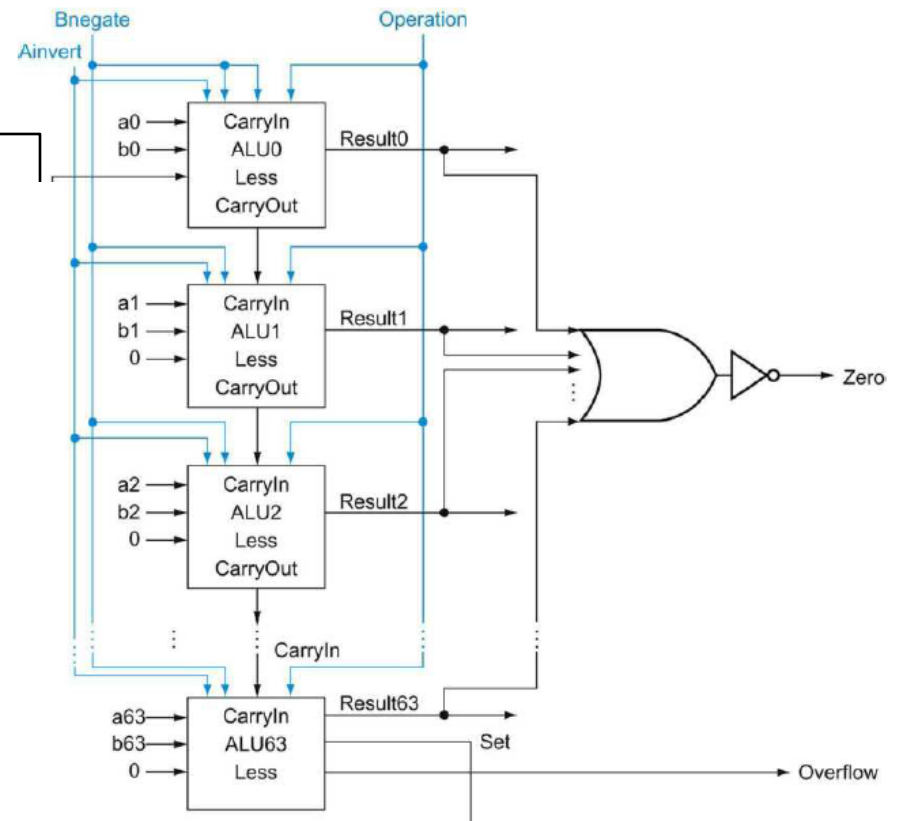
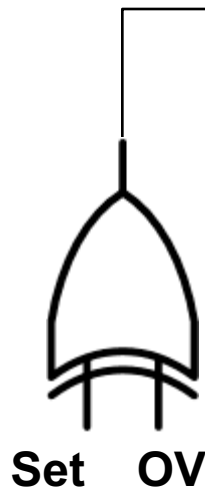


Set if less than correction

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

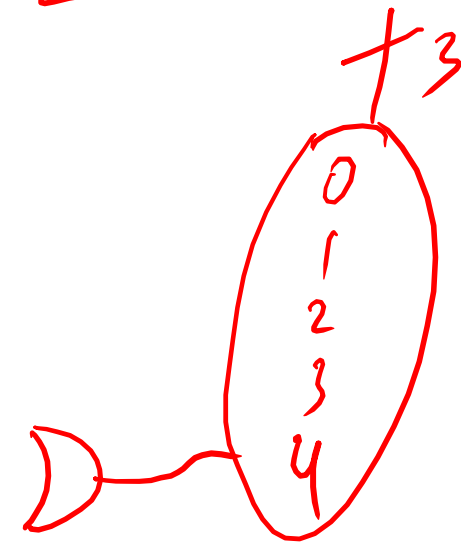
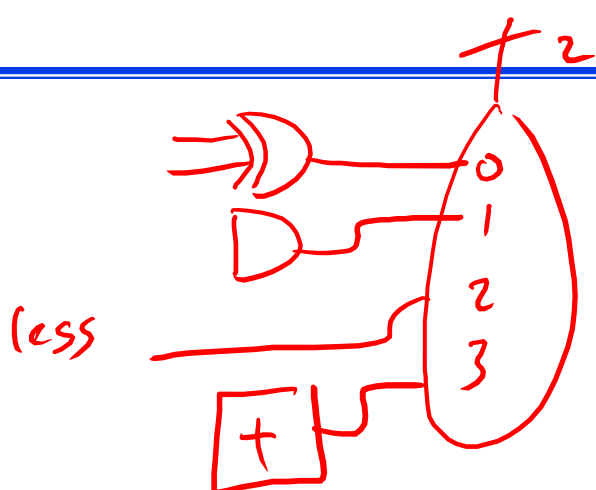
$$Less_0 = Set \oplus OV$$

Set	OV	Less ₀
0	0	0
0	1	1
1	0	1
1	1	0



$less_0 = Set \oplus OV$

	operation
xor	000
AND	001
sh	010
ADD/sub	011
OR	100



~~sh~~

. |

set if greater than or equal

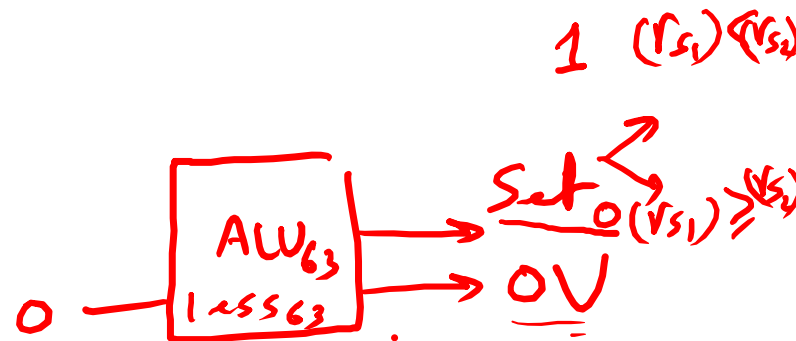
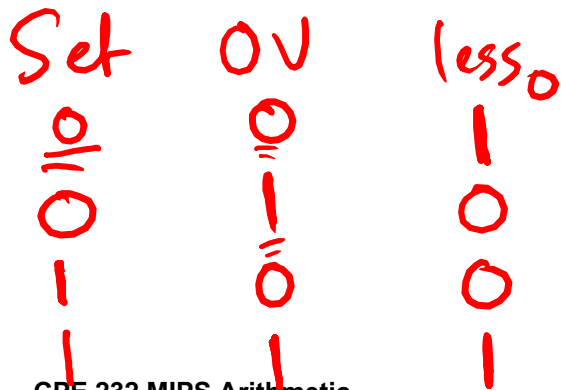
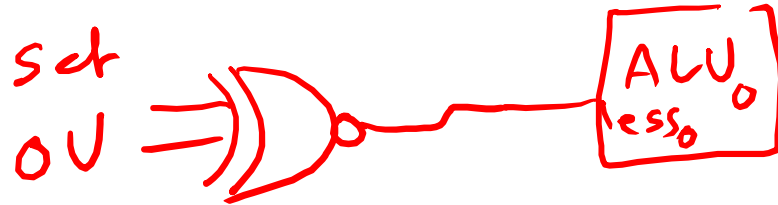
sge rd, rs1, rs2

if $(rs1) \geq (rs2)$

rd = 000...0

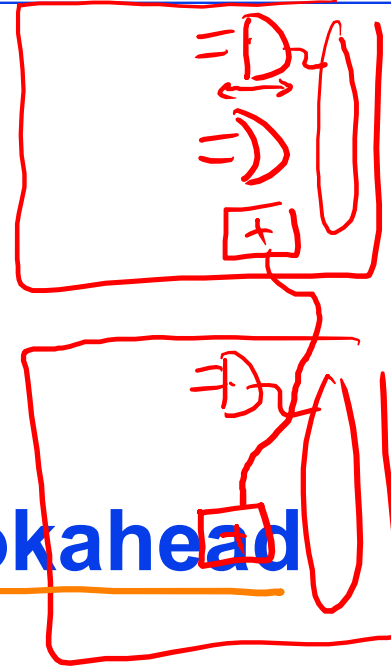
else

rd = 000...0



Appendix A.6:

Faster Addition: Carry Lookahead



Delay of logical operation = 1 gate delay + delay of MUXes

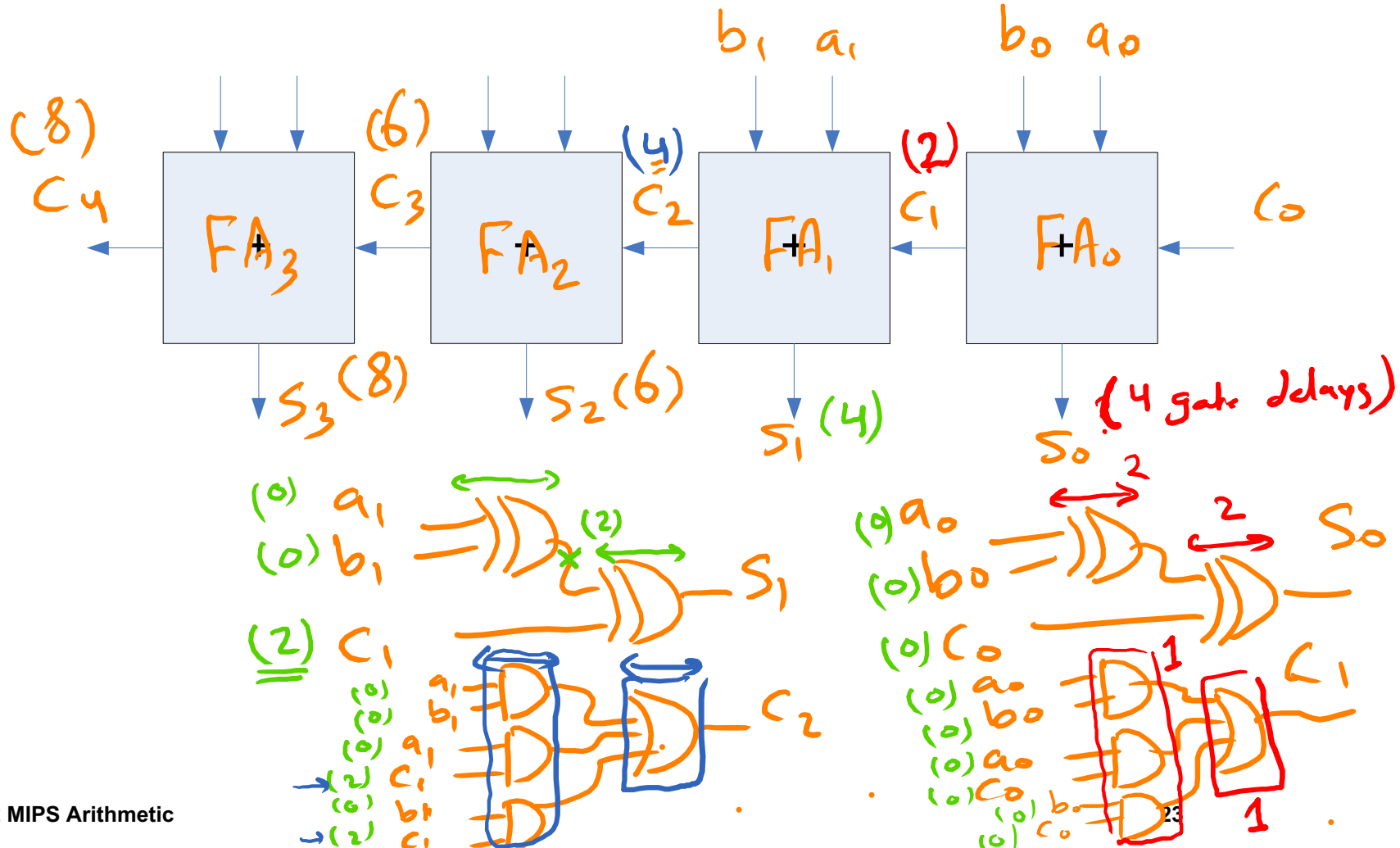
Delay of add/sub & slt = delay of 64-bit Ripple Carry adder/sub + delay of MUXes

Improving Addition Performance

❑ The ripple-carry adder is slow

Delay of 4-bit RCA = 8 gate delays

Delay of basic gate = 1 gate delay
 " " complex " = 2 gate delays
 XOR, XNOR



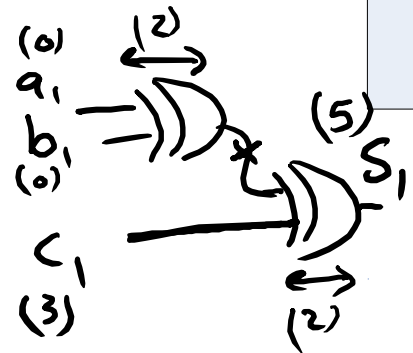
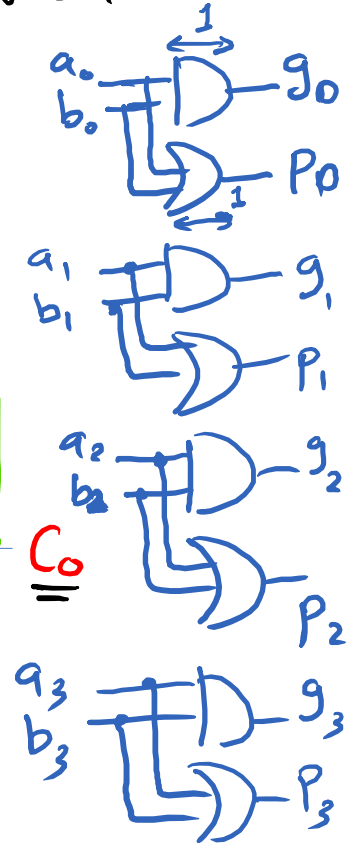
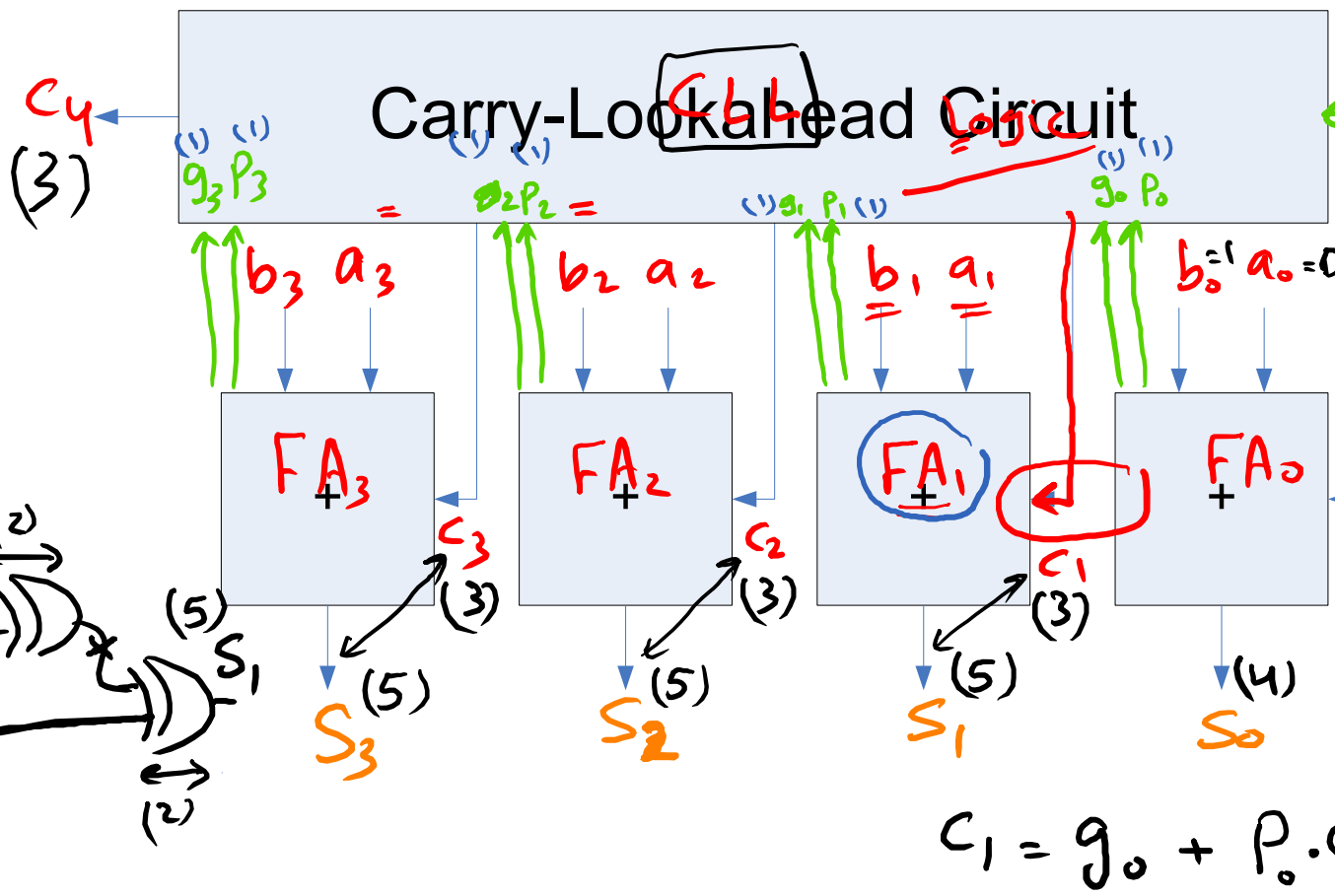
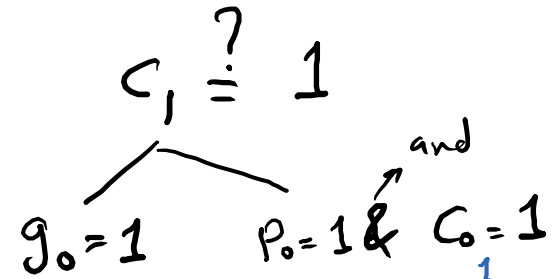
Delay of w -bit RCA = $2 * n$

Delay of 64-bit RCA = 128 gate delay

Carry-Lookahead Adder

Need fast way to find the carry

4-bit CLA

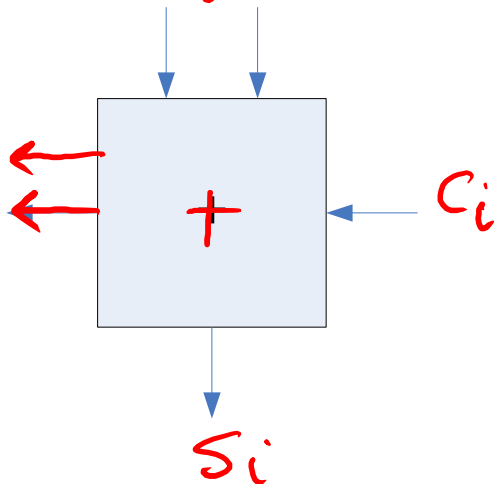


$$C_1 = g_0 + P_0 \cdot C_0$$

Carry-Lookahead Adder

- Carry generate and carry propagate

Carry propagate $\equiv P_i$
 carry generate $\equiv g_i$



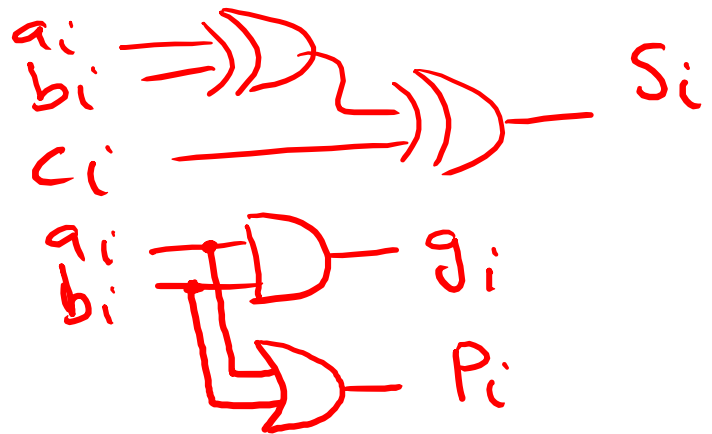
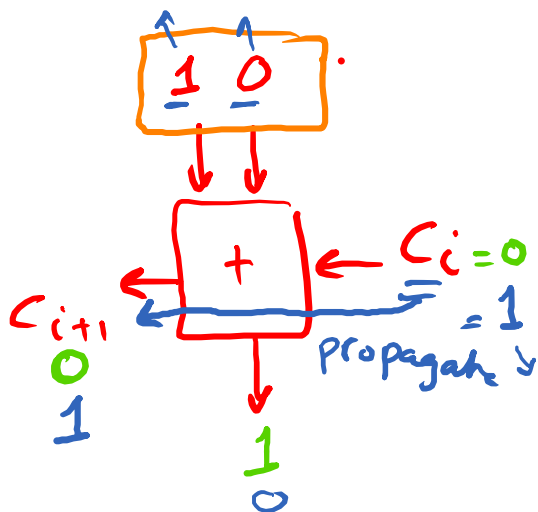
<u>a_i</u>	<u>b_i</u>	<u>g_i</u>	<u>p_i</u>
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

$g_i = a_i \cdot b_i$

$p_i = a_i + b_i$

logical OR

$S_i = a_i \oplus b_i \oplus C_i$

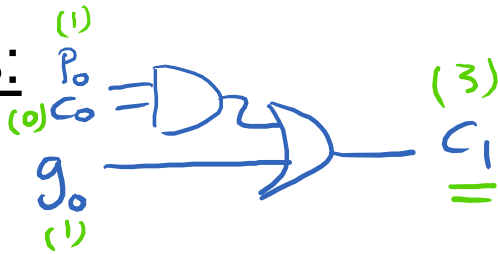


Carry-Lookahead Adder

Carry Equations:

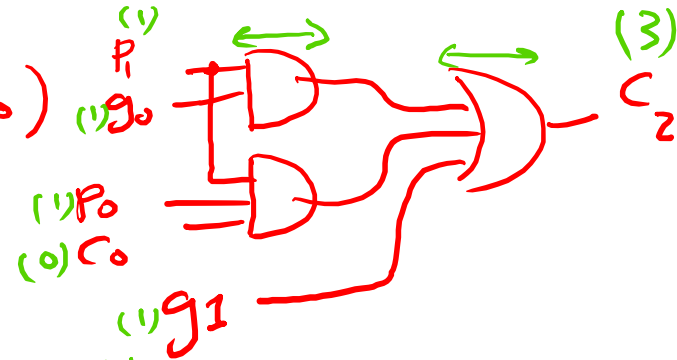
$$c_1 = g_0 + p_0 c_0$$

SOP
"2-level circuit"



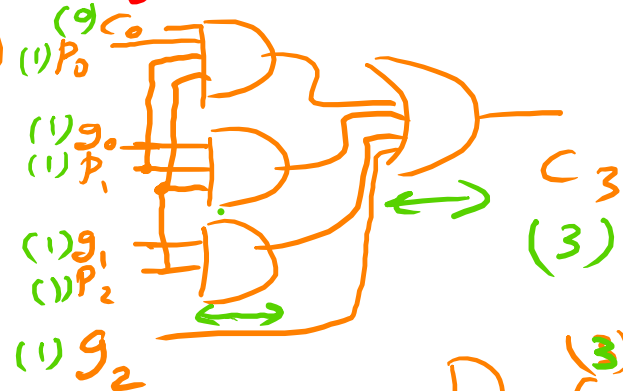
X $c_2 = g_1 + p_1 c_1 = g_1 + p_1 (g_0 + p_0 c_0)$

✓ $c_2 = g_1 + p_1 g_0 + p_1 p_0 c_0$



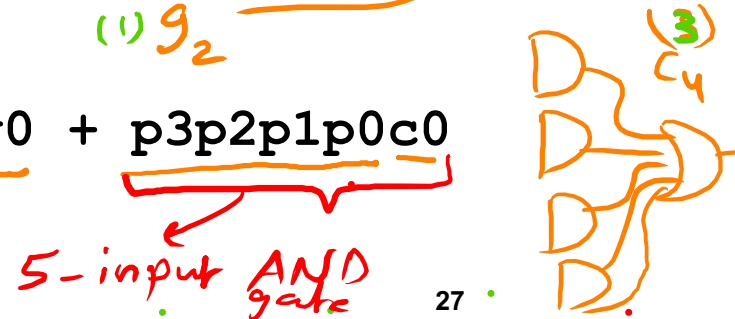
X $c_3 = g_2 + p_2 c_2 = g_2 + p_2 (g_1 + p_1 g_0 + p_1 p_0 c_0)$

✓ $c_3 = g_2 + p_2 g_1 + p_2 p_1 g_0 + p_2 p_1 p_0 c_0$



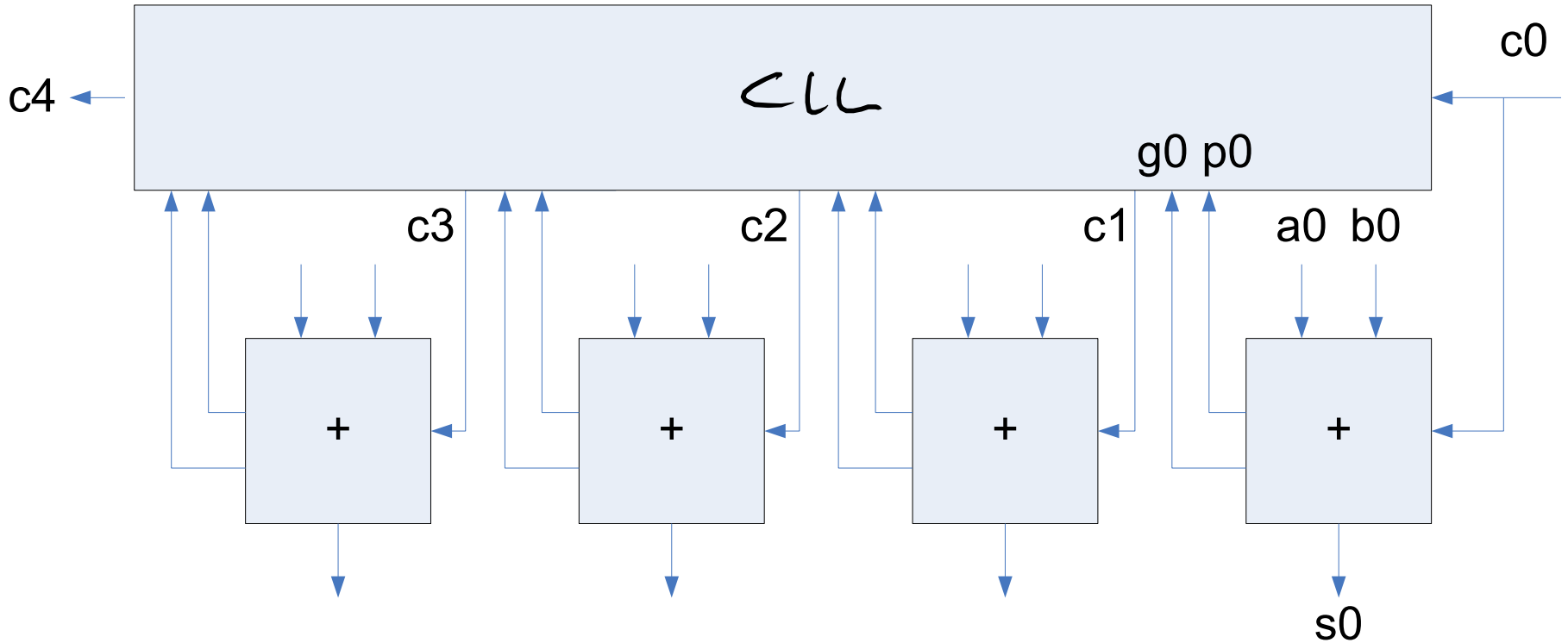
X $c_4 = g_3 + p_3 c_3$

✓ $c_4 = g_3 + p_3 g_2 + p_3 p_2 g_1 + p_3 p_2 p_1 g_0 + p_3 p_2 p_1 p_0 c_0$



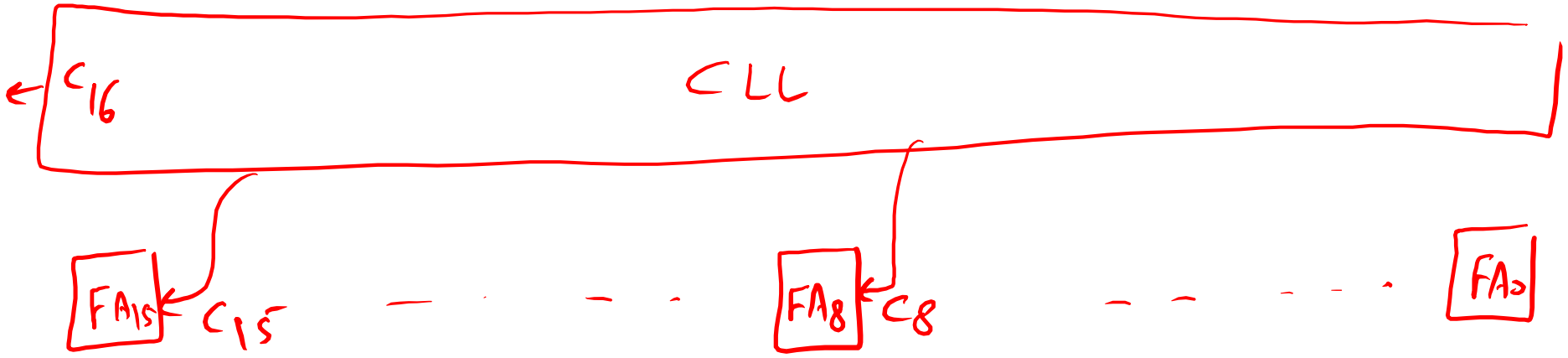
4-bit Carry-Lookahead Adder

Delay of 4-bit CLA = 5 gate delays
" " " RCA = 8 gate delays

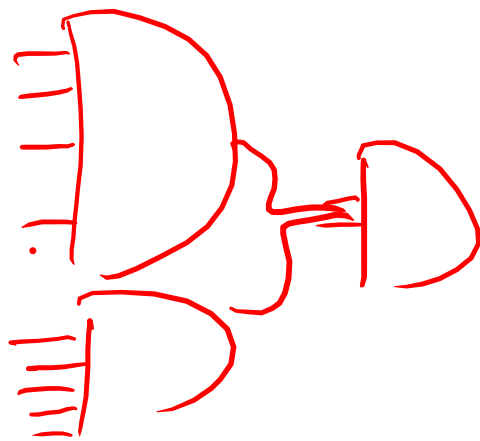
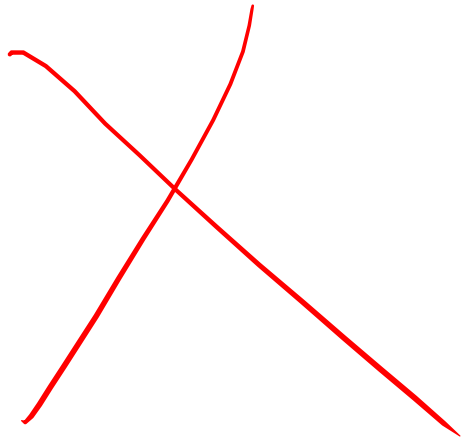


Wrong 16-bit CLA

(Delay of 16-bit RCA = 32 gate delays)



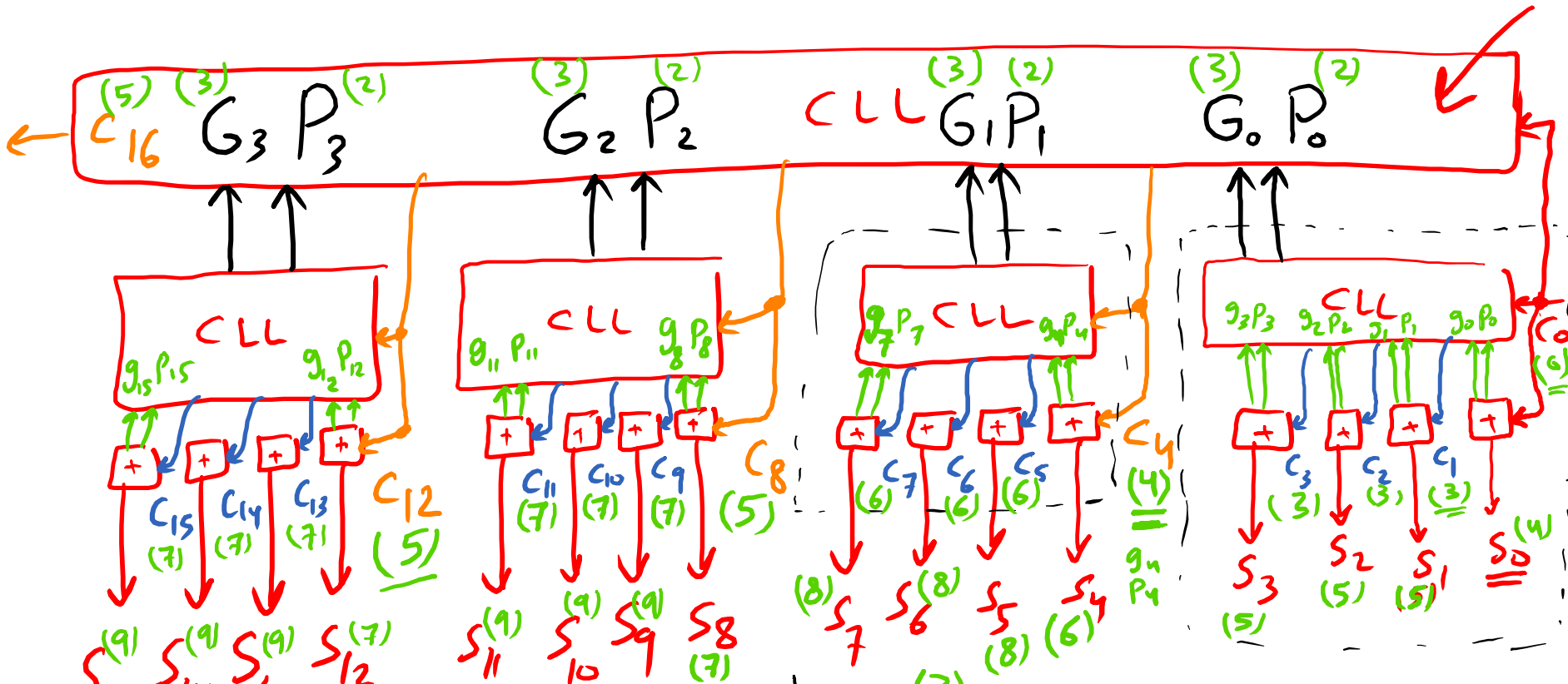
$$c_8 = g_7 + P_7 g_6 + P_7 P_6 g_5 + P_7 P_6 P_5 g_4 + \dots + \boxed{\begin{matrix} P_7 P_6 P_5 P_4 P_3 P_2 \\ P_1 P_0 C_0 \end{matrix}}$$



9-input AND gate

Correct 16-bit CLA

HW: Write equations for $G_2, P_2, G_3, P_3, C_{16}$



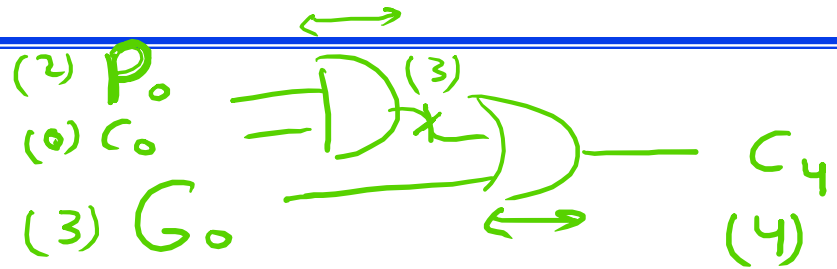
$$G_1^{(3)} = g_7 + P_7 g_6 + P_7 P_6 g_5 + P_7 P_6 P_5 g_4$$

$$P_1^{(2)} = P_7 P_6 P_5 P_4$$

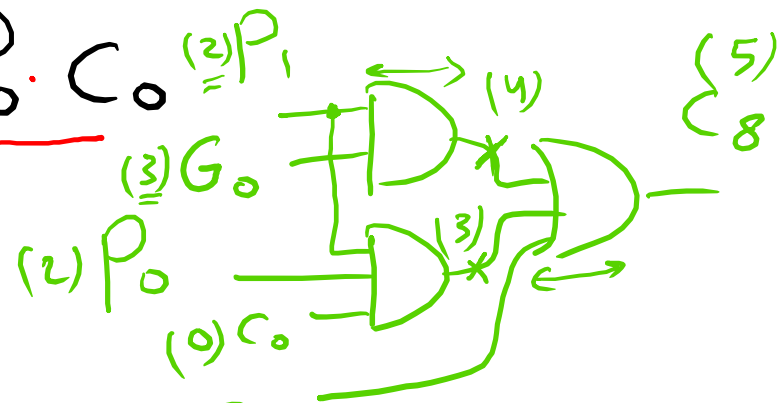
$$G_0^{(3)} = g_3 + P_3 g_2 + P_3 P_2 g_1 + P_3 P_2 P_1 g_0$$

$$P_0^{(2)} = P_3 P_2 P_1 P_0$$

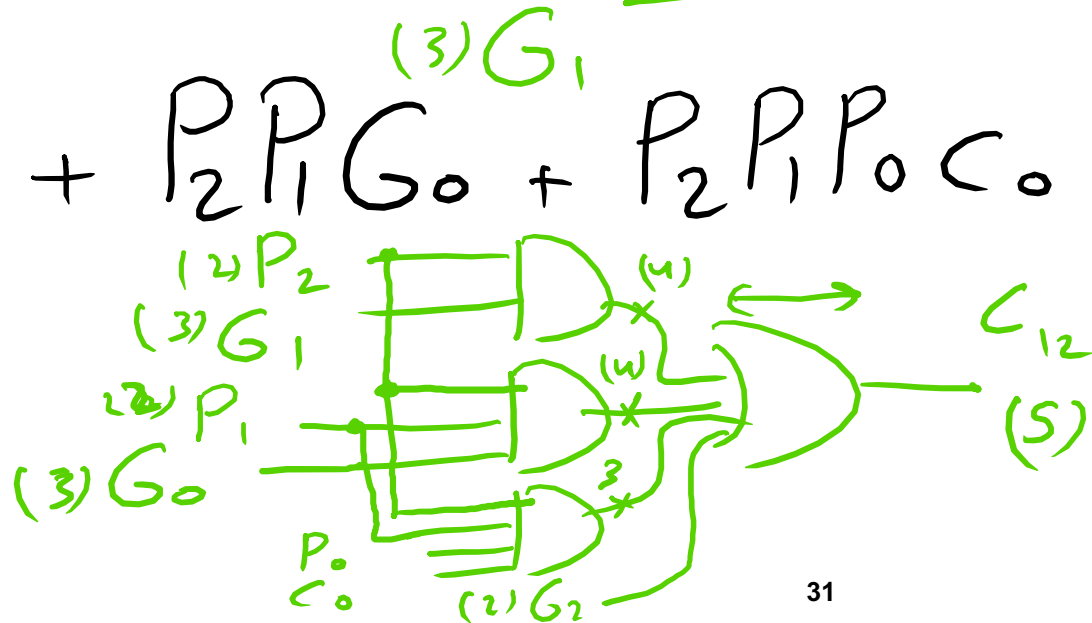
$$C_4 = G_0 + P_0 \cdot C_0$$

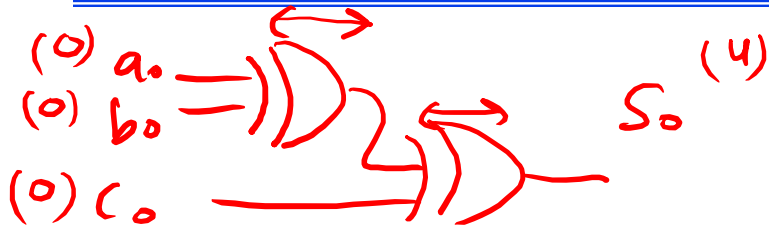


$$C_8 = G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot C_0$$

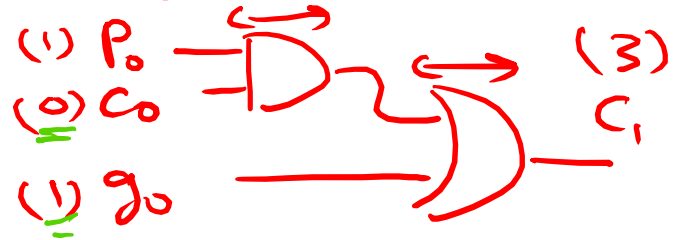


$$C_{12} = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0$$

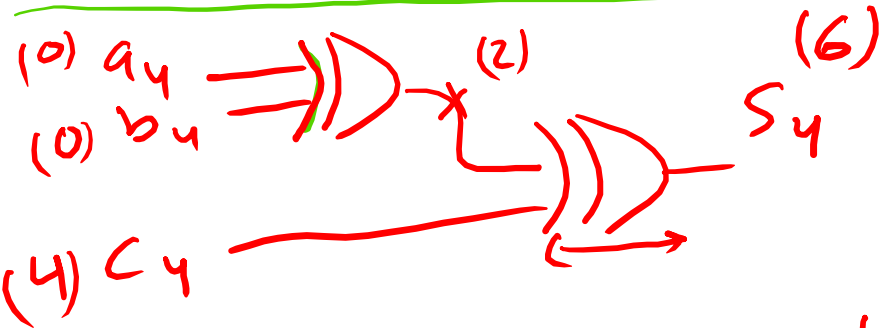
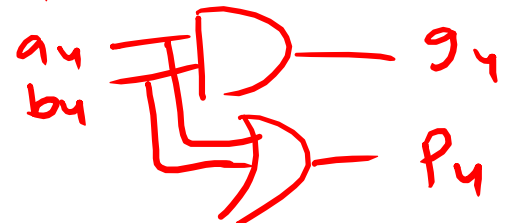
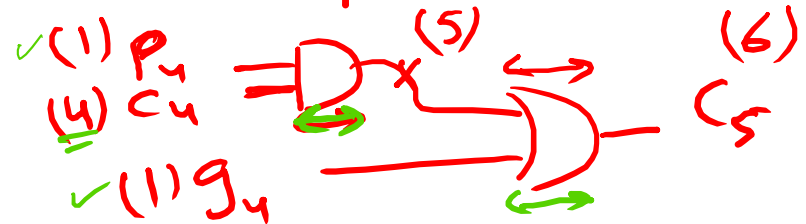




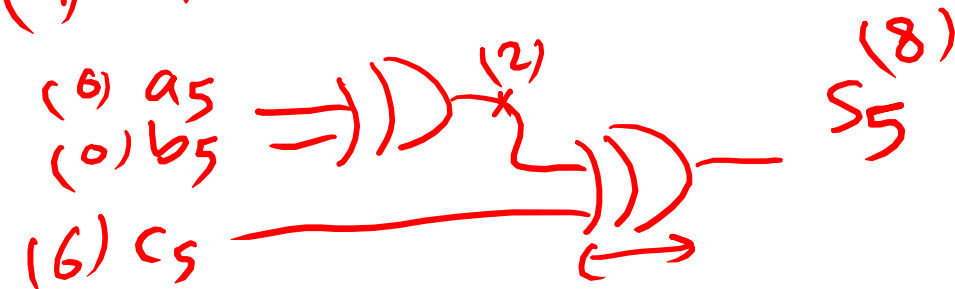
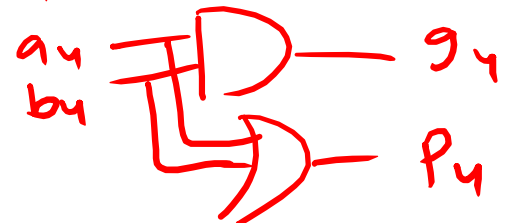
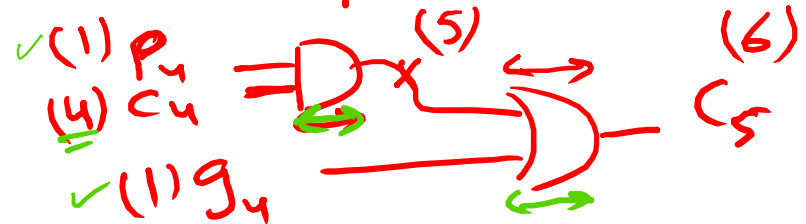
$$C_1 = g_0 + P_0 \cdot C_0$$

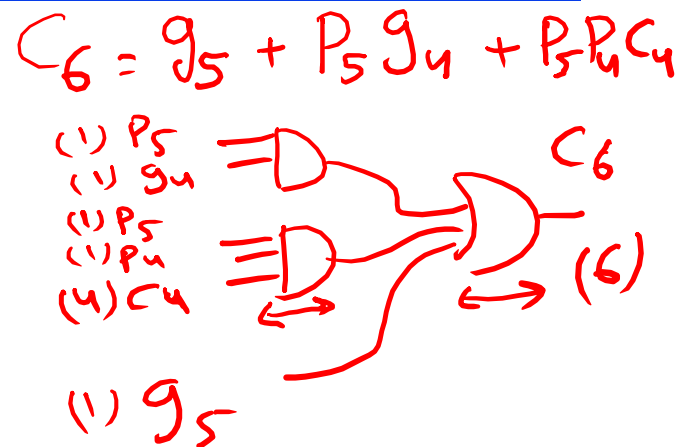
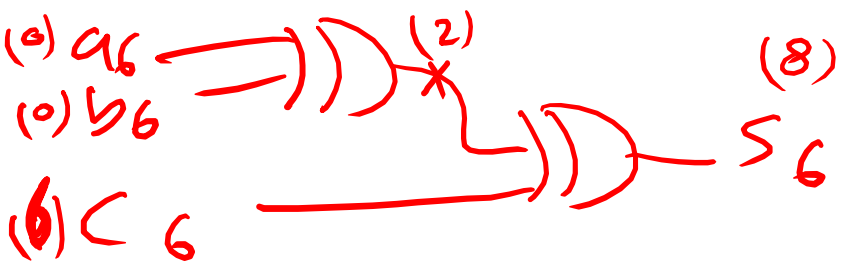


$$C_2 = g_1 + P_1 \cdot C_1$$



$$C_5 = g_4 + P_4 \cdot C_4$$





Delay of 16-bit CLA = 9 gate delays

// // // RCA = 32 // //

16-bit CLA is 3.5 times faster than
16-bit RCA

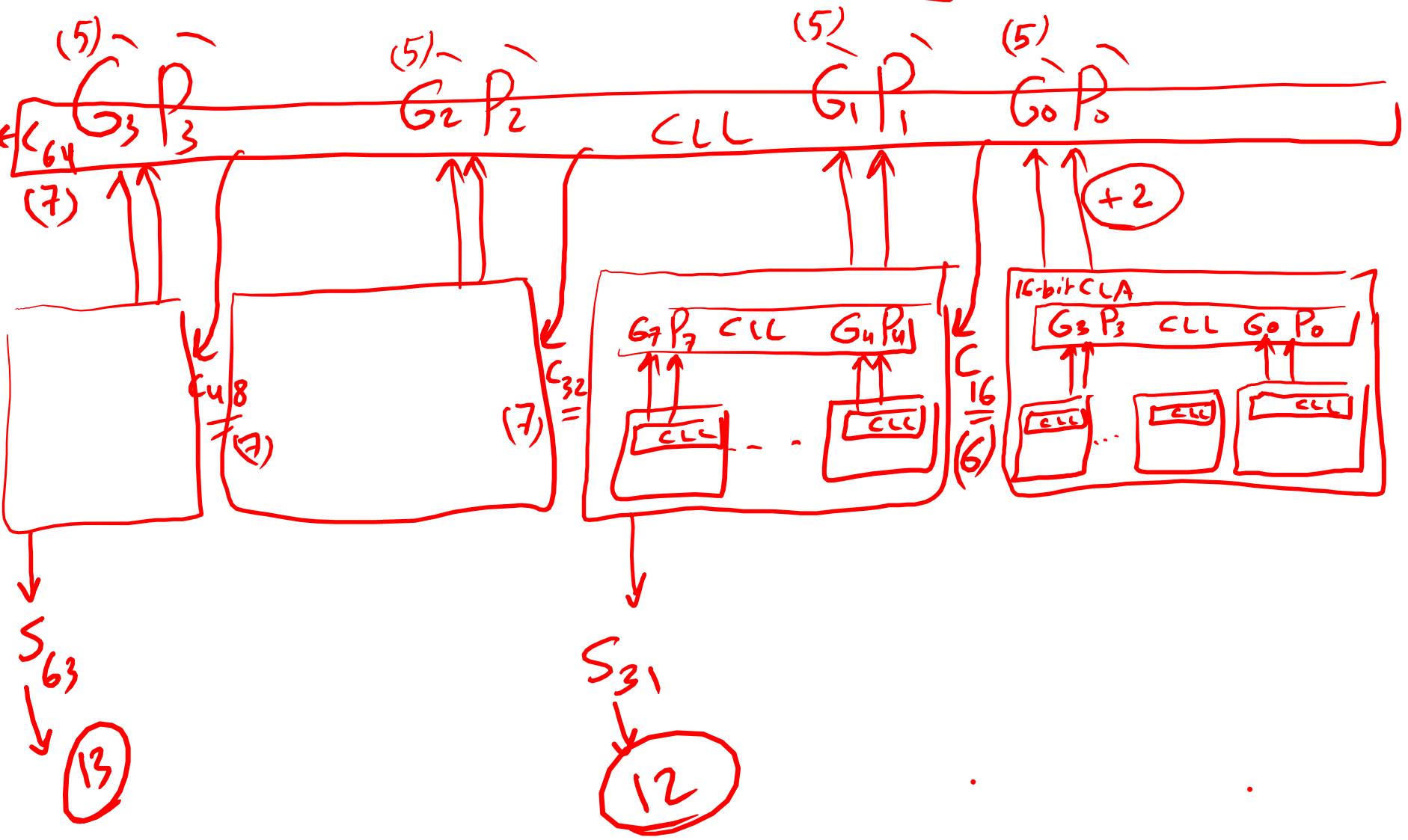
4-bit CLA \Rightarrow Delay = 5 gate delays
16-bit " \Rightarrow " = 9 gate delays

For every additional "CLA" level, the delay of CLA is incremented by 4

Why? ① we need 2 ^{extra} gate delays to compute G_i, P_i

② we need 2 extra gate delays to compute the group carry ins (C_4, C_8, C_{12}, \dots)

64-bit CLA



Larger Carry-Lookahead Adders

$$P_0 = p_0 p_1 p_2 p_3$$

$$G_0 = g_3 + g_2 p_3 + g_1 p_2 p_3 + g_0 p_1 p_2 p_3$$

16-bit CLA

