Computer Architecture

Ngo Lam Trung & Pham Ngoc Hung
Faculty of Computer Engineering
School of Information and Communication Technology (SoICT)
Hanoi University of Science and Technology
E-mail: [trungnl, hungpn]@soict.hust.edu.vn

Course administration

- Instructor: Ngo Lam Trung/Pham Ngoc Hung
  803 B1, SoICT, HUST
- Text: [Required] Computer Organization and Design, 5th edition revised printing Patterson & Hennessy 2014.

  [Optional] Computer Organization and Architecture, 10th Edition, William Stalling
- Slides: pdf
- Schedule: as in timetable

Course content

- Chapter 1: Introduction
- Chapter 2: Computer Functions and Interconnection
- Chapter 3: Computer Arithmetic
- Chapter 4: Instruction Set Architecture
- Chapter 5: CPU
- Chapter 6: Memory
- Chapter 7: I/O system
- Chapter 8: Multicores and multiprocessors

Computers are so important

- Current modern life
  - Industrial revolutions, the 3rd (Automation) and the 4th (Digital revolution).
  - Cell phones, the Internet, Grab, Google Maps...
  - WWW, search engines, social networks, e-commerce…
  - Robotics, EV, UAV, self-driving cars,…
- Future
  - Tailored medical care based on individual genome.
  - Super-human: transfer human's brain to a mechanical body (robot) for interstellar traveling (The Matrix franchise, Michio Kaku, *Physics of the Future 2011* and *The Future of the Mind 2015).*
  - *…many more*

Outcomes from this course

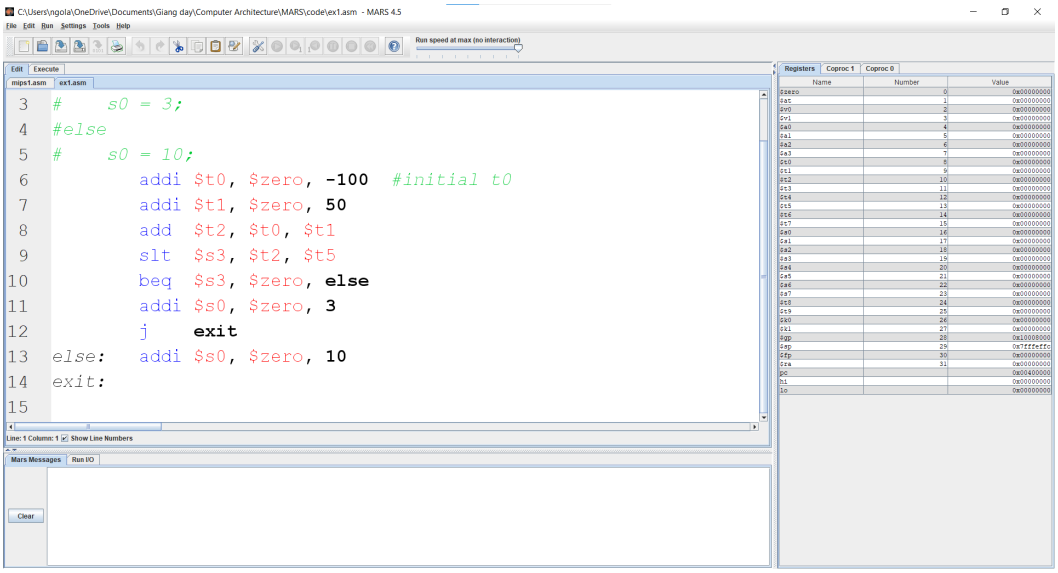
- Computer Architecture and Organization
  - Understanding of basic computer system organization.
  - Abstraction and instruction set architecture: how high-level language programs translate into computer language programs, and how hardware execute the latter programs.
  - Hardware/software interface, and how software instructs hardware to perform functions.
- Computer performance
  - How to evaluate performance
  - Basic techniques to improve computer performance.

Study guide

- Do read the textbook!
- Attend class regularly, stay focused.
- Comprehend all exercises and homework.
- Old-school approach: pen and paper for doing exercise and taking notes.
- Experience in C/C++ will be useful.
- Code of conduct:
  - No web surfing, music, video, game in class.
  - Food is not allowed (water/soft drink OK).
- Final exam (and possibly mid-term) will be online quiz, with topics from exercises and homework.

Homework/exercises

- MIPS assembly programming
- MARS simulator



Chapter 1: Introduction

1. Computer Abstraction and Technology
2. Performance Evaluation

[with materials from *Computer Organization and Design, 5th Edition*, Patterson & Hennessy, ©2014, MK and M.J. Irwin's presentation, PSU 2008]

1. Computer Abstraction and Technology

- What is a computer?
- Computer classification
- Computer generations
- The key of computer evolution: IC making technology
- Computer organization

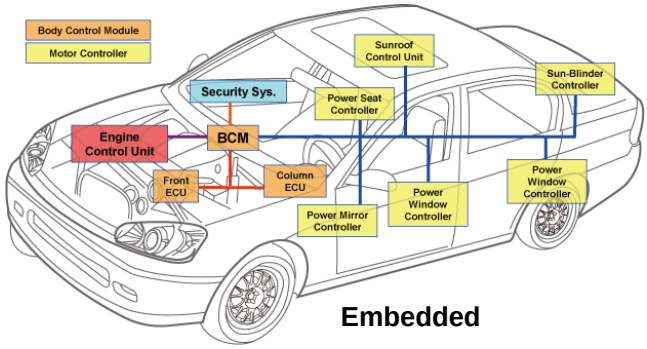
1. Computer Abstraction and Technology

- **What is a computer?**
- A machine that
  - Accepts input data
  - Processes data by executing a stored program
  - Produces output
- Which one is computer?

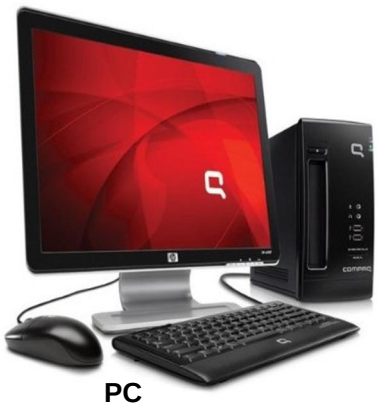
Classes of Computers

- Supercomputers
  - Super fast + expensive for high-end applications
- Server
  - Network based
  - High capacity, performance, reliability
  - Range from small servers to building sized
- Desktop computers
  - General purpose, variety of software
  - Subject to cost/performance tradeoff
- Embedded computers
  - Hidden as components of systems
  - Stringent power/performance/cost constraints

Dominant look and feel of computer classes



Embedded

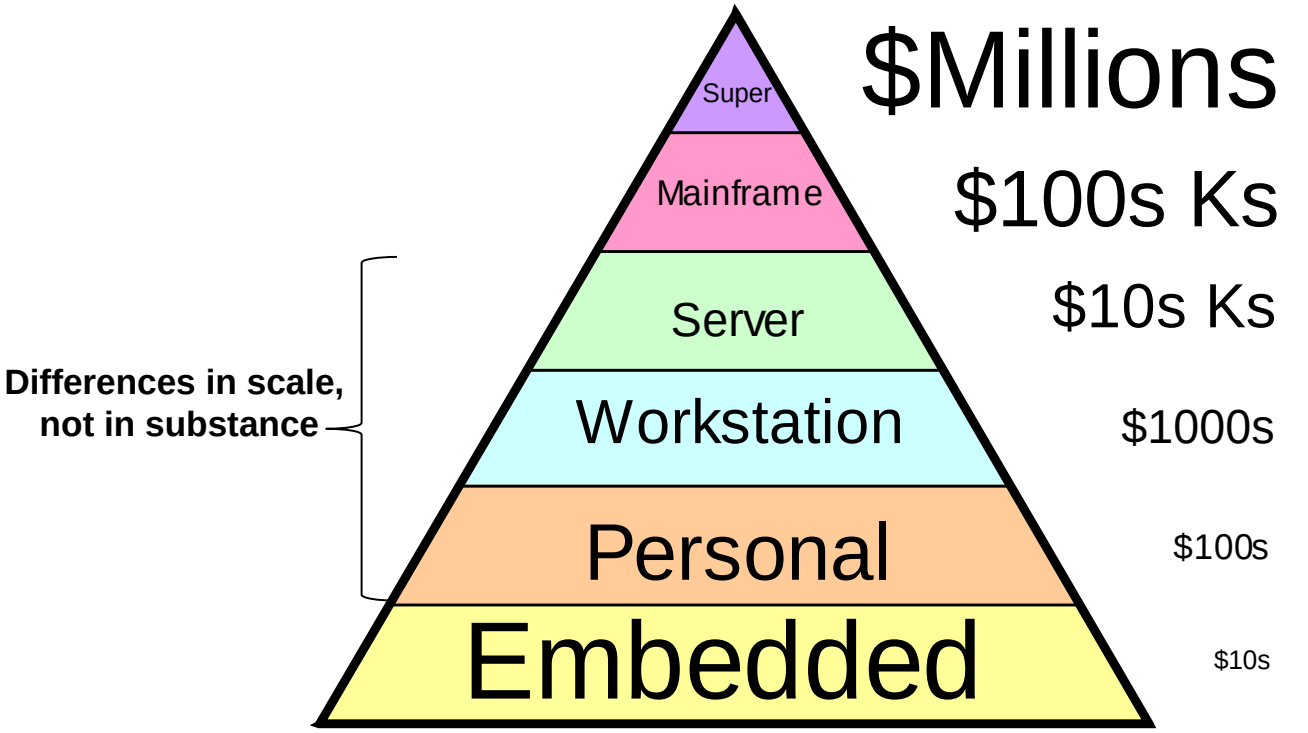


PC

Server

Super computer

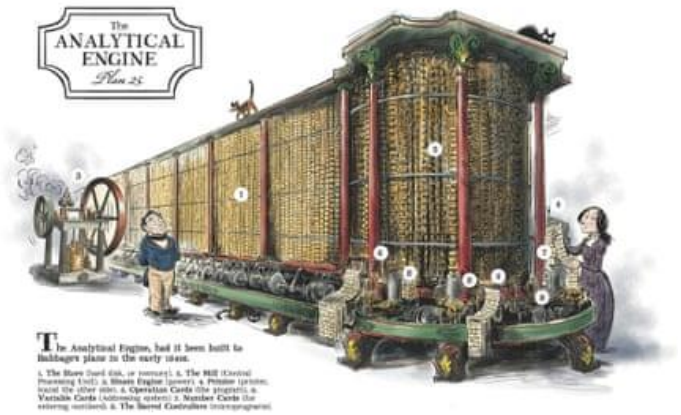
Price/performance of computer classes



A brief history of computers

- 0th generation: mechanical/analog calculators
  - Jacquard's punch card: for textile factories, later used for the first computers
  - Pascalite machine
  - Babage's Analytical Engine
  - Ada Lovelace: first computer program!!!

Pascalite machine



Babbage's Analytical Engine (plan 25)

Curiosity Stream - Calculating Ada: The Countess of Computing

A brief history of computers

- 1st generation: Vacuum tubes
  - ENIAC: 1st general purpose computer
    - Computing artillery-firing tables
    - Enormous in size and energy consumption
  - IAS: computer with Von Newman architecture
    - Memory, ALU, Control, Input/Output, stored-program concept
  - UNIVAC: 1st commercial computer

A brief history of computers

- 2nd generation: transistor
- Computer became smaller and faster

IBM System/360

A brief history of computers

- Later generations: IC and VLSI
- Increasing price/performance
- Moore's law

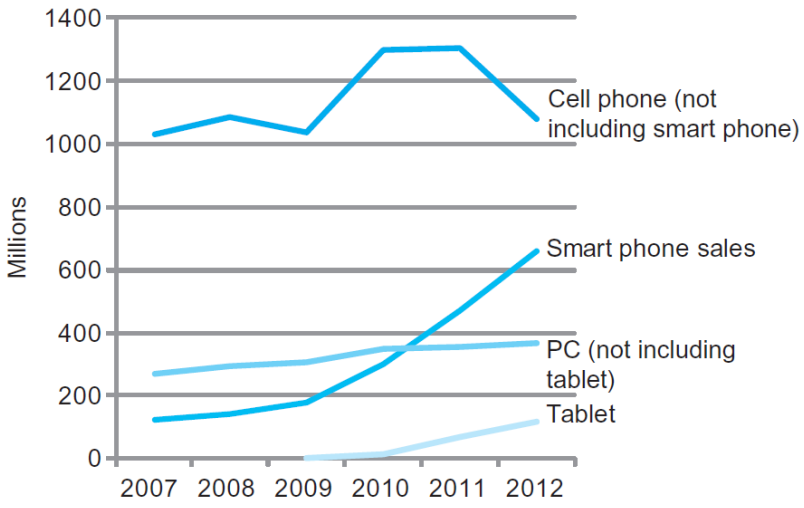
Table 1.2 Computer Generations

Generation	Approximate Dates	Technology	Typical Speed (operations per second)
1	1946–1957	Vacuum tube	40,000
2	1957–1964	Transistor	200,000
3	1965–1971	Small- and medium-scale integration	1,000,000
4	1972–1977	Large scale integration	10,000,000
5	1978–1991	Very large scale integration	100,000,000
6	1991–	Ultra large scale integration	>1,000,000,000

W.Stallings, COA, 10th edition

Post-PC era

- PDA, smart phone, tablet…
- Smart TV, set top box…
- Cloud computing (AMZ EC2, cloud gaming…)



The number manufactured per year of tablets and smart phones

Eight important ideas

Design for Moore's law

Simplification via abstraction

Make common cases fast

Performance via Parallelism

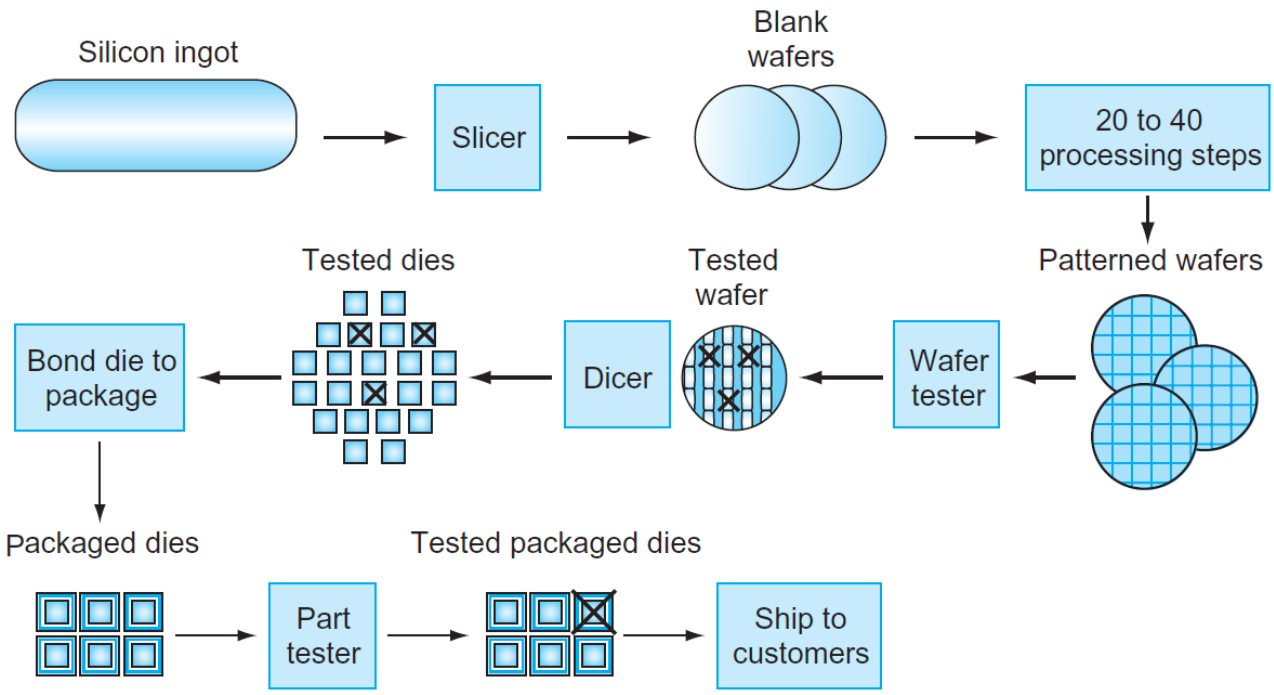
Performance via Pipelining

Performance via Prediction

Memory hierarchy

Dependability via redundancy

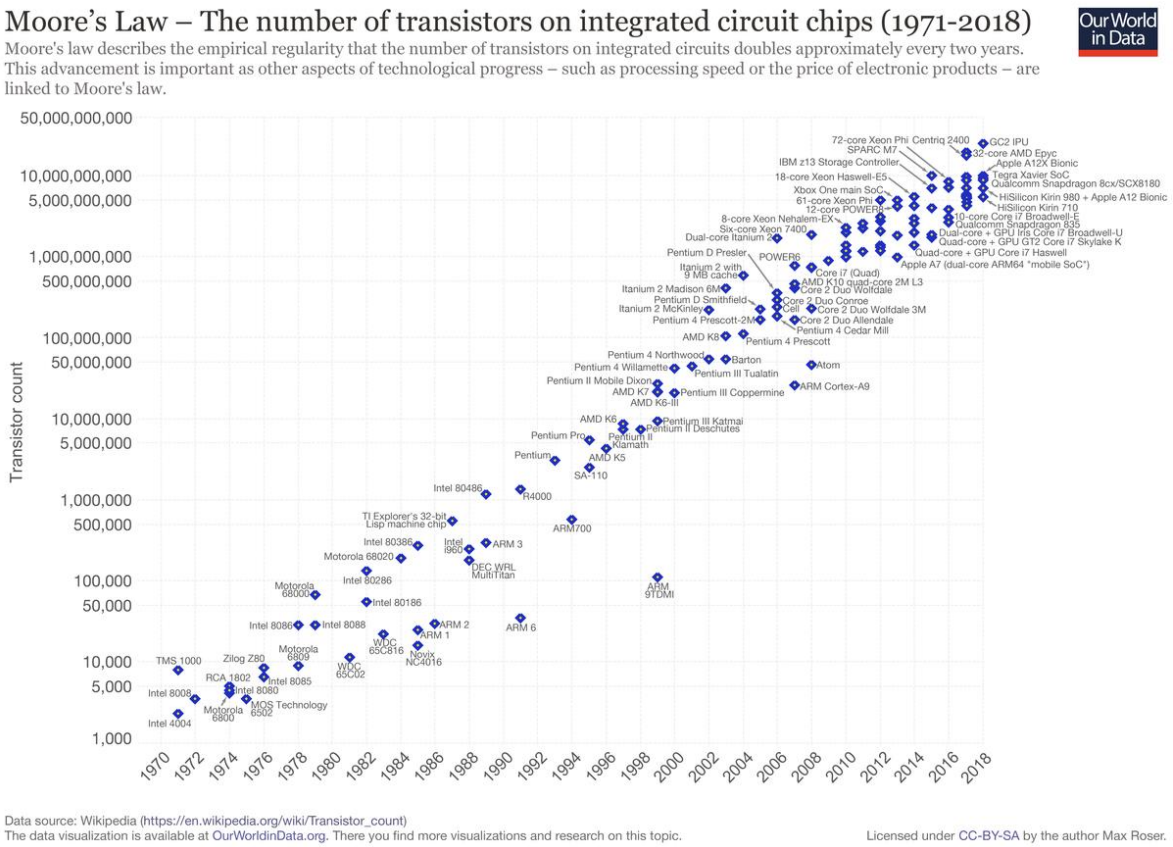
Key to computer evolution: IC making technology



The chip manufacturing process

Video: How an IC is made

Moore's Law



How do we benefit from this?

Key to computer evolution: IC making technology

- Electronics technology continues to evolve
  - Increased capacity and performance
  - Reduced cost

Year	Name	Size (cu. ft.)	Power (watts)	Performance (adds/sec)	Memory (KB)	Price	Price/ performance vs. UNIVAC	Adjusted price (2007 $)	Adjusted price/ performance vs. UNIVAC
1951	UNIVAC I	1,000	125,000	2,000	48	$1,000,000	1	$7,670,724	1
1964	IBM S/360 model 50	60	10,000	500,000	64	$1,000,000	263	$6,018,798	319
1965	PDP-8	8	500	330,000	4	$16,000	10,855	$94,685	13,367
1976	Cray-1	58	60,000	166,000,000	32,000	$4,000,000	21,842	$13,509,798	47,127
1981	IBM PC	1	150	240,000	256	$3,000	42,105	$6,859	134,208
1991	HP 9000/ model 750	2	500	50,000,000	16,384	$7,400	3,556,188	$11,807	16,241,889
1996	Intel PPro PC (200 MHz)	2	500	400,000,000	16,384	$4,400	47,846,890	$6,211	247,021,234
2003	Intel Pentium 4 PC (3.0 GHz)	2	500	6,000,000,000	262,144	$1,600	1,875,000,000	$2,009	11,451,750,000
2007	AMD Barcelona PC (2.5 GHz)	2	250	20,000,000,000	2,097,152	$800	12,500,000,000	$800	95,884,051,042

[Textbook]

What's below your program?

- 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;
}
```

one-to-many

C compiler

- Assembly language program (for MIPS CPU)

```
swap:   sll     $2, $5, 2
        add     $2, $4, $2
        lw      $15, 0($2)
        lw      $16, 4($2)
        sw      $16, 0($2)
        sw      $15, 4($2)
        jr      $31
```

one-to-one

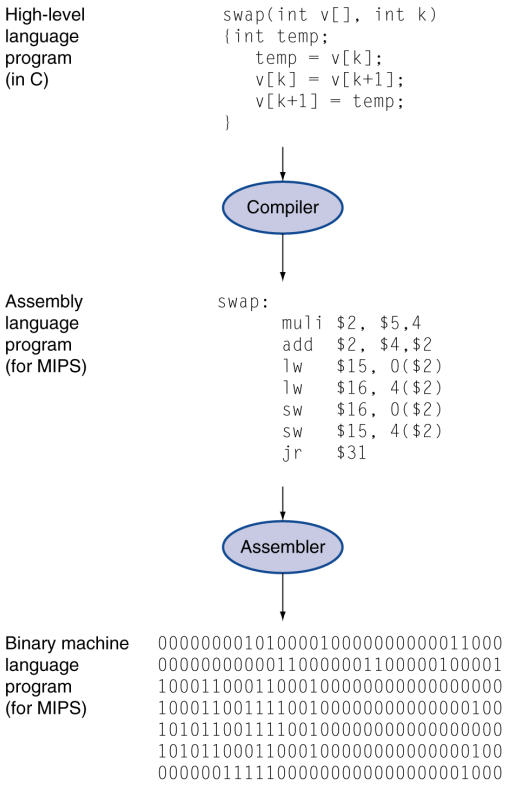
assembler

- Machine (object, binary) code (for MIPS CPU)

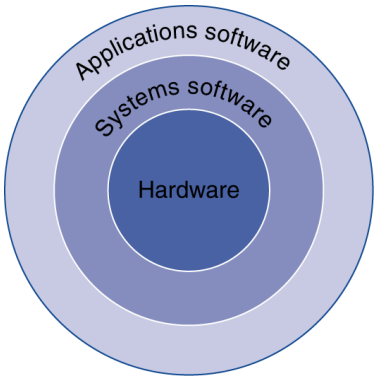
```
000000 00000 00101 0001000010000000
000000 00100 00010 0001000000100000
. . .
```

Levels of Program Code

- High-level language
  - Level of abstraction closer to problem domain
  - Provides for productivity and portability
- Assembly language
  - Textual representation of instructions
- Hardware representation
  - Binary digits (bits)
  - Encoded instructions and data



Hardware/software interface: below your program



- Application software
  - Written in high-level language (HLL)
- System software
  - Compiler: translates HLL code to machine code
  - Operating System: service code
    - Handling input/output
    - Managing memory and storage
    - Scheduling tasks & sharing resources
- Hardware
  - Processor, memory, I/O controllers

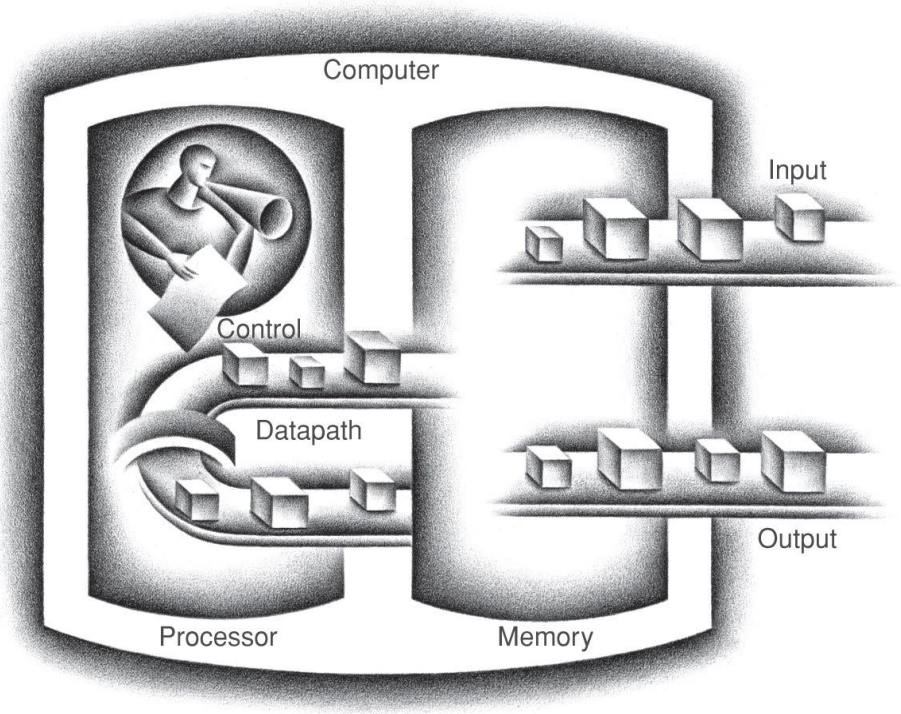
Computer Organization

- Computer's basic operation
  - Input data
  - Process data by executing stored program
  - Output data
- What are required components of computer?
  - For data input:
  - For storing information:
  - For program execution and data processing:
  - For data output:

Computer Organization

- Five classic components of a computer – input, output, memory, datapath, and control

- **datapath + control = processor (CPU)**



2. Computer performance evaluation

- What is performance?
- A storage system
  - How much time to find a file/object?
  - How much time to transfer a file?
  - How many files can be served simultaneously?
- A web server
  - How fast a request can be served?
  - How many request can be served per second?
- Different criteria to define performance
  - Throughput
  - **Response time**
- We focus on response time

2. Computer performance evaluation

- Response time:
  - System performance: elapsed time on unload system
  - CPU performance: user CPU time, the time that CPU actually spent on executing user program.
- To maximize performance, need to minimize execution time

$$\text{performance}_X = 1 / \text{execution\_time}_X$$

If computer X is n times faster than Y, then

$$\frac{\text{performance}_X}{\text{performance}_Y} = \frac{\text{execution\_time}_Y}{\text{execution\_time}_X} = n$$

Relative Performance Example

- If computer A runs a program in 10 seconds and computer B runs the same program in 15 seconds, how much faster is A than B?

We know that A is n times faster than B if

$$\frac{\text{performance}_A}{\text{performance}_B} = \frac{\text{execution\_time}_B}{\text{execution\_time}_A} = n$$

The performance ratio is $\frac{15}{10} = 1.5$

Assume performance of B is 1, then performance of A is 1.5

Performance Factors

- CPU execution time (CPU time) – time the CPU spends working on a task
  - Does not include time waiting for I/O or running other programs

$$\text{CPU execution time for a program} = \text{\# CPU clock cycles for a program} \times \text{clock cycle time}$$

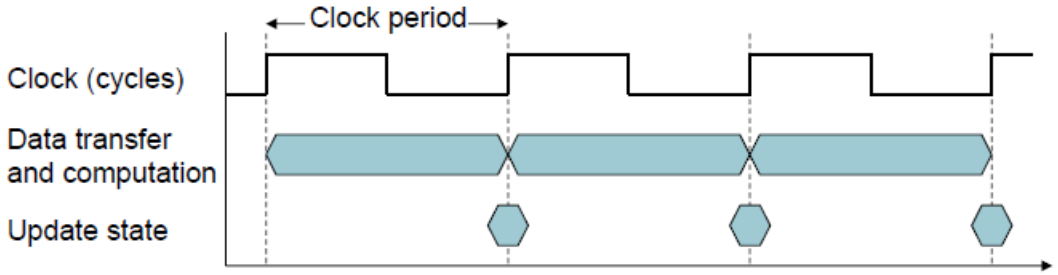
$$= \frac{\text{\# CPU clock cycles for a program}}{\text{clock rate}}$$

- Can improve performance by reducing either the length of the clock cycle or the number of clock cycles required for a program

Review: Machine Clock Rate

- Clock rate (clock cycles per second in MHz or GHz) is inverse of clock cycle time (clock period)

$$CC = 1 / CR$$



1 nsec (10^{-9}) clock cycle => 1 GHz (10^{9}) clock rate

500 psec clock cycle => 2 GHz clock rate

250 psec clock cycle => 4 GHz clock rate

200 psec clock cycle => 5 GHz clock rate

Improving Performance Example

- A program runs on computer A with a 2 GHz clock in 10 seconds. What clock rate must computer B run at to run this program in 6 seconds? Assume that, computer B will require 1.2 times as many clock cycles as computer A to run the program.

$$\text{CPU time}_A = \frac{\text{CPU clock cycles}_A}{\text{clock rate}_A}$$

$$\text{CPU clock cycles}_A = 10 \text{ sec} \times 2 \times 10^9 \text{ cycles/sec} = 20 \times 10^9 \text{ cycles}$$

$$\text{CPU time}_B = \frac{1.2 \times 20 \times 10^9 \text{ cycles}}{\text{clock rate}_B}$$

$$\text{clock rate}_B = \frac{1.2 \times 20 \times 10^9 \text{ cycles}}{6 \text{ seconds}} = 4 \text{ GHz}$$

Clock Cycles per Instruction

- Not all instructions take the same amount of time to execute
  - Average execution time ~ average clock cycles per instruction

 # CPU clock cycles for a program = # Instructions for a program x Average clock cycles per instruction

- **Clock cycles per instruction (CPI) – the average number of clock cycles each instruction takes to execute**
  - **A way to compare two different implementations of the same ISA**

	CPI for this instruction class		
	A	B	C
CPI	1	2	3

Using the Performance Equation

- Computers A and B implement the same ISA. Computer A has a clock cycle time of 250 ps and an effective CPI of 2.0 for some program and computer B has a clock cycle time of 500 ps and an effective CPI of 1.2 for the same program. Which computer is faster and by how much?

Each computer executes the same number of instructions, *I*, so

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

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

Clearly, A is faster … by the ratio of execution times

$$\frac{\text{performance}_A}{\text{performance}_B} = \frac{\text{execution\_time}_B}{\text{execution\_time}_A} = \frac{600 \times I \text{ ps}}{500 \times I \text{ ps}} = 1.2$$

The Performance Equation

- Our basic performance equation is then calculated

$$\text{CPU time} = \text{Instruction\_count} \times \text{CPI} \times \text{clock\_cycle}$$

$$= \frac{\text{Instruction\_count} \times \text{CPI}}{\text{clock\_rate}}$$

- Key factors that affect performance (CPU execution time)
  - The clock rate: CPU specification
  - CPI: varies by instruction type and ISA implementation
  - Instruction count: measure by using profilers/ simulators

Dynamic Instruction Count

How many instructions are executed in this program fragment?

Each "for" consists of two instructions: increment index, check exit condition

```
250 instructions
for i = 1, 100 do
20 instructions
   for j = 1, 100 do
   40 instructions
      for k = 1, 100 do
      10 instructions
      endfor
   endfor
endfor
```

Static count = 326

12,422,450 Instructions

2 + 20 + 124,200 instructions
100 iterations
12,422,200 instructions in all

2 + 40 + 1200 instructions
100 iterations
124,200 instructions in all

2 + 10 instructions
100 iterations
1200 instructions in all

```
for i = 1, n
while x > 0
```

Improving performance by CPI

Op	Freq	CPI_i	Freq x CPI_i
ALU	50%	1	
Load	20%	5	
Store	10%	3	
Branch	20%	2	
$Avg\ CPI = \sum freq_i * CPIi$			=

- How much faster would the machine be if a better data cache reduced the average load time to 2 cycles?
- What if branch instruction is only one cycle?
- What if two ALU instructions could be executed at once?

Improving performance by CPI

Op	Freq	CPI_i	Freq x CPI_i			
ALU	50%	1	.5	.5	.5	.25
Load	20%	5	1.0	.4	1.0	1.0
Store	10%	3	.3	.3	.3	.3
Branch	20%	2	.4	.4	.2	.4
$Avg\ CPI = \sum freq_i * CPIi$			= 2.2	1.6	2.0	1.95

- How much faster would the machine be if a better data cache reduced the average load time to 2 cycles?

  CPU time new = 1.6 x IC x CC so 2.2/1.6 means 37.5% faster
- What if branch instruction is only one cycle?

  CPU time new = 2.0 x IC x CC so 2.2/2.0 means 10% faster
- What if two ALU instructions could be executed at once?

  CPU time new = 1.95 x IC x CC so 2.2/1.95 means 12.8% faster

How to improve performance?

- Shorter clock cycle = faster clock rate

  → latest CPU technology

- Smaller CPI

  → optimizing Instruction Set Architecture

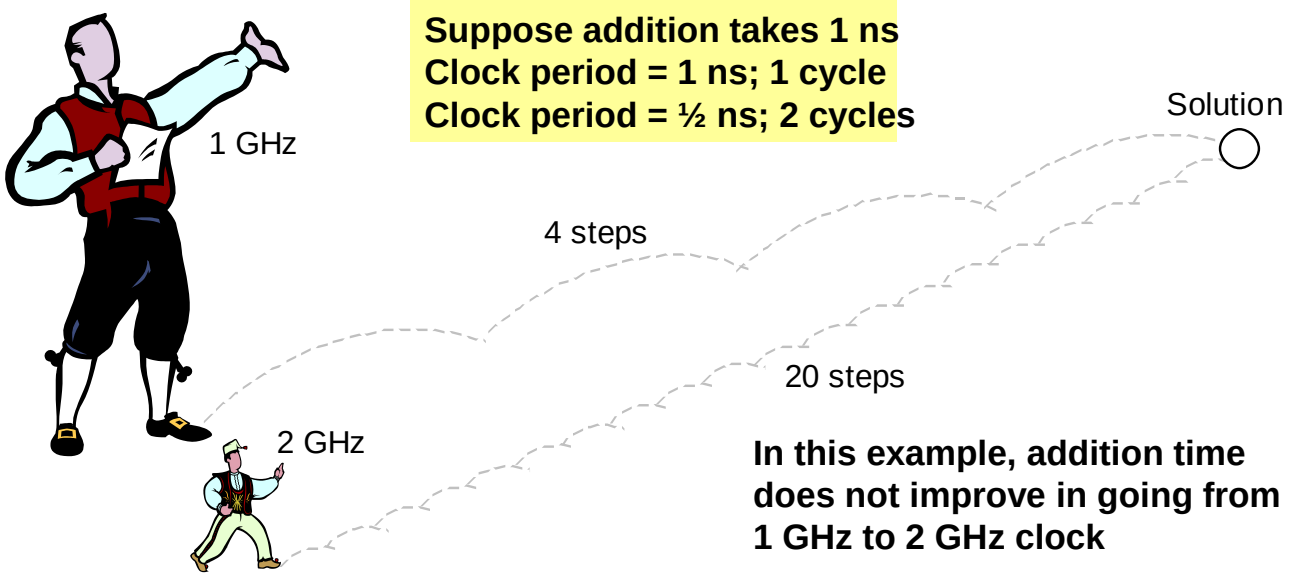
- Smaller instruction count

  → optimizing algorithm and compiler

- To get best performance, multiple criteria are combined and considered at design time

→ specific CPU for specific class computation problem

Faster Clock ≠ Shorter Running Time



Suppose addition takes 1 ns
Clock period = 1 ns; 1 cycle
Clock period = ½ ns; 2 cycles

In this example, addition time does not improve in going from 1 GHz to 2 GHz clock

Faster steps do not necessarily mean shorter travel time.

Measuring/benchmarking PC performance

- SPEC CPU benchmark
  - Started in 1989
  - SPEC CPU2006: 12 integer, 17 floating point benchmarks
  - Reference machine: Sun Ultra Enterprise 2 (1997) running on a 296 MHz UltraSPARC II CPU.

Description	Name	Instruction Count x 10^9	CPI	Clock cycle time (seconds x 10^{-9})	Execution Time (seconds)	Reference Time (seconds)	SPECratio
Interpreted string processing	perl	2252	0.60	0.376	508	9770	19.2
Block-sorting compression	bzip2	2390	0.70	0.376	629	9650	15.4
GNU C compiler	gcc	794	1.20	0.376	358	8050	22.5
Combinatorial optimization	mcf	221	2.66	0.376	221	9120	41.2
Go game (AI)	go	1274	1.10	0.376	527	10490	19.9
Search gene sequence	hmmer	2616	0.60	0.376	590	9330	15.8
Chess game (AI)	sjeng	1948	0.80	0.376	586	12100	20.7
Quantum computer simulation	libquantum	659	0.44	0.376	109	20720	190.0
Video compression	h264avc	3793	0.50	0.376	713	22130	31.0
Discrete event simulation library	omnetpp	367	2.10	0.376	290	6250	21.5
Games/path finding	astar	1250	1.00	0.376	470	7020	14.9
XML parsing	xalancbmk	1045	0.70	0.376	275	6900	25.1
Geometric mean	–	–	–	–	–	–	25.7

FIGURE 1.18 SPECINTC2006 benchmarks running on a 2.66 GHz Intel Core i7 920.

End of chapter 1

Computer Architecture

Ngo Lam Trung & Pham Ngoc Hung
Faculty of Computer Engineering
School of Information and Communication Technology (SoICT)
Hanoi University of Science and Technology
E-mail: [trungnl, hungpn]@soict.hust.edu.vn

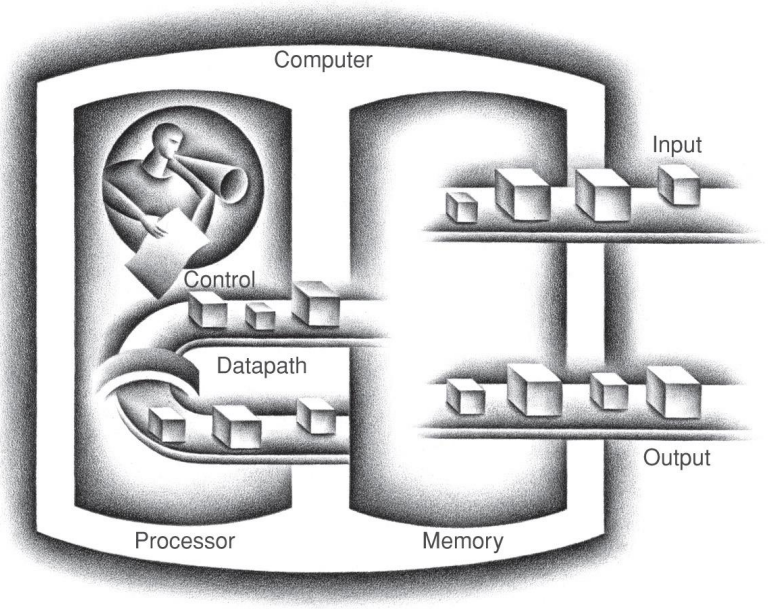
Chapter 2: Computer Functions and Interconnection

1. Computer Components
2. Computer Functions
3. Inter-connection Structures

[with materials from *Computer Organization and Architecture, 10th Edition*, William Stallings, ©2016, Pearson]

Computer Organization

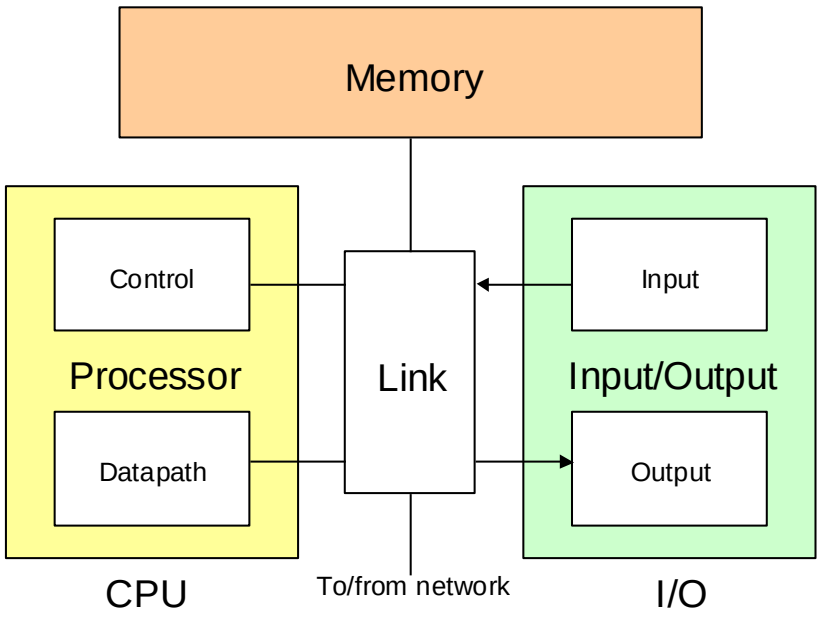
- From Chap.1: classic components of a computer



- Computer
  - Input data
  - Execute stored programs
  - Output result

1. Computer Components

- More detailed computer organization



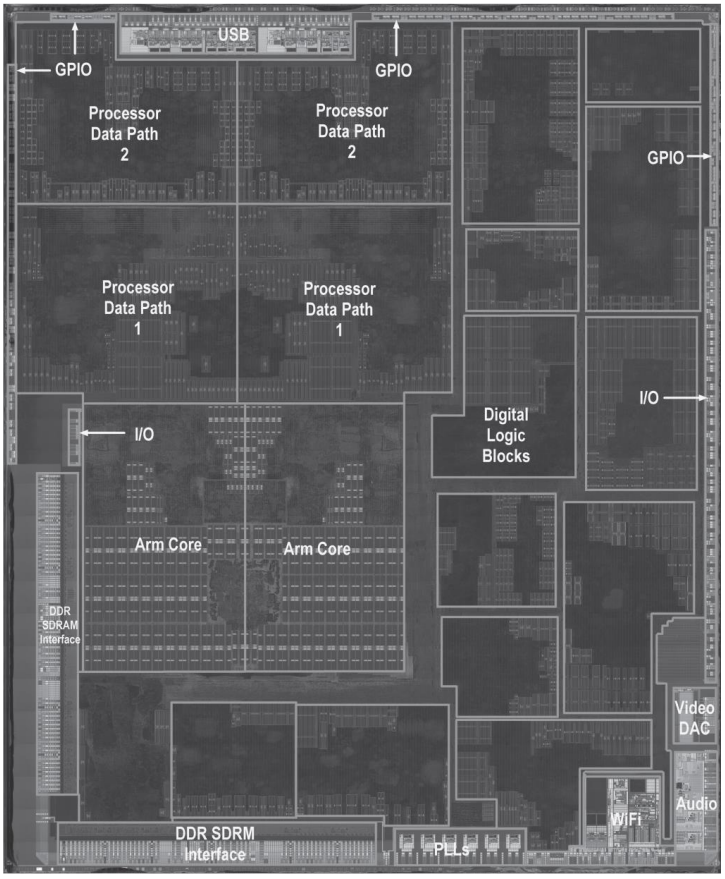
Computer organization with system link

CPU (Central Processing Unit)

- Control Unit
  - Fetch instruction from memory.
  - Interpret instruction.
  - Control other components to execute instruction.
- Datapath: performs arithmetic operations to process data.
- Register file (chapter 3): small and fast data storage for instruction execution.
- Some other dedicated components

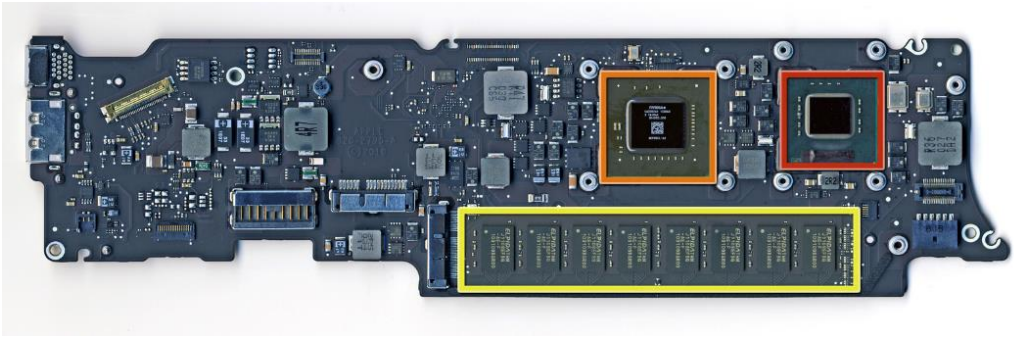
CPU

- Example: Apple A5



Memory

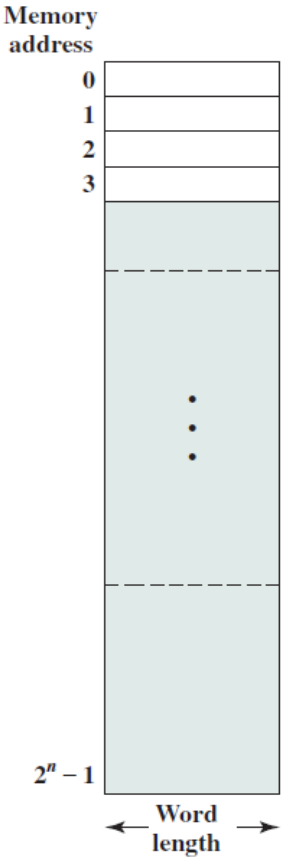
- Store instructions of the running programs.
- Store data that are currently in use.



Further reading: memory technologies

Memory

- Logical organization
  - Array of memory cells
  - Each cell holds one byte of data
  - Each cell is assigned an unique adress
  - Data value can be changed, address is fixed
- Data are stored on memory cells
  - 8-bit integer requires 1 cell
  - 32-bit integer requires 4 cells
  - Array requires consecutive cells according tc its size.
  - …

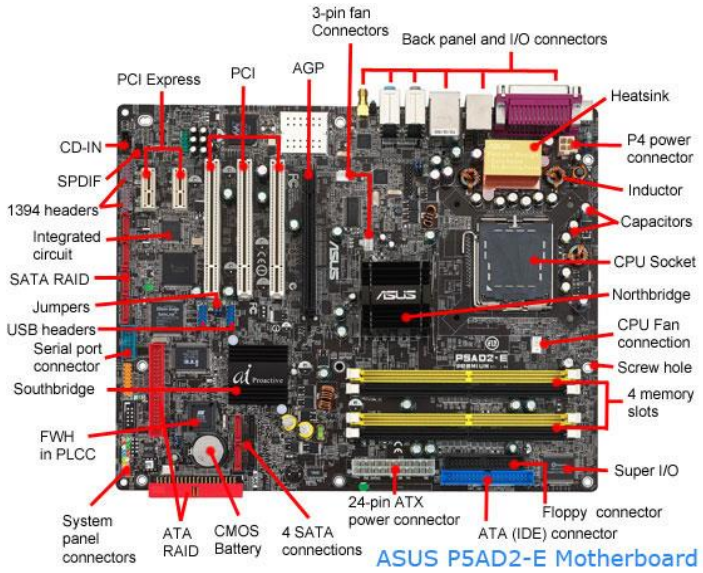


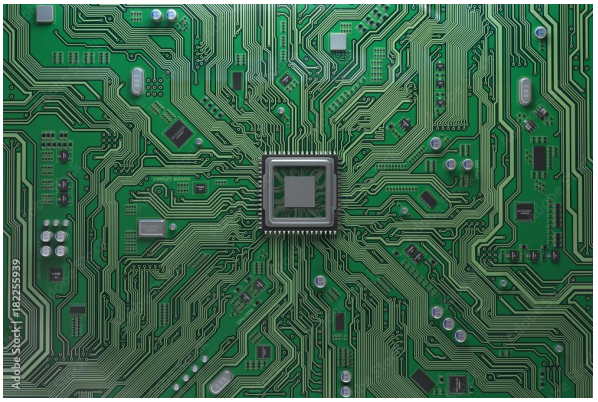
Input/output

- Interfacing computer with physical world/environment.
- Types of I/O device
  - Input: mouse, keyboard, webcam…
  - Output: display, printer, speaker…
  - Storage: HDD, SSD, optical, USB drives…
  - Communication: WiFi, Ethernet, Bluetooth modules…

Link: System interconnection

- The fabric to connect all components
- Huge number of connection, requires very good design so that all components function properly





2. Computer functions

- Executing program
- Interrupt
- Input/Output

2.1 Executing program

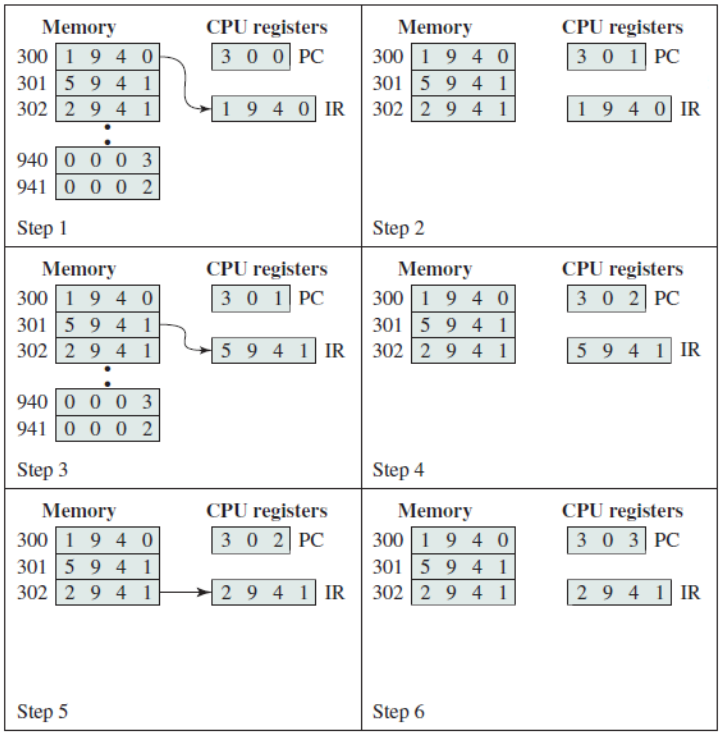
- ➔ the most basic function of computers.
- Program: a set of instructions.
- Computers execute instructions sequentially.
- Instruction cycle:
  - Instruction fetch: control unit fetches an instruction from memory
  - Instruction execution:
    - control unit decodes instruction,
    - then “tells” datapath and other components to perform the required action.
    - More details in Chapter 5.

Instruction fetch

- Importance
  - To get the correct instruction.
  - To execute all instructions in a program sequentially.
- At the beginning of each instruction cycle the processor fetches an instruction from memory.
- The program counter (PC) holds the address of the instruction to be fetched.
- The processor increases PC after each instruction fetch so that PC points to the next instruction in sequence.
- The fetched instruction is loaded into the instruction register (IR).

Instruction fetch

- Automatic increment of PC in fetch cycle

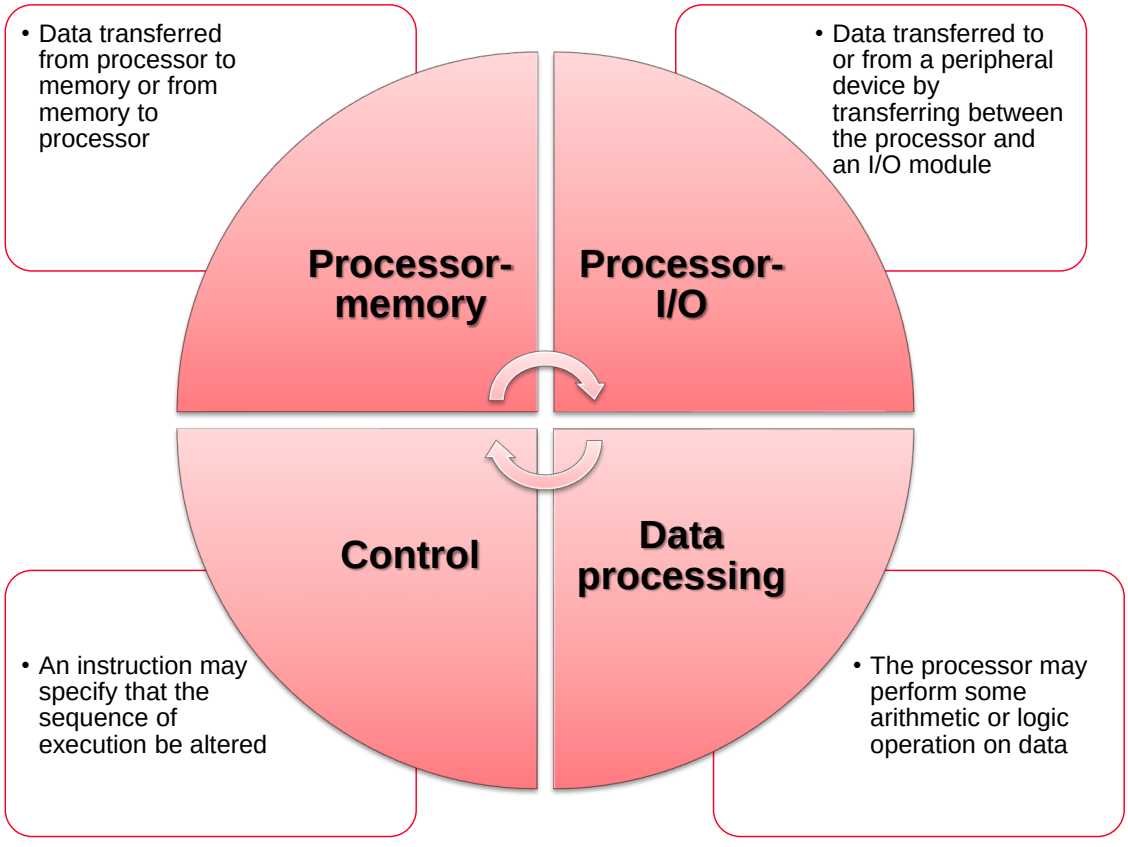


Elaboration: How to support branching?

Instruction execution

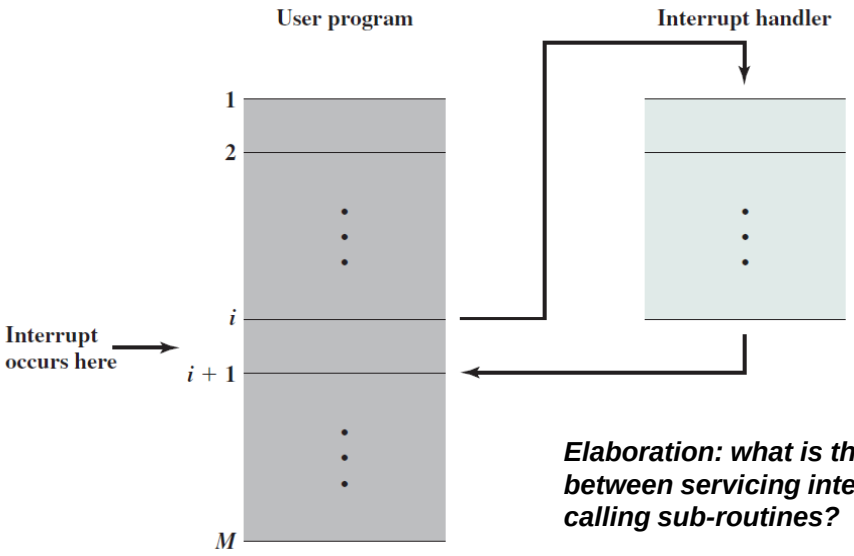
- Instruction (fetched and stored in IR) is decoded to get
  - The operation that the processor needs to do
  - The location to get input data (source operands)
  - The location to store output data (destination operand)
- Operand address calculation: calculate the address of operands
- Operand fetch: fetch source operands
- Data operation: perform the action on source operands and get result
- Operand store: store result into destination operand

Types of operation



2.2 Interrupt

- The mechanism to allow other components (memory, I/O) interrupt the normal processing of processor.
- Servicing interrupt: processor temporarily switch from the current program to execute a different (rather short) program, before continuing the original program.



Elaboration: what is the difference between servicing interrupt and calling sub-routines?

Sources of interrupt

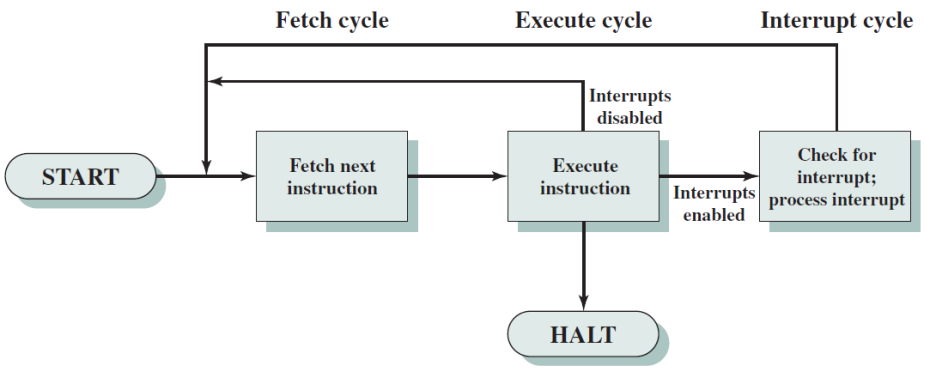
- Typical sources of interrupt:
  - Software/program: occurs during instruction execution upon some special condition such as division by 0, arithmetic overflow… Can also be called exception.
  - Timer: generated by system timer inside processor, to provide timing service, such as for operating system task scheduler service.
  - I/O: generated by I/O modules, to request service from processor or acknowledge the completion of an operation.
  - Hardware failure: generated when error happens with hardware.
- Example: detecting keyboard events (key up/key down)
  - Method 1: CPU checks keyboard status frequently
  - Method 2: keyboard issue interrupt to notify CPU upon key up/down
  - Which is better regarding CPU usage?

Interrupt handler/Interrupt service routine (ISR)

- Special programs to be executed to service interrupts.
- Usually a part of operating system or system software.
- Typical operation:
  - Determine the nature of interrupt: source and reason of interrupt.
  - Perform corresponding operation.
  - Return control to the interrupted program.
- Structure:
  - Interrupt handler ends with a special instruction, to restore context and value of PC, so that CPU can continue the interrupted program properly.

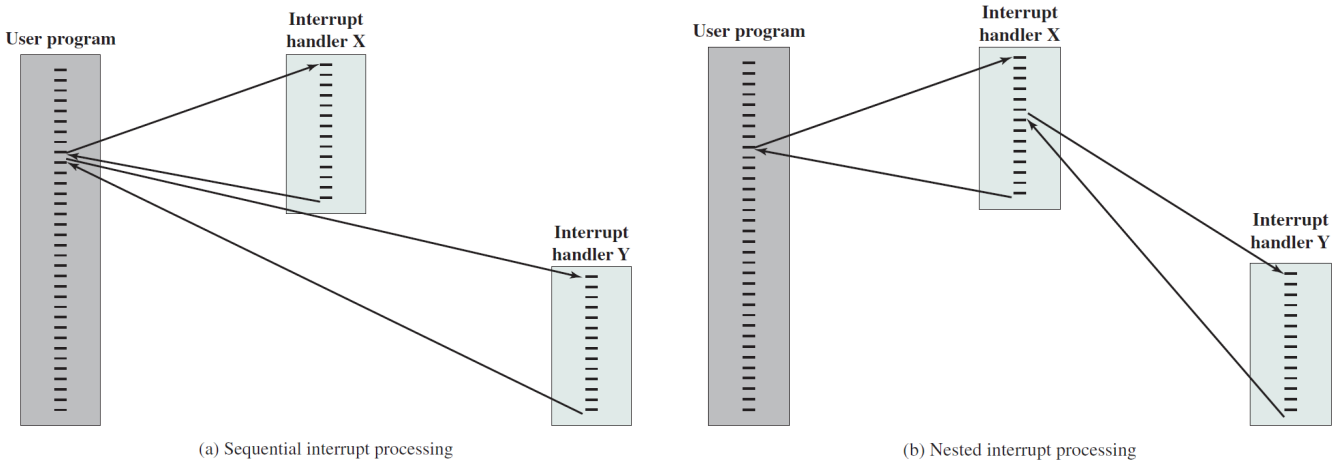
Servicing interrupts: instruction and interrupt cycle

- Interrupt is checked at the end of each instruction cycle
- Interrupt cycle if interrupt occurred:
  - CPU saves context of current program (current value of PC).
  - Address of interrupt handler is loaded to PC.
  - CPU continues with new instruction cycles, with new PC. Interrupt handler will be executed instead of original program.
  - At the end of interrupt handler, context will be restored including PC value. CPU return to the interrupted program.



Multiple interrupt procesing

- Number of interrupt sources is (always) high.
- Interrupts can occur at the same time or overlap.
- Sequential vs nested interrupt processing
  - Usually priority-based.
  - More details in chapter 7.

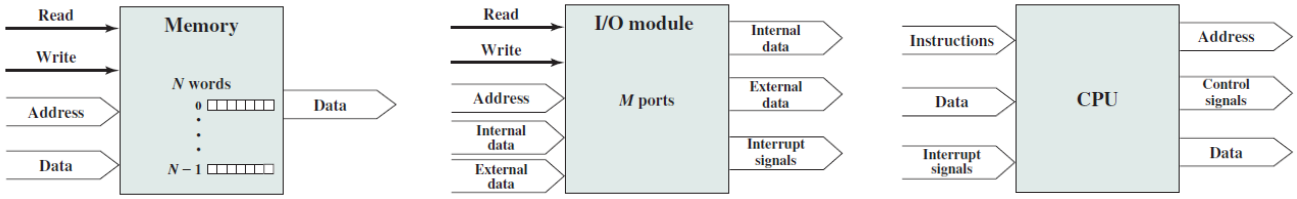


2.3 Input/output

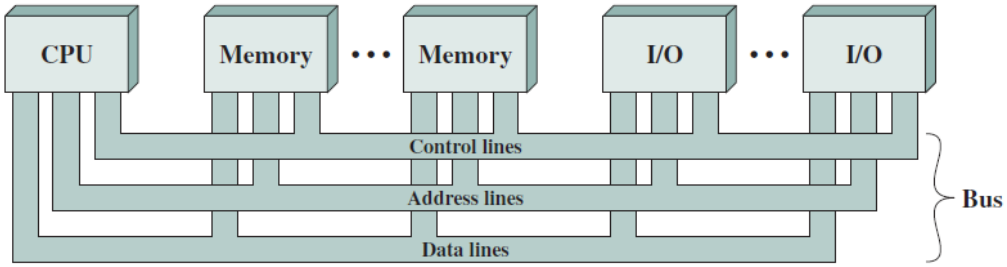
- The operation when data is transferred between I/O modules and CPU/memory.
- CPU-controlled data transfer: data is transferred between CPU and I/O, under the control of CPU.
- Direct memory access: data is transferred between memory and I/O, under the control of special controllers called DMAC.

3. System interconnection

- Interconnection model of each component
  - ➔ Need to connect these components

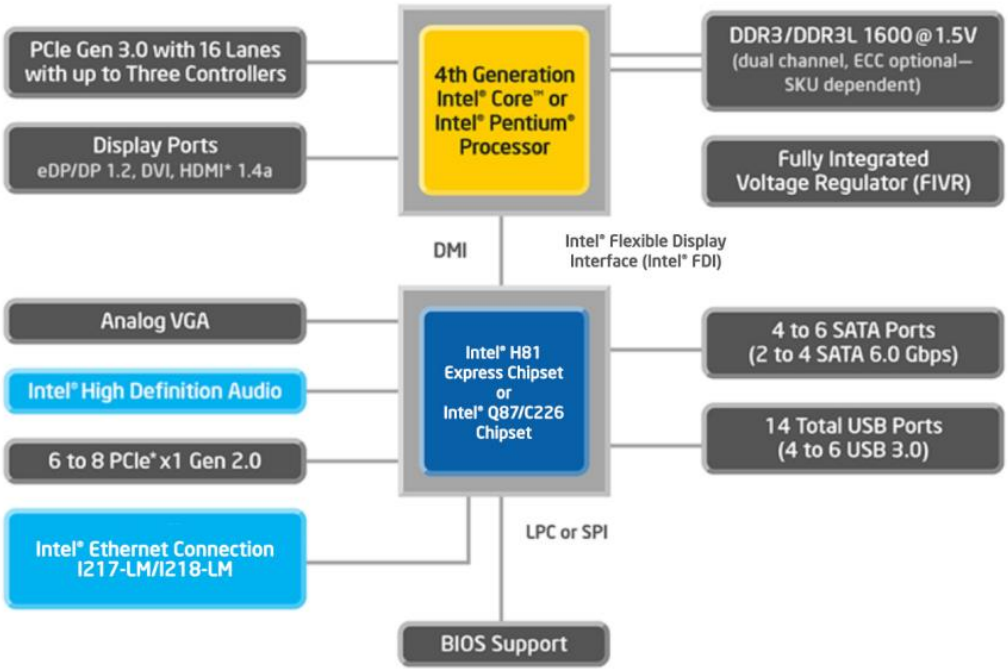


- Theoretical bus interconnection scheme



System interconnection

- Interconnection system for high performance computers: **hierarchical bus**



End of chapter 2

Computer Architecture

Ngo Lam Trung & Pham Ngoc Hung
Faculty of Computer Engineering
School of Information and Communication Technology (SoICT)
Hanoi University of Science and Technology
E-mail: [trungnl, hungpn]@soict.hust.edu.vn

Chapter 3: Arithmetic for Computers

[with materials from *Computer Organization and Design, 4th Edition*,
Patterson & Hennessy, © 2008, MK
and M.J. Irwin's presentation, PSU 2008]

Content

- (Super) Basics of logic design
- Integer representation and arithmetic
- Floating point number representation and arithmetic

What are stored inside computer?

- Data, of course!
- Data is represented as binary numbers.
- How binary numbers are treated by CPUs?
  - By logic circuits
  - Integers
    - Unsigned
    - Signed
  - Floating point numbers
    - Single precision
    - Double precision
    - Other formats

Basics of logic design (Appendix B)

- Boolean logic: logic variable and operators
- Logic variable: values of 1 (TRUE) or 0 (FALSE)
- Basic operators: AND, OR, NOT
  - A AND B : $A \cdot B$ hay AB
  - A OR B : $A + B$
  - NOT A : $\overline{A}$
  - Order: NOT > AND > OR
- Additional operators: NAND, NOR, XOR
  - A NAND B: $\overline{A \cdot B}$
  - A NOR B : $\overline{A + B}$
  - A XOR B: $A \oplus B = A \bullet \overline{B} + \overline{A} \bullet B$

Truth tables

A	B	A AND B A•B
0	0	0
0	1	0
1	0	0
1	1	1

A	B	A OR B A + B
0	0	0
0	1	1
1	0	1
1	1	1

A	NOT A $\overline{A}$
0	1
1	0

Unary operator NOT

A	B	A NAND B $\overline{A \cdot B}$
0	0	1
0	1	1
1	0	1
1	1	0

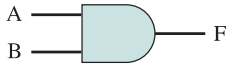
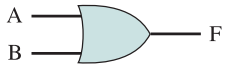
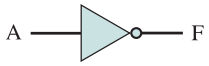
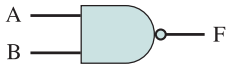
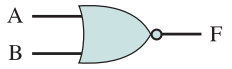
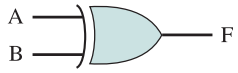
A	B	A XOR B $A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

A	B	A NOR B $\overline{A + B}$
0	0	1
0	1	0
1	0	0
1	1	0

Laws of Boolean algebra

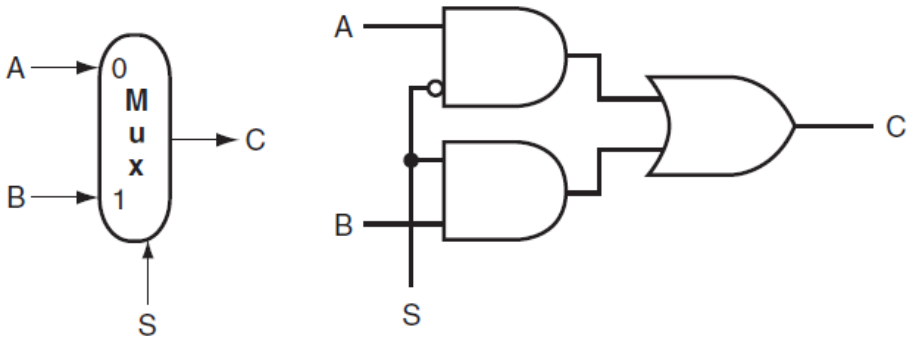
$A \cdot B = B \cdot A$	$A + B = B + A$
$A \cdot (B + C) = (A \cdot B) + (A \cdot C)$	$A + (B \cdot C) = (A + B) \cdot (A + C)$
$1 \cdot A = A$	$0 + A = A$
$A \cdot \overline{A} = 0$	$A + \overline{A} = 1$
$0 \cdot A = 0$	$1 + A = 1$
$A \cdot A = A$	$A + A = A$
$A \cdot (B \cdot C) = (A \cdot B) \cdot C$	$A + (B + C) = (A + B) + C$
$\overline{A \cdot B} = \overline{A} + \overline{B}$ (DeMorgan's law)	$\overline{A + B} = \overline{A} \cdot \overline{B}$ (DeMorgan's law)

Logic gates

Name	Graphical Symbol	Algebraic Function	Truth Table
AND		$F = A \cdot B$ or $F = AB$	A B | F 0 0 | 0 0 1 | 0 1 0 | 0 1 1 | 1
OR		$F = A + B$	A B | F 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 1
NOT		$F = \overline{A}$ or $F = A'$	A | F 0 | 1 1 | 0
NAND		$F = \overline{AB}$	A B | F 0 0 | 1 0 1 | 1 1 0 | 1 1 1 | 0
NOR		$F = \overline{A + B}$	A B | F 0 0 | 1 0 1 | 0 1 0 | 0 1 1 | 0
XOR		$F = A \oplus B$	A B | F 0 0 | 0 0 1 | 1 1 0 | 1 1 1 | 0

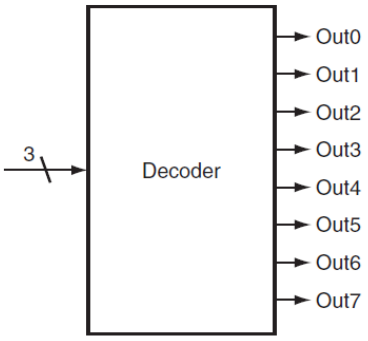
Example: multiplexor

- Depending on S, output C is equal to one of the two inputs A, B
- Explain how this circuit works?



Example: 3-to-8 decoder

- Very important in address decoder circuits



a. A 3-bit decoder

Inputs			Outputs							
12	11	10	Out7	Out6	Out5	Out4	Out3	Out2	Out1	Out0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

b. The truth table for a 3-bit decoder

Unsigned Binary Integers

- Using n-bit binary number to represent non-negative integer

$$X = X_{n-1}X_{n-2}\ldots X_1X_0$$
$$= X_{n-1}2^{n-1} + X_{n-2}2^{n-2} + \cdots + X_1 2^1 + X_0 2^0$$

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

  $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
  $= 0 + \ldots + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$
  $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$

- **Data range using 32 bits**

  0 to 2^{32}-1 = 4,294,967,295

Eg: 32 bit Unsigned Binary Integers

Hex	Binary	Decimal
0x00000000	0…0000	0
0x00000001	0…0001	1
0x00000002	0…0010	2
0x00000003	0…0011	3
0x00000004	0…0100	4
0x00000005	0…0101	5
0x00000006	0…0110	6
0x00000007	0…0111	7
0x00000008	0…1000	8
0x00000009	0…1001	9
	…	
0xFFFFFFFC	1…1100	2^{32}-4
0xFFFFFFFD	1…1101	2^{32}-3
0xFFFFFFFE	1…1110	2^{32}-2
0xFFFFFFFF	1…1111	2^{32}-1

Exercise

- Convert to 32-bit integers

  25 = 0000 0000 0000 0000 0000 0000 0001 1001

  125 = 0000 0000 0000 0000 0000 0000 0111 1101

  255 = 0000 0000 0000 0000 0000 0000 1111 1111

- Convert 32-bit integers to decimal

  0000 0000 0000 0000 0000 0000 1100 1111 = 207

  0000 0000 0000 0000 0000 0001 0011 0011 = 307

Signed binary integers

- Using n-bit binary number to represent integer, including negative values

$$x = x_{n-1}x_{n-2}\ldots x_1x_0$$
$$= -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_12^1 + x_02^0$$

- **Range: -2^{n-1} to $+2^{n-1} - 1$**
- **Example**

  $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
  $= -1\times2^{31} + 1\times2^{30} + \ldots + 1\times2^2 + 0\times2^1 + 0\times2^0$
  $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$

- **Using 32 bits**

  –2,147,483,648 to +2,147,483,647

Signed integer negation

- Given $x = xn\_1x_{n\_2}\ldots x_1x_0$, how to calculate $-x$?
- Let $\bar{x} = 1's\ complement\ of\ x$

$$\bar{x} = 1111\ldots11_2 - x$$

(1 → 0, 0 → 1)

Then

$$\bar{x} + x = 1111\ldots112 = -1$$

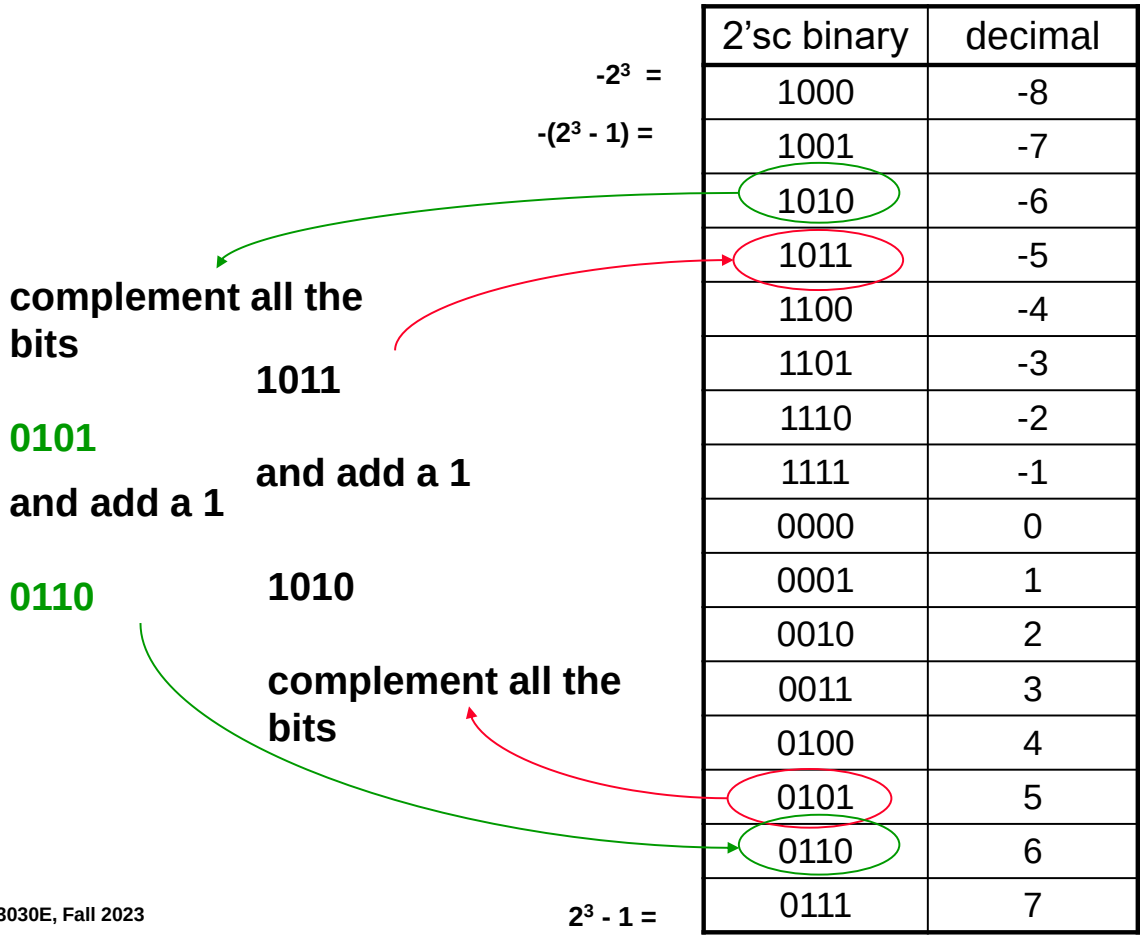
➔ $$\bar{x} + 1 = -x$$

- **Example: find binary representation of -2**

  +2 = 0000 0000 … 0010_2

  –2 = 1111 1111 … 1101_2 + 1
  = 1111 1111 … 1110_2

Signed binary negation



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

Exercise

- Find 16 bit signed integer representation of

  16 = 0000 0000 0001 0000

  -16 = 1111 1111 1111 0000

  100 = 0000 0000 0110 0100

  -100 = 1111 1111 1001 1100

Sign extension

- Given n-bit integer $x = xn\_1x_{n\_2}\ldots x_1x_0$
- Find corresponding m-bit representation (m > n) with the same numeric value

$$x = xm\_1x_{m\_2}\ldots x_1x_0$$

- → Replicate the sign bit to the left
- Examples: 8-bit to 16-bit

  +2: 0000 0010 => 0000 0000 0000 0010

  –2: 1111 1110 => 1111 1111 1111 1110

Addition and subtraction

- Addition
  - Similar to what you do to add two numbers manually
  - Digits are added bit by bit from right to left
  - Carries passed to the next digit to the left
- Subtraction
  - Negate the second operand then add to the first operand

$$\begin{array}{rll} & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0111_{two} & = 7_{ten} \\ + & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0110_{two} & = 6_{ten} \\ \hline & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1101_{two} & = 13_{ten} \end{array}$$

Examples

- All numbers are 8-bit signed integer

```
  12 + 8 =

 122 + 8 =

122 + 80 =
```

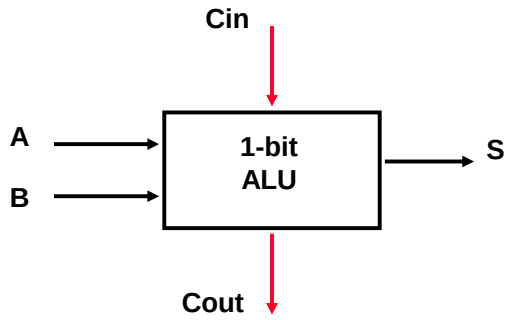

Dealing with Overflow

- **Overflow occurs when the result of an operation cannot be represented in 32-bits, i.e., when the sign bit contains a value bit of the result and not the proper sign bit**
  - **When adding operands with different signs or when subtracting operands with the same sign, overflow can *never* occur**

Operation	Operand A	Operand B	Result indicating overflow
A + B	≥ 0	≥ 0	< 0
A + B	< 0	< 0	≥ 0
A - B	≥ 0	< 0	< 0
A - B	< 0	≥ 0	≥ 0

Adder implementation

- 1-bit full adder

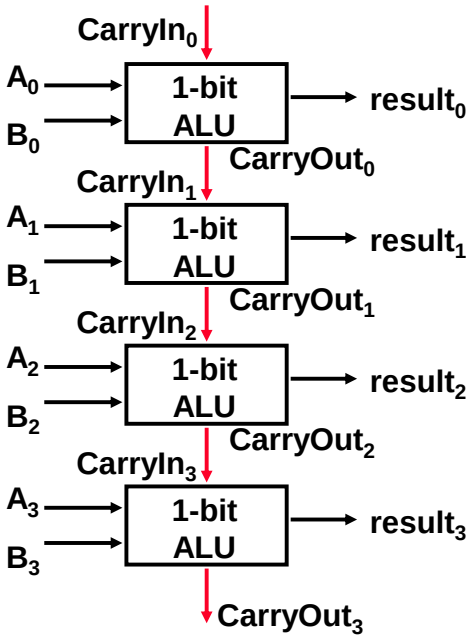


Inputs			Outputs	
A	B	Cin	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

- $S = Cin \oplus (A \oplus B)$
- $Cout = AB + BCin + ACin$

Adder implementation

- N-bit ripple-carry adder



Performance depends on data length

➔ Performance is low

Carry lookahead

- With 4-bit adder

$$c1 = g0 + (p0 \cdot c0)$$

$$c2 = g1 + (p1 \cdot g0) + (p1 \cdot p0 \cdot c0)$$

$$c3 = g2 + (p2 \cdot g1) + (p2 \cdot p1 \cdot g0) + (p2 \cdot p1 \cdot p0 \cdot c0)$$

$$c4 = g3 + (p3 \cdot g2) + (p3 \cdot p2 \cdot g1) + (p3 \cdot p2 \cdot p1 \cdot g0) + (p3 \cdot p2 \cdot p1 \cdot p0 \cdot c0)$$

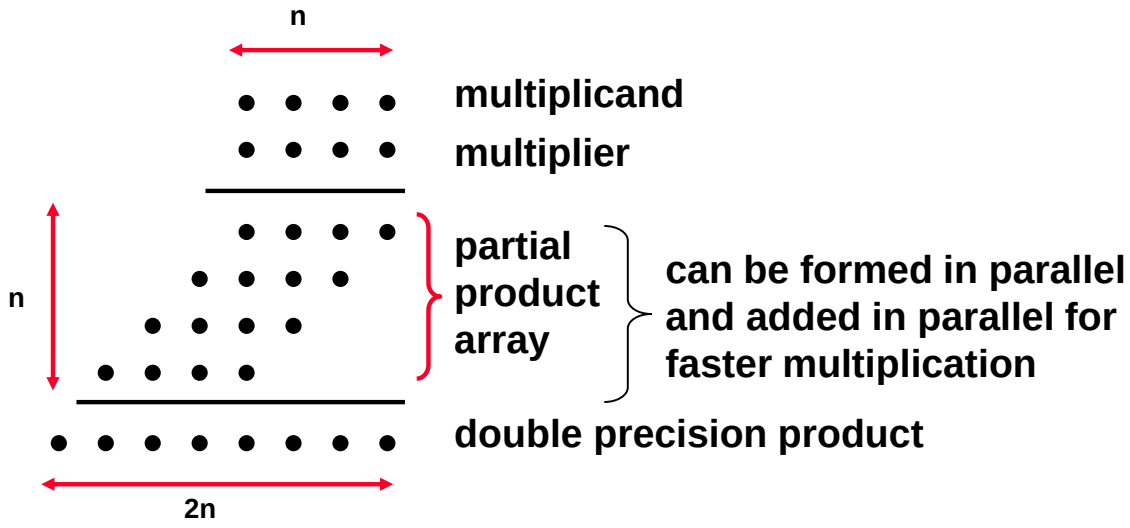
➔ All carry bits can be calculated after 3 gate delay

➔ All result bits can be calculated after maximum of 4 gate delay

➔ How to implement bigger adder?

Multiply

- Binary multiplication is just a *bunch* of right shifts and adds

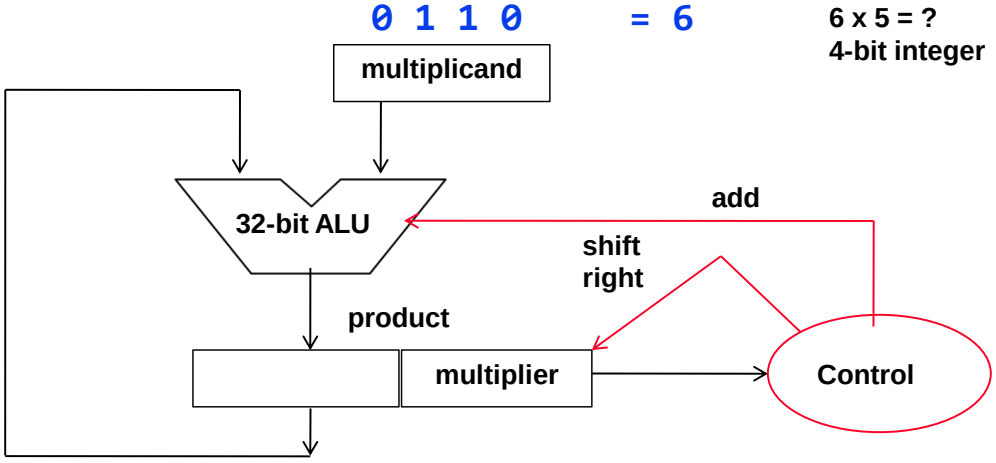


n-bit multiplicand and multiplier → 2n-bit product

Example

Multiplicand		1000_{ten}
Multiplier	x	1001_{ten}
		1000
		0000
		0000
		1000
Product		1001000_{ten}

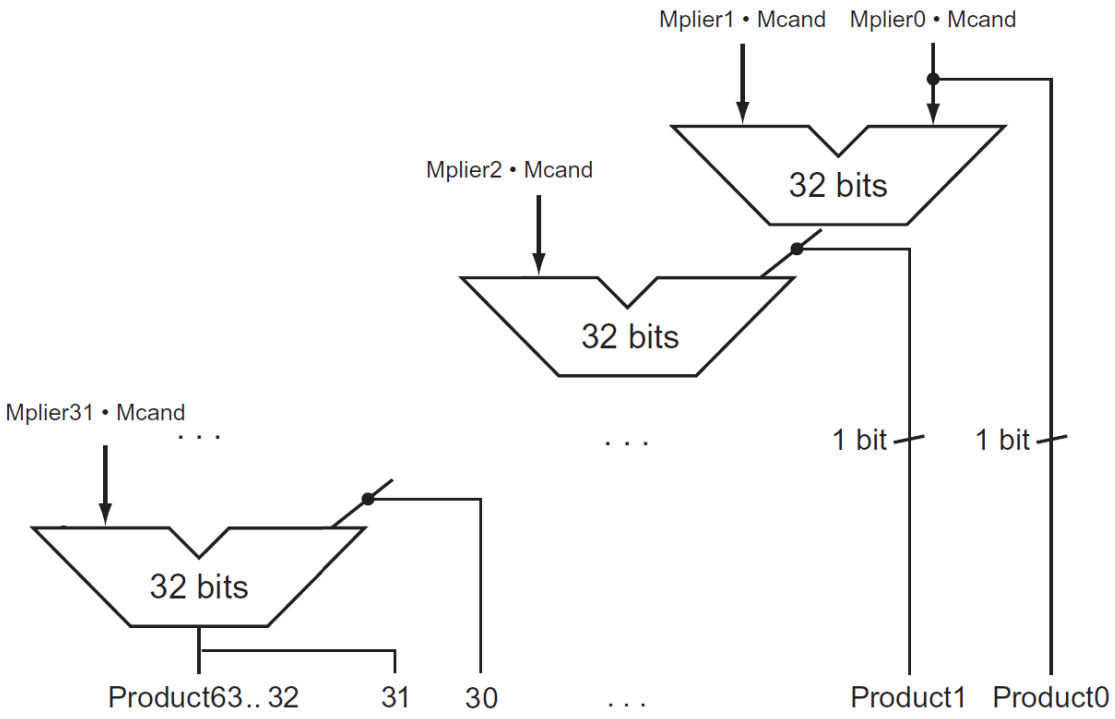
Add and Right Shift Multiplier Hardware



	0 0 0 0 0 1 0 1	= 5
add	0 1 1 0 0 1 0 1	LSB=1 → add multiplicand
	0 0 1 1 → 0 0 1 0	shift right
add	0 0 1 1 0 0 1 0	LSB=0 → no change
	0 0 0 1 → 1 0 0 1	shift right
add	0 1 1 1 1 0 0 1	LSB=1 → add multiplicand
	0 0 1 1 → 1 1 0 0	shift right
add	0 0 1 1 1 1 0 0	LSB=0 → no change
	0 0 0 1 → 1 1 1 0	shift right = 30

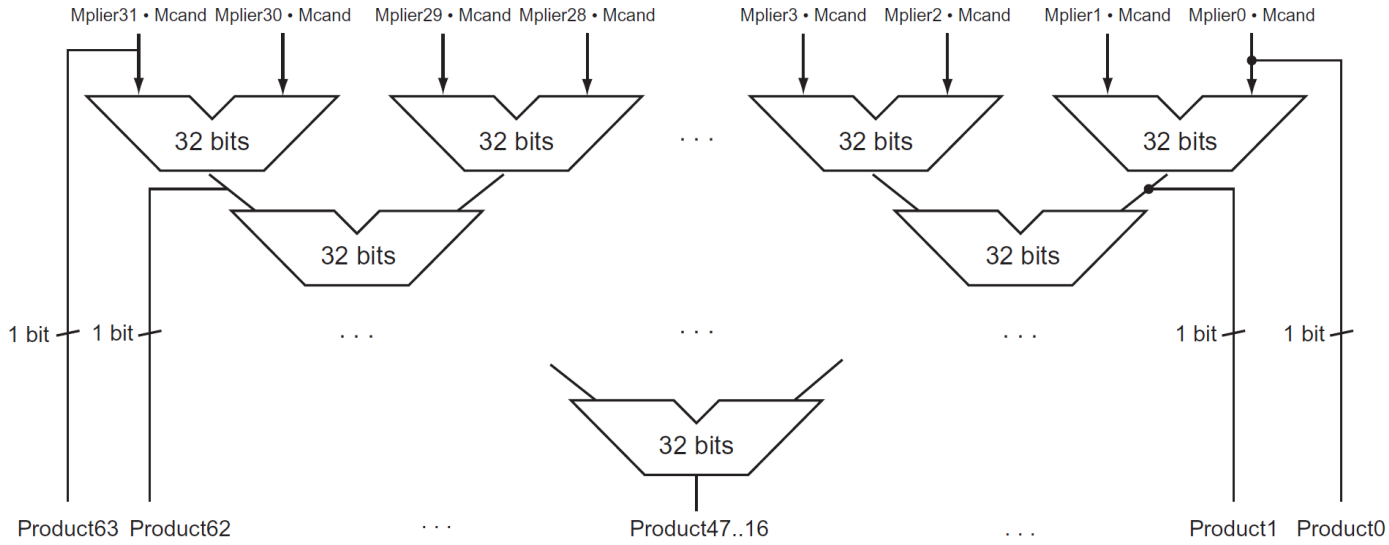
Fast multiplier – Design for Moore

- Why is this fast?



Fast multiplier – Design for Moore

- How fast is this?
- Anything wrong?



MIPS Multiply Instruction

- Multiply (`mult` and `multu`) produces a double precision product (2 x 32 bit)

```
mult   $s0, $s1      # hi||lo = $s0 * $s1
```

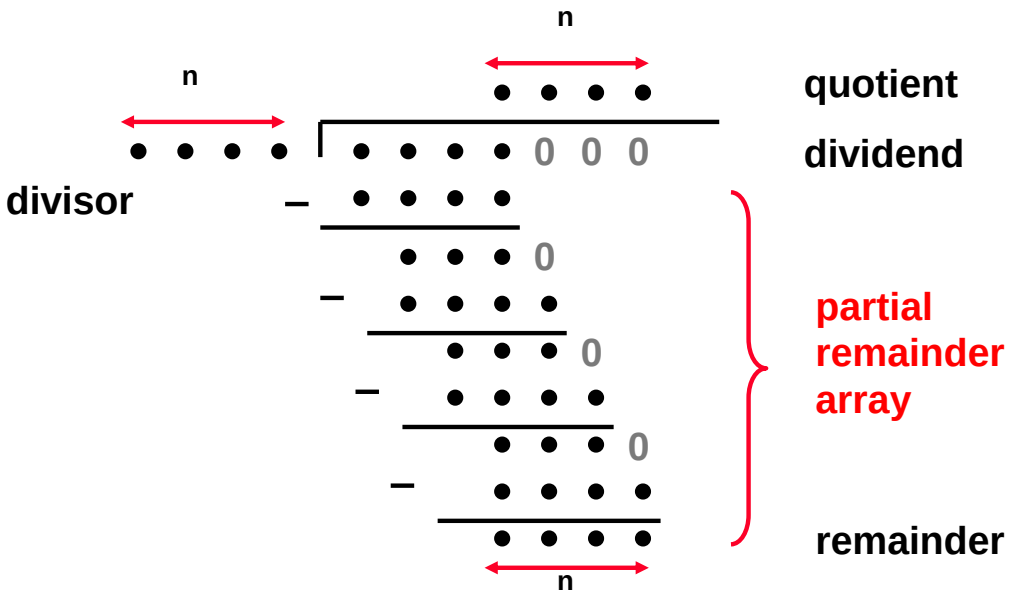
0	16	17	0	0	0x18	

  - Two additional registers: **hi** and **lo**
  - Low-order word of the product is stored in processor register `lo` and the high-order word is stored in register `hi`
  - Instructions `mfhi rd` and `mflo rd` are provided to move the product to (user accessible) registers in the register file

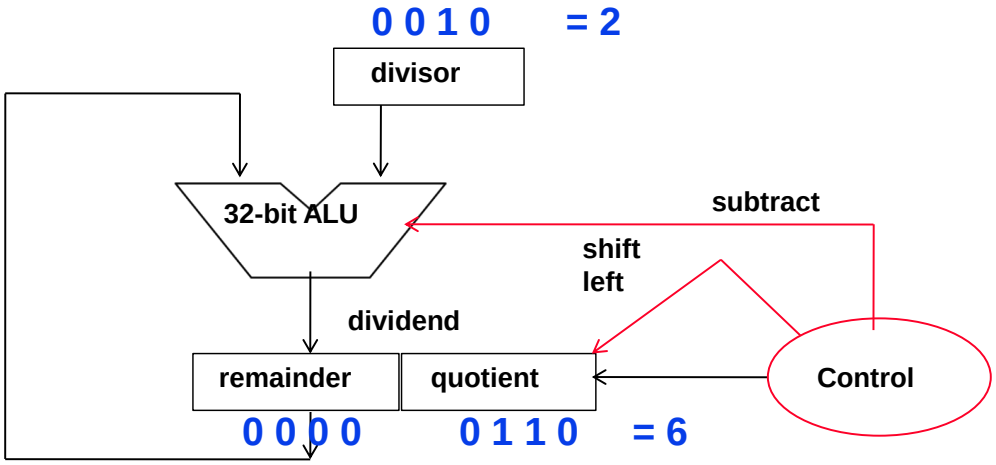
Division

- Division is just a *bunch* of quotient digit guesses and left shifts and subtracts

dividend = quotient x divisor + remainder



Left Shift and Subtract Division Hardware



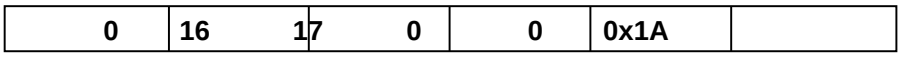
0 0 0 0 1 1 0 0
sub 1 1 1 0 1 1 0 0 rem neg, so 'ient bit = 0
0 0 0 0 1 1 0 0 restore remainder
0 0 0 1 1 0 0 0
sub 1 1 1 1 1 0 0 0 rem neg, so 'ient bit = 0
0 0 0 1 1 0 0 0 restore remainder
0 0 1 1 0 0 0 0
sub 0 0 0 1 0 0 0 1 rem pos, so 'ient bit = 1
0 0 1 0 0 0 1 0
sub 0 0 0 0 0 0 1 1 rem pos, so 'ient bit = 1

= 3 with 0 remainder

Divide Instruction

- Divide (`div` and `divu`) generates the reminder in `hi` and the quotient in `lo`

```
div    $s0, $s1        # lo = $s0 / $s1
                       # hi = $s0 mod $s1
```



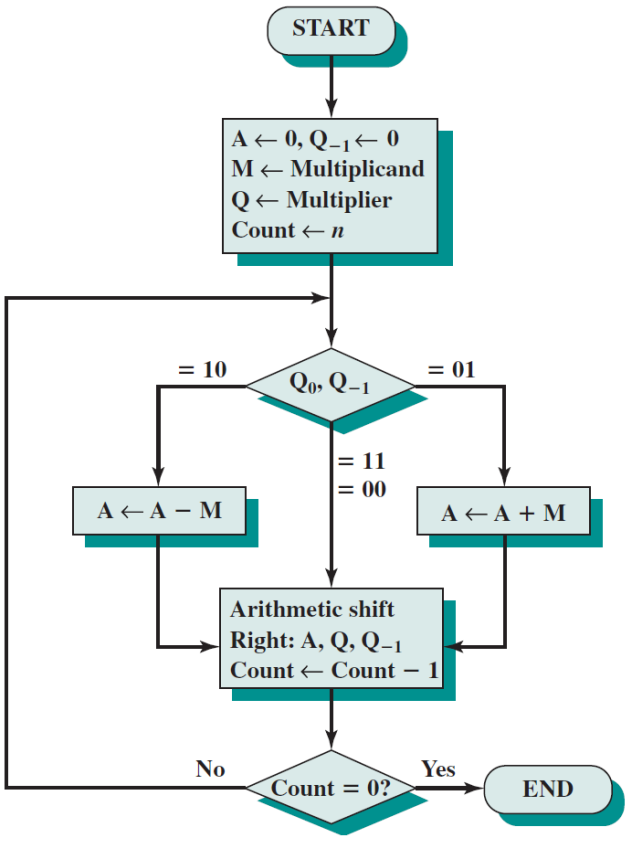
  - Instructions `mfhi rd` and `mflo rd` are provided to move the quotient and reminder to (user accessible) registers in the register file

- **As with multiply, divide ignores overflow so software must determine if the quotient is too large. Software must also check the divisor to avoid division by 0.**

Signed integer multiplication and division

- Reuse unsigned multiplication then fix product sign later
- Multiplication
  - Multiplicand and multiplier are of the same sign: keep product
  - Multiplicand and multiplier are of different sign: negate product
- Division:
  - Dividend and divisor of the same sign:
    - Keep quotient
    - Keep/negate remainder so it is of the same sign with dividend
  - Dividend and divisor of different sign:
    - Negate quotient
    - Keep/negate remainder so it is of the same sign with dividend

Signed integer with Booth algorithm



Representing Big (and Small) Numbers

- Encoding non-integer value?
  - Earth mass: (5.9722±0.0006)×1024 (kg)
  - Weight of an amu (atomic mass unit, 1/12 mass of C_{12})
    0.00000000000000000000000000166 or 1.6×10^{-27} (kg)
  - PI number
    PI = 3.14159….
- Problem: how to represent the above numbers?
- ➔ We need reals or floating-point numbers!
- ➔ Floating point numbers in decimal:
  - ➔ 1000
  - ➔ 1×10^3
  - ➔ 0.1×10^4

Floating point number

- In decimal system

$$2013.1228 = 201.31228 * 10$$
$$= 20.131228 * 10^2$$
$$= 2.0131228 * 10^3$$
$$= 20131228 * 10^{-4}$$

- What is the “standard” form?

$2.0131228 * 10^3 =$ 2.0131228E+03

mantissa exponent

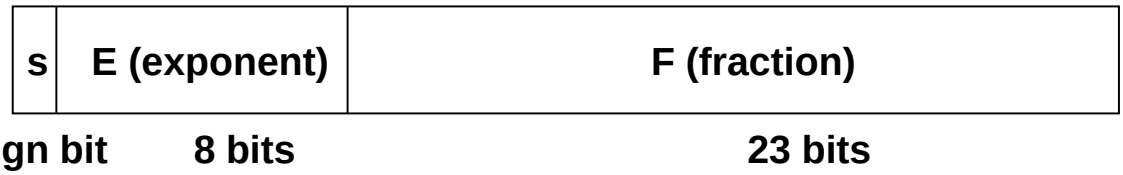
- In binary $X = \pm 1.xxxxx * 2^{yyyy}$
- ***Sign, mantissa, and exponent need to be represented***

Floating point number

- Floating point representation in binary

$$(-1)^{sign} \times 1.F \times 2^{E-bias}$$

  - Still have to fit everything in 32 bits (single precision)
  - Bias = 127 with single precision floating point number



- Defined by the IEEE 754-1985 standard
  - Single precision: 32 bit
  - Double precision: 64 bit
  - Correspond to float and double in C

Examples

- Ex1: convert X into decimal value

X = 1100 0001 0101 0110 0000 0000 0000 0000

sign = 1 → X is negative
E = 1000 0010 = 130
F = 10101100...00
→ X = $(-1)^1 \times 1.101011000..00 \times 2^{130-127}$
= $-1.101011 \times 2^3 = -1101.011$
= -13.375

Example

- Ex2: find decimal value of X

X = 0011 1111 1000 0000 0000 0000 0000 0000

sign = 0

e = 0111 1111 = 127

m = 000…0000 (23 bit 0)

$X = (-1)^0 \times 1.00\ldots000 \times 2^{127-127} = 1.0$

Example

- Ex3: find binary representation of X = 9.6875 in IEEE 754 single precision

Converting X to plain binary

$9_{10} = 1001_2$

0.6875 x 2 = 1.375 → get bit 1
0.375 x 2 = 0.75 → get bit 0
0.75 x 2 = 1.5 → get bit 1
0.5 x = 1.0 → get bit 1

→ $9.6875_{10} = 1001.1011_2$

Example

- Ex3: find binary representation of X = 9.6875 in IEEE 754 single precision

$X = 9.6875_{(10)} = 1001.1011_{(2)} = 1.0011011 \times 2^3$

Then

S = 0

$e = 127 + 3 = 130_{(10)} = 1000\ 0010_{(2)}$

m = 001101100...00 (23 bit)

Finally

X = 0100 0001 0001 1011 0000 0000 0000 0000

Examples

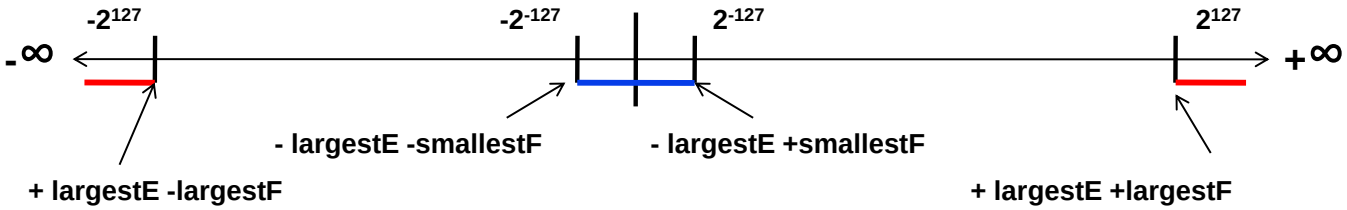
- $1.0_2 \times 2^{-1} =$
- $100.75_{10} =$

Some special values

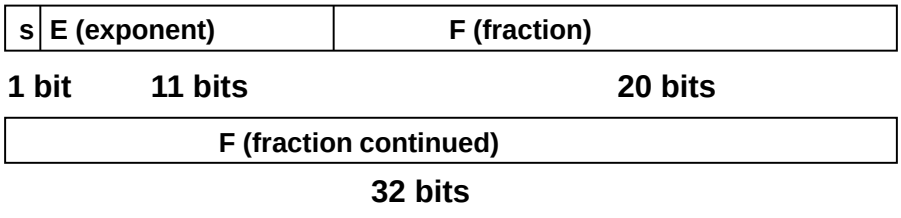
- Smallest+: 0 00000001 1.00000000000000000000000
  = 1 x 2^{1-127}
- Zero: 0 00000000 00000000000000000000000
  = true 0
- Largest+: 0 11111110 1.11111111111111111111111
  = $(2-2^{-23}) \times 2^{254-127}$

Too large or too small values

- Overflow (floating point) happens when a positive exponent becomes too large to fit in the exponent field
- Underflow (floating point) happens when a negative exponent becomes too large to fit in the exponent field



- **Reduce the chance of underflow or overflow is to offer another format that has a larger exponent field**
  - **Double precision – takes two MIPS words**



Reduce underflow with the same bit length?

- De-normalized number

IEEE 754 FP Standard Encoding

- Special encodings are used to represent unusual events
  - ± infinity for division by zero
  - NAN (not a number) for invalid operations such as 0/0
  - True zero is the bit string all zero

Single Precision		Double Precision		Object Represented
E (8)	F (23)	E (11)	F (52)	
0000 0000	0	0000 … 0000	0	true zero (0)
0000 0000	nonzero	0000 … 0000	nonzero	± denormalized number
0111 1111 to +127,-126	anything	0111 …1111 to +1023,-1022	anything	± floating point number
1111 1111	+ 0	1111 … 1111	- 0	± infinity
1111 1111	nonzero	1111 … 1111	nonzero	not a number (NaN)

Floating Point Addition

- Addition (and subtraction)

$$(\pm F1 \times 2^{E1}) + (\pm F2 \times 2^{E2}) = \pm F3 \times 2^{E3}$$

  - **Step 0: Restore the hidden bit in F1 and in F2**
  - **Step 1: Align fractions by right shifting F2 by E1 - E2 positions (assuming E1 ≥ E2) keeping track of (three of) the bits shifted out in G R and S**
  - **Step 2: Add the resulting F2 to F1 to form F3**
  - **Step 3: Normalize F3 (so it is in the form 1.XXXXX …)**
    - **If F1 and F2 have the same sign → F3 ∈[1,4) → 1 bit right shift F3 and increment E3 (check for overflow)**
    - **If F1 and F2 have different signs → F3 may require *many* left shifts each time decrementing E3 (check for underflow)**
  - **Step 4: Round F3 and possibly normalize F3 again**
  - **Step 5: Rehide the most significant bit of F3 before storing the result**

Floating Point Addition Example

- Add

$$(0.5 = 1.0000 \times 2^{-1}) + (-0.4375 = -1.1100 \times 2^{-2})$$

  - **Step 0:**
  - **Step 1:**
  - **Step 2:**
  - **Step 3:**
  - **Step 4:**
  - **Step 5:**

Floating Point Addition Example

- Add: 0.5 + (-0.4375) = ?

$$(0.5 = 1.0000 \times 2^{-1}) + (-0.4375 = -1.1100 \times 2^{-2})$$

  - **Step 0:** **Hidden bits restored in the representation above**
  - **Step 1:** **Shift significand with the smaller exponent (1.1100) right until its exponent matches the larger exponent (so once)**
  - **Step 2:** **Add significands**

    1.0000 + (-0.111) = 1.0000 – 0.111 = 0.001

  - **Step 3:** **Normalize the sum, checking for exponent over/underflow**

    $0.001 \times 2^{-1} = 0.010 \times 2^{-2} = .. = 1.000 \times 2^{-4}$
  - **Step 4:** **The sum is already rounded, so we're done**
  - **Step 5:** **Rehide the hidden bit before storing**

Floating Point Multiplication

- Multiplication

$$(\pm F1 \times 2^{E1}) \times (\pm F2 \times 2^{E2}) = \pm F3 \times 2^{E3}$$

  - **Step 0: Restore the hidden bit in F1 and in F2**
  - **Step 1: Add the two (biased) exponents and subtract the bias from the sum, so E1 + E2 – 127 = E3**

    also determine the sign of the product (which depends on the sign of the operands (most significant bits))
  - **Step 2: Multiply F1 by F2 to form a double precision F3**
  - **Step 3: Normalize F3 (so it is in the form 1.XXXXX …)**
    - **Since F1 and F2 come in normalized → F3 ∈[1,4) → 1 bit right shift F3 and increment E3**
    - **Check for overflow/underflow**
  - **Step 4: Round F3 and possibly normalize F3 again**
  - **Step 5: Rehide the most significant bit of F3 before storing the result**

Floating Point Multiplication Example

- Multiply

$$(0.5 = 1.0000 \times 2^{-1}) \times (-0.4375 = -1.1100 \times 2^{-2})$$

  - **Step 0:**
  - **Step 1:**
  - **Step 2:**
  - **Step 3:**
  - **Step 4:**
  - **Step 5:**

Floating Point Multiplication Example

- Multiply

$$(0.5 = 1.0000 \times 2^{-1}) \times (-0.4375 = -1.1100 \times 2^{-2})$$

  - **Step 0:** **Hidden bits restored in the representation above**
  - **Step 1:** **Add the exponents (not in bias would be -1 + (-2) = -3 and in bias would be (-1+127) + (-2+127) – 127 = (-1 -2) + (127+127-127) = -3 + 127 = 124**
  - **Step 2:** **Multiply the significands**
    1.0000 x 1.110 = 1.110000
  - **Step 3:** **Normalized the product, checking for exp over/underflow**
    1.110000×2^{-3} is already normalized
  - **Step 4:** **The product is already rounded, so we're done**
  - **Step 5:** **Rehide the hidden bit before storing**

Support for Accurate Arithmetic

- **IEEE 754 FP rounding modes**
  - **Always round up (toward +∞)**
  - **Always round down (toward -∞)**
  - **Truncate**
  - **Round to nearest even (when the Guard || Round || Sticky are 100) – always creates a 0 in the least significant (kept) bit of F**
- Rounding (except for truncation) requires the hardware to include extra F bits during calculations
  - Guard and Round bit – 2 additional bits to increase accuracy
  - Sticky bit – used to support Round to nearest even; is set to a 1 whenever a 1 bit shifts (right) through it (e.g., when aligning F during addition/subtraction)

F = 1 . xxxxxxxxxxxxxxxxxxxxxxx G R S

http://pages.cs.wisc.edu/~markhill/cs354/Fall2008/notes/flpt.apprec.html

Example

- Calculate:

  0.2 x 5 = ?

  0.333 x 3 = ?

  (1.0/3) x 3 = ?

Computer Architecture

Ngo Lam Trung & Pham Ngoc Hung
Faculty of Computer Engineering
School of Information and Communication Technology (SoICT)
Hanoi University of Science and Technology
E-mail: [trungnl, hungpn]@soict.hust.edu.vn

Chapter 4: Instruction Set Architecture
(Language of the Computer)

[with materials from *Computer Organization and Design, 5th Edition*,
Patterson & Hennessy, © 2014, MK
and M.J. Irwin's presentation, PSU 2008]

Content

- Introduction
- MIPS Instruction Set Architecture
  - MIPS operands
  - MIPS instruction set
- Programming structures
  - Branching
  - Procedure call
  - Array and string

What is MIPS, and why MIPS?

- CPU designed by John Hennessy's team
  - Stanford's president 2000-2016
  - 2017 Turing award for RISC development
  - "god father" of Silicon Valley
- Very successful CPU in 80s-90s, the first that have 64 bit architecture
- Still very popular in embedded market: set top box, game console,…
- Simple instruction set, appropriate for education (the mini instruction set)

Computer language: hardware operation

- Want to command the computer?

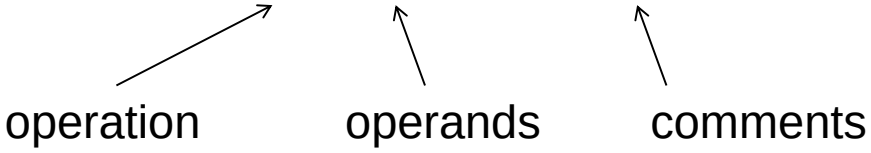
  ➔ You need to speak its language!!!

- Example: MIPS assembly instruction

  add a, b, c #a ← b + c

- Operation performed
  - add b and c,
  - then store result into a

add a, b, c #a ← b + c



Hardware operation

- What does the following code do?

```
add t0, g, h
add t1, i, j
sub f, t0, t1
```

- Equivalent C code

  f = (g + h) – (i + j)

➔ ***Why not making 4 or 5 inputs instructions?***

➔ ***DP1: Simplicity favors regularity!***

Operands

- Object of operation
  - Source operand: provides input data
  - Destination operand: stores the result of operation
- MIPS operands
  - Registers
  - Memory locations
  - Constant/Immediate

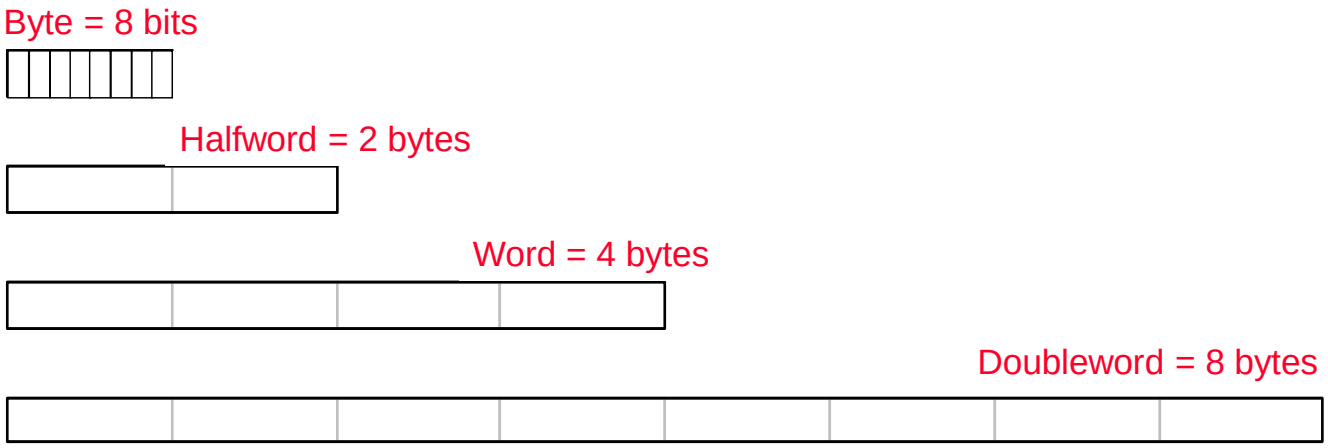
MIPS operands

Name	Example	Comments
32 registers	`$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic, register `$zero` always equals 0, and register `$at` is reserved by the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.

Register operand: MIPS Register File

- Special memory inside CPU, called register file
- 32 slots, each slot is called a register
- Each register holds 32 bits of data (a word)
- Each register has an unique address, and a name
- Register's address is from 0 to 31, represented by 5 bits

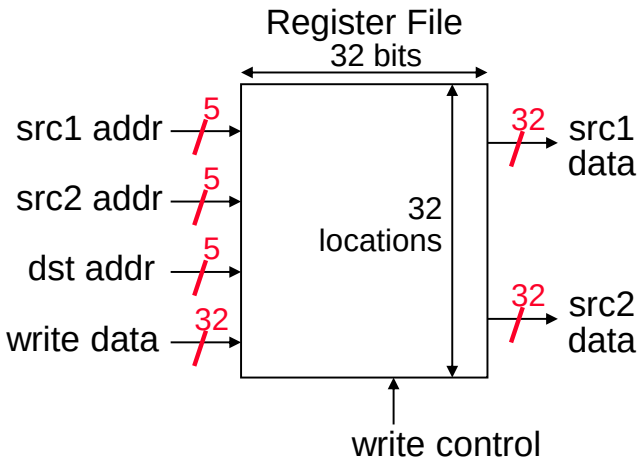
Data types in MIPS



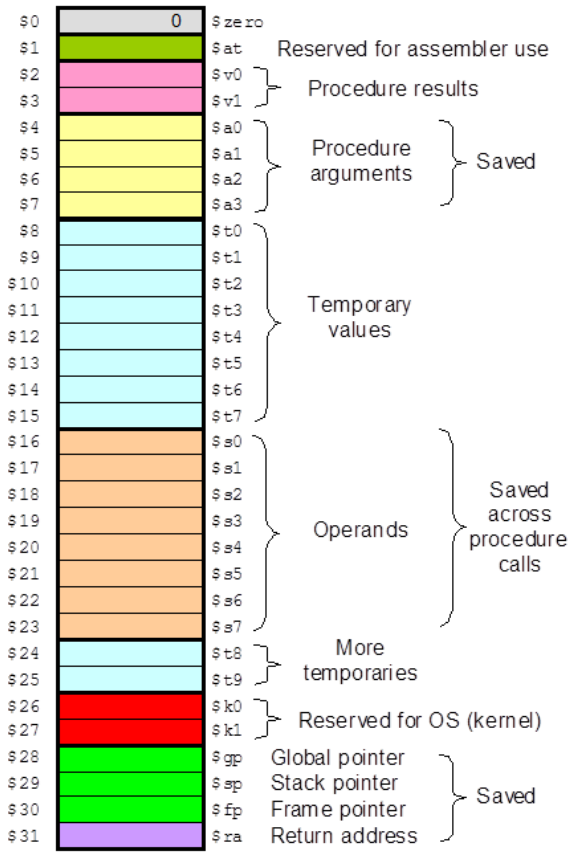
MIPS32 registers hold 32-bit (4-byte) words. Other common data sizes include byte, halfword, and doubleword.

Register operand: MIPS Register File

- Register file in MIPS CPU
  - Two read ports with two source address
  - One write port with one destination address
  - Located in CPU → fast, small size



MIPS Register Convention



- MIPS: load/store machine.
- Typical operation
  - Load data from memory to register
  - **Data processing in CPU**
  - Store data from register to memory

Register operand: MIPS Register File

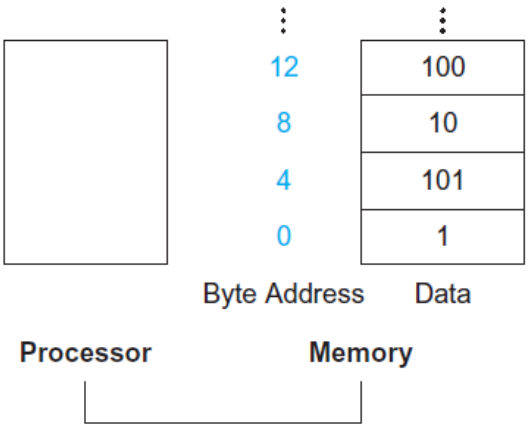
- Register file: “work place” right inside CPU.
- Larger register file should be better, more flexibility for CPU operation.
- Moore’s law: doubled number of transistor every 18 mo.
- Why only 32 registers, not more?

➔ ***DP2: Smaller is faster!***

Effective use of register file is critical!

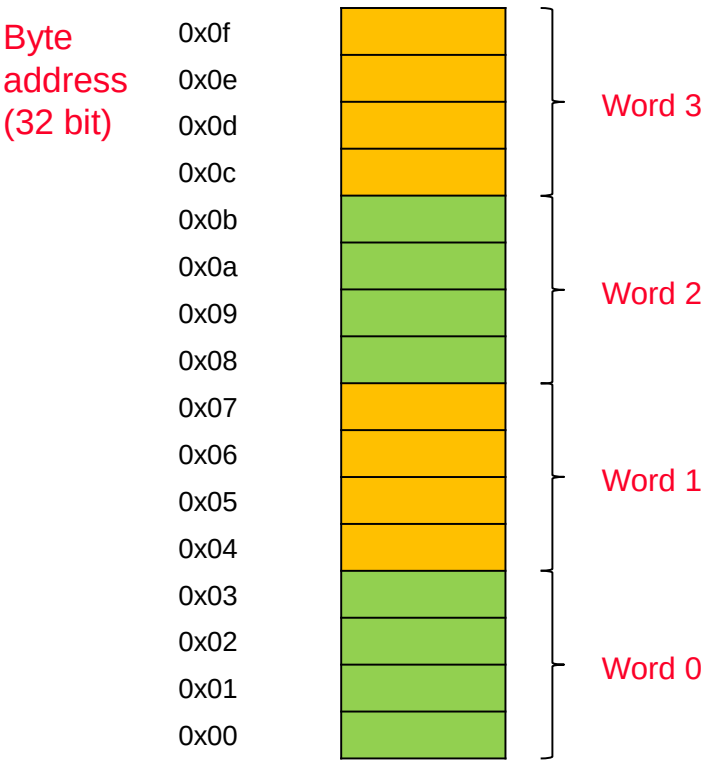
Memory operand

- Data stored in computer's main memory
  - Large size
  - Outsize CPU →Slower than register
- Operations with memory operand
  - Load values from memory to register
  - Store result from register to memory



Byte addressable
Word aligned

MIPS memory organization



- Byte addressable
- Word data access via byte address
- **Only accessible via load/store instructions**

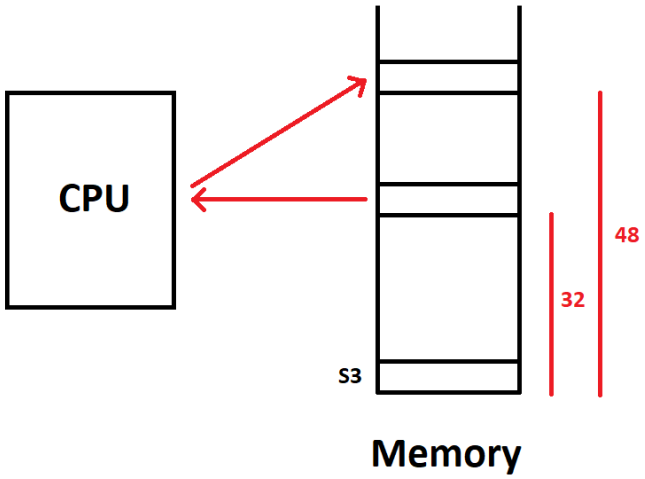
Alignment
In decimal:
Word address = 4 * word number

In binary:
Word address
= word number + 00

Memory operand

- Sample instruction

```
lw $t0,32($s3)
#do sth
#
sw $t0,48($s3)
```



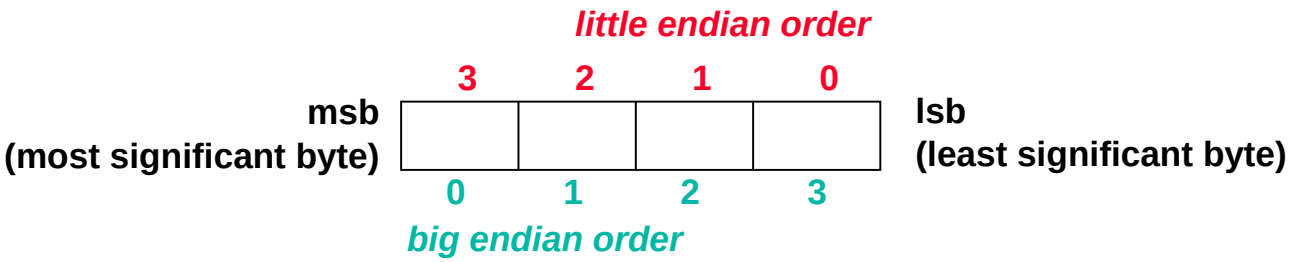
Byte Addresses

- Big Endian: leftmost byte is word address

  IBM 360/370, Motorola 68k, **MIPS**, Sparc, HP PA

- Little Endian: rightmost byte is word address

  Intel 80x86, DEC Vax, DEC Alpha (Windows NT)



Example

- Consider a word in MIPS memory consists of 4 byte with hexa value as below
- What is the word's value?

address	value
X+3	68
X+2	1B
X+1	5D
X	FA

- MIPS is big-endian: address of MSB is X

➔ word's value: FA5D1B68

Immediate operand

- Immediate value specified by the constant number
- Does not need to be stored in register file or memory
  - Value encoded right in instruction → very fast
  - Fixed value specified when developing the program
  - Cannot change value at run time

Immediate operand

- What is the mostly used constant?
- The special register: $zero
- Constant value of 0
- Why?

➔ ***DP3: Making common cases fast!***

Instruction set

- 3 instruction formats:
  - Register (R)
  - Immediate (I)
  - Branch (J)
- R-instruction: all operands are register
- I-instruction: one operand is immediate
- J-instruction: the unconditional branch
- **Note: All MIPS instructions are 32 bits long**

➔ ***Why not only one format?***

➔ ***DP4: Good design demands good compromises!***

5 instruction types

- Arithmetic: addition, subtraction
- Data transfer: transfer data between registers, memory, and immediate
- Logical: and, or, shift
- Conditional branch
- Unconditional branch

Overview of MIPS instruction set

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	$s1 = $s2 + $s3	Three register operands
	subtract	`sub $s1,$s2,$s3`	$s1 = $s2 – $s3	Three register operands
	add immediate	`addi $s1,$s2,20`	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	`lw $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	`sw $s1,20($s2)`	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	`lh $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	`lhu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	`sh $s1,20($s2)`	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	`lb $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	`lbu $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	`sb $s1,20($s2)`	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	`ll $s1,20($s2)`	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	`sc $s1,20($s2)`	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	`lui $s1,20`	$s1 = 20 * 2^{16}	Loads constant in upper 16 bits
Logical	and	`and $s1,$s2,$s3`	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	`or $s1,$s2,$s3`	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	`nor $s1,$s2,$s3`	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	`andi $s1,$s2,20`	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	`ori $s1,$s2,20`	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	`sll $s1,$s2,10`	$s1 = $s2 << 10	Shift left by constant
	shift right logical	`srl $s1,$s2,10`	$s1 = $s2 >> 10	Shift right by constant
Conditional branch	branch on equal	`beq $s1,$s2,25`	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1,$s2,25`	if ($s1!= $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for `beq`, `bne`
	set on less than unsigned	`sltu $s1,$s2,$s3`	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than unsigned
	set less than immediate	`slti $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant
	set less than immediate unsigned	`sltiu $s1,$s2,20`	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant unsigned
Unconditional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to $ra	For switch, procedure return
	jump and link	`jal 2500`	$ra = PC + 4; go to 10000	For procedure call

Fig. 2.1

MIPS Instruction set: Arithmetic operations

- MIPS arithmetic statement

```
add   rd, rs, rt      #rd ← rs + rt
sub   rd, rs, rt      #rd ← rs – rt
addi  rd, rs, const   #rd ← rs + const
```

- rs 5-bits register file address of the first source operand
- rt 5-bits register file address of the second source operand
- rd 5-bits register file address of the result's destination

Example

- Currently $s1 = 6
- What is value of $s1 after executing the following instruction

```
addi  $s2, $s1, 3
addi  $s1, $s1, -2
sub   $s1, $s2, $s1
```

MIPS Instruction set: Logical operations

- Basic logic operations

```
and  rd, rs, rt      #rd ← rs & rt
andi rd, rs, const   #rd ← rs & const
or   rd, rs, rt      #rd ← rs | rt
ori  rd, rs, const   #rd ← rs | const
nor  rd, rs, rt      #rd ← ~(rs | rt)
```

- Example $s1 = 8 = 0000 1000, $s2 = 14 = 0000 1110

```
and  $s3, $s1, $s2
or   $s4, $s1, $s2
```


MIPS Instruction set: Logical operations

- Logical shift and arithmetic shift: move all the bits left or right

```
sll  rd, rs, const   #rd ← rt << const
srl  rd, rs, const   #rd ← rt >> const
sra  rd, rs, const   #rd ← rt >> const
                     (keep sign bit)
```

MIPS Instruction set: Memory Access Instructions

- MIPS has two basic data transfer instructions for accessing memory

```
lw $t0, 4($s3)   #load word from memory

sw $t0, 8($s3)   #store word to memory
```

- The data is loaded into (lw) or stored from (sw) a register in the register file

- The memory address is formed by adding the contents of the base address register to the offset value
- Offset can be negative, and must be multiple of 4

MIPS Instruction set: Load Instruction

- Load/Store Instruction Format:

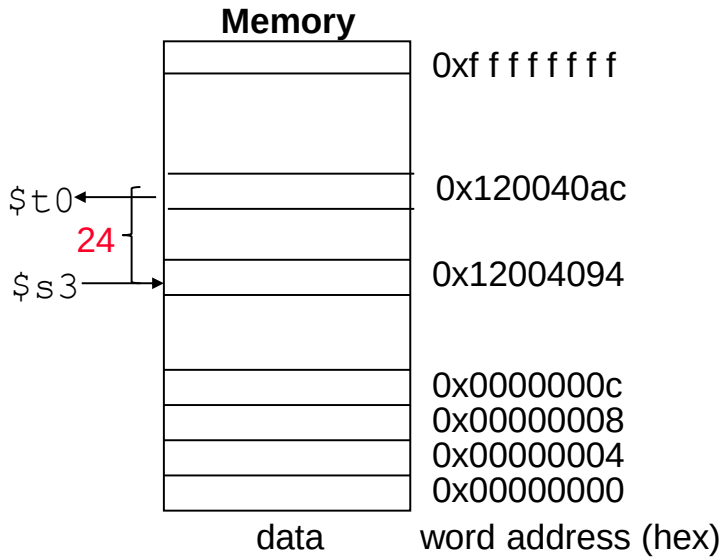
```
lw $t0, 24($s3)    #$t0← mem at 24+$s3
```

(move a word from memory to $t0)

24_{10} + $s3 =

```
   . 0001 1000 (24)
+  . 1001 0100 (94)
   . 1010 1100 (ac)
= 0x1200 40ac
```

0x..94 = ..1001 0100



MIPS Control Flow Instructions

- MIPS conditional branch instructions:

```
bne $s0, $s1, Exit #go to Exit if $s0≠$s1
beq $s0, $s1, Exit #go to Exit if $s0=$s1
```

  - Ex:

```
    if (i==j)
        h = i + j;

    bne $s0, $s1, Exit
    add $s3, $s0, $s1
Exit :  ...
```


Example

```
start:
        addi    s0, zero, 2   #load value for s0
        addi    s1, zero, 2
        addi    s3, zero, 0
        beq     s0, s1, Exit
        add     s3, s2, s1
Exit:   add     s2, s3, s1
.end start
```

What is final value of s2?

In Support of Branch Instructions

- How to use `beq`, `bne`, to support other kinds of branches (e.g., branch-if-less-than)?
- Set flag based on condition: `slt`
- Set on less than instruction:

```
slt $t0, $s0, $s1     # if $s0 < $s1     then
                      # $t0 = 1          else
                      # $t0 = 0
```

- Alternate versions of `slt`

```
slti $t0, $s0, 25     # if $s0 < 25 then $t0=1 ...
sltu $t0, $s0, $s1    # if $s0 < $s1 then $t0=1 ...
sltiu $t0, $s0, 25    # if $s0 < 25 then $t0=1 ...
```

- How about set on bigger than?

Unconditional branch

- MIPS also has an unconditional branch instruction or jump instruction:

```
j  label        #go to label
```

Example

- Write assembly code to do the following

```
if (i<5)
    X = 3;
else
    X = 10;
```

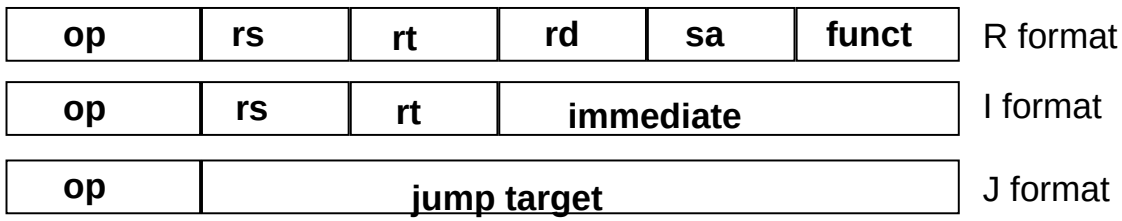
Solution

```
      slti $t0,$s1,5       # i<5? (inverse condition)
      beq  $t0,$zero,else  # if i>=5 goto else part
      addi $t1,$zero,3     # X = 3
      j    endif           # skip the else part
else: addi $t1,$zero,10    # X = 10
endif:...
```


Representation of MIPS instruction

- All MIPS instructions are 32 bits wide
- Instructions are 32 bits binary number

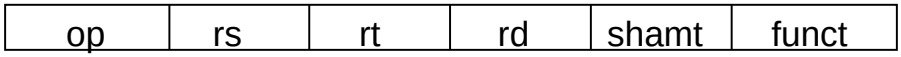
3 Instruction Formats: all 32 bits wide



Reference: MIPS Instruction Reference (MIPS_IR.pdf)

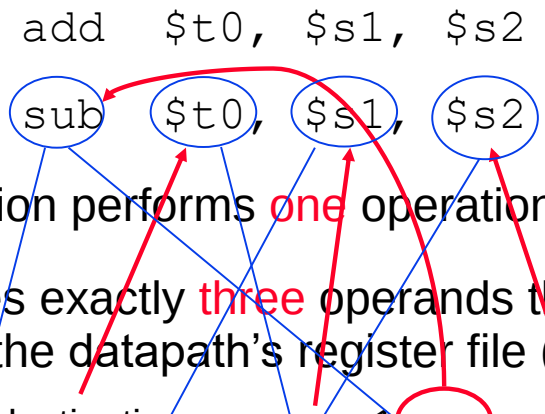
R-format instruction

- All fields are encoded by mnemonic names

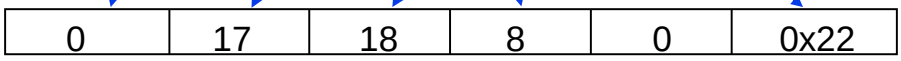


op	6-bits	opcode that specifies the operation
rs	5-bits	register file address of the first source operand
rt	5-bits	register file address of the second source operand
rd	5-bits	register file address of the result's destination
shamt	5-bits	shift amount (for shift instructions)
funct	6-bits	function code augmenting the opcode

Example of R-format instruction



- Each instruction performs one operation
- Each specifies exactly three operands that are all contained in the datapath's register file (`$t0,$s1,$s2`)
- Binary code of Instruction



Example

- Find machine codes of the following instructions

```
lw    $t0,0($s1)   # initialize maximum to A[0]
addi  $t1,$zero,0  # initialize index i to 0
addi  $t1,$t1,1    # increment index i by 1
```


Example of I-format instruction

```
slti $t0, $s2, 15    #$t0 = 1 if $s2<15
```

- Machine format (I format):

0x0A	18	8	0x0F

- The constant is kept inside the instruction itself!
  - Immediate format limits values to the range $+2^{15}-1$ to -2^{15}

Example

The simple switch

```
switch(test) {
  case 0:
       a=a+1; break;
  case 1:
       a=a-1; break;
  case 2:
       b=2*b; break;
  default:
}
```

Assuming that: `test,a,b` are stored in `$s1,$s2,$s3`

Solution

```
      beq   s1,t0,case_0
      beq   s1,t1,case_1
      beq   s1,t2,case_2
      b     default
case_0:
      addi  s2,s2,1      #a=a+1
      b     continue
case_1:
      sub   s2,s2,t1     #a=a-1
      b     continue
case_2:
      add   s3,s3,s3     #b=2*b
      b     continue
default:
continue:
```


Exercise

- How branch instruction is executed?

```
        slti $t0,$s1,5
        bne  $t0,$zero,0x02   ⟶ How can CPU jump from here
                                to the "else" label?
        addi $t1,$zero,3
        j    endif
else:   addi $t1,$zero,10
endif:...
```

Example

- Write assembly code correspond to the following C code

```
        for (i = 0; i < n; i++)
            sum = sum + A[i];
```

```
loop:
     addi  s1,s1,1      #i=i+step
     add   t1,s1,s1     #t1=2*s1
     add   t1,t1,t1     #t1=4*s1
     add   t1,t1,s2     #t1 <- address of A[i]
     lw    t0,0(t1)     #load value of A[i] in t0
     add   s5,s5,t0     #sum = sum+A[i]
     bne   s1,s3,loop   #if i != n, goto loop
```


Example

The simple while loop: `while (A[i]==k) i=i+1;`

Assuming that: `i, A, k` are stored in `$s1,$s2,$s3`

Solution

```
loop: add   $t1,$s1,$s1        # t1 = 4*i
      add   $t1,$t1,$t1        #
      add   $t1,$t1,$s2        # t1 = A + 4*I,
                                 address of A[i]
      lw    $t0,0($t1)         # load data in A[i]
                                 into t0
      bne   $t0,$s3,endwhl     #
      addi $s1,$s1,1           #
      j     loop               #
endwhl: …                      #
```

Instructions for Accessing Procedures

- MIPS procedure call instruction:

```
jal   ProcedureAddress     #jump and link
```

- Saves PC+4 in register $ra to have a link to the next instruction for the procedure return
- Machine format (J format):

0x03	26 bit address

- Then can do procedure return with

```
jr    $ra                  #return
```

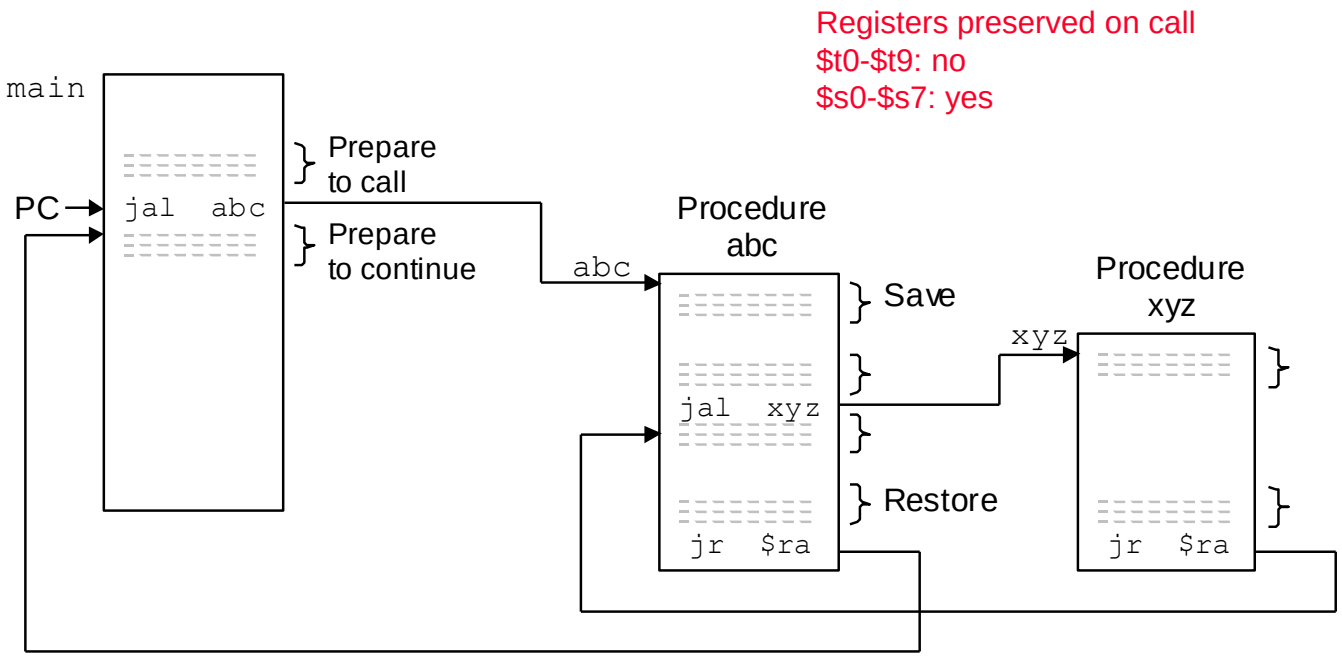
- Instruction format (R format):

0	31				0x08

Six Steps in the Execution of a Procedure

1. Main routine (caller) places parameters in a place where the procedure (callee) can access them
   - `$a0` - `$a3`: four argument registers
2. Caller transfers control to the callee (jal)
3. Callee acquires the storage resources needed
4. Callee performs the desired task
5. Callee places the result value in a place where the caller can access it
   - `$v0` - `$v1`: two value registers for result values
6. Callee returns control to the caller (jr)
   - `$ra`: one return address register to return to the point of origin

Procedure call and nested procedure call



Example of nested procedure call

Procedure that does not call another proc.

- C code:

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

  - g, h, i, j stored in $a0, $a1, $a2, $a3
  - f in $s0 (need to be saved)
  - $t0 and $t1 used for temporary data, also need to be saved
  - Result in $v0

Sample code

```
leaf_example:
        addi   $sp, $sp, -12      # room for 3 items
        sw     $t1, 8($sp)        # save $t1
        sw     $t0, 4($sp)        # save $t0
        sw     $s0, 0($sp)        # save $s0
        add    $t0, $a0, $a1      # $t0 = g+h
        add    $t1, $a2, $a3      # $t1 = i+j
        sub    $s0, $t0, $t1      # $s0 = (g+h)-(i+j)
        add    $v0, $s0, $zero    # return value in $v0
        lw     $s0, 0($sp)        # restore $s0
        lw     $t0, 4($sp)        # restore $t0
        lw     $t1, 8($sp)        # restore $t1
        addi   $sp, $sp, 12       # shrink stack
        jr     $ra                # return to caller
```


Stack usage

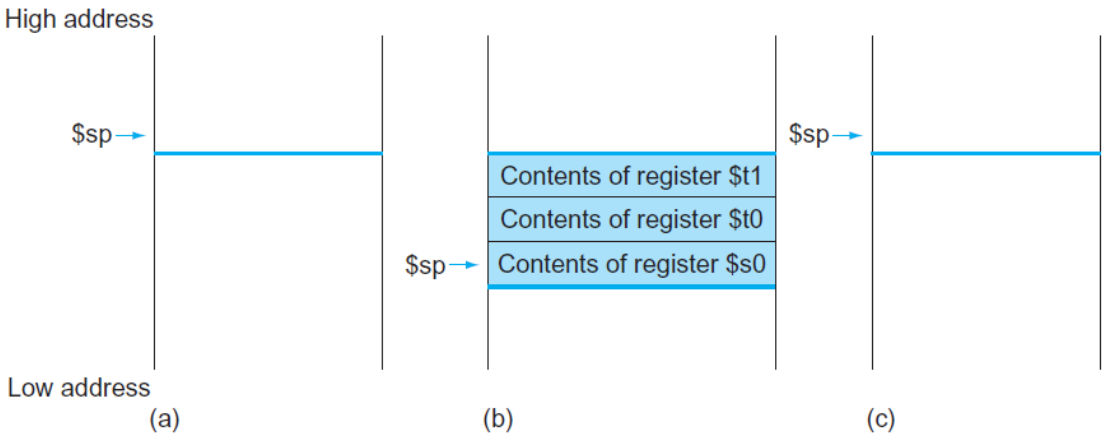


FIGURE 2.10 The values of the stack pointer and the stack (a) before, (b) during, and (c) after the procedure call. The stack pointer always points to the "top" of the stack, or the last word in the stack in this drawing.

Procedure with nested proc.

- C code:

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

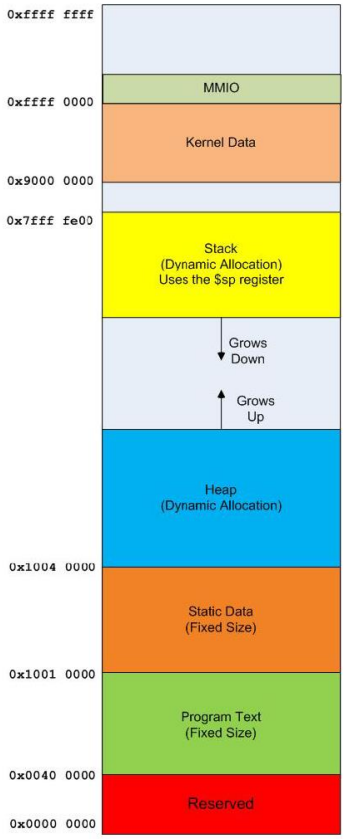
  - n in $a0
  - Result in $v0

Sample code

```
fact:
     addi  $sp, $sp, -8      #2 items in stack
     sw    $ra, 4($sp)       #save return address
     sw    $a0, 0($sp)       #and current n
     slti  $t0, $a0, 1       #check base case
     beq   $t0, $zero, L1    #
     addi  $v0, $zero, 1     #value 1 for base case
     addi  $sp, $sp, 8       #then shrink stack
     jr    $ra               #and return
L1:  addi  $a0, $a0, -1      #otherwise reduce n
     jal   fact              #then call fact again
     lw    $a0, 0($sp)       #restore n
     lw    $ra, 4($sp)       #and return address
     addi  $sp, $sp, 8       #shrink stack
     mul   $v0, $a0, $v0     #value for normal case
     jr    $ra               #and return
```

MIPS memory configuration

- Program text: stores machine code of program, declared with *.text*
- Static data: data segment, declared with *.data*
- Heap: for dynamic allocation
- Stack: for local variable and dynamic allocation via push/pop
- Kernel: for OS's use
- MIMO: memory mapped IO for accessing input/output devices



Working with 32 bit immediates and addresses

- Operations that needs 32-bit literals
  - Loading 32-bit integers to registers
  - Loading variable addresses to registers
- I-format instructions only support 16-bit literals → combine two instructions
- Example: load the value 0x3D0900 into $s0

```
lui $s0, 0x003D        #$s0 ← 0x003D0000

ori #s0, $s0, 0x0900   #$s0 ← 0x003D0900
```

- Pseudo-instructions: combination of real instructions, for convenience
  - li, la, move…
  - bge, bgt, ble…

Accessing characters and string

- Accessing characters

```
lb $s0, 0($s1)      #load byte with sign-extension
lbu $s0, 0($s1)     #load byte with zero-extension
sb $s0, 0($s1)      #store LSB to memory
```

- String is accessed as array of characters
- Example: string copy

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


Accessing characters and string

```
#x and y are in $a0 and $a1, i in $s0
strcpy:
    addi $sp,$sp,-4         # adjust stack for 1 more item
    sw $s0, 0($sp)          # save $s0
    add $s0,$zero,$zero     # i = 0 + 0
L1: add $t1,$s0,$a1         # address of y[i] in $t1
    lbu $t2, 0($t1)         # $t2 = y[i]
    add $t3,$s0,$a0         # address of x[i] in $t3
    sb $t2, 0($t3)          # x[i] = y[i]
    beq $t2,$zero,L2        # if y[i] == 0, go to L2
    addi $s0, $s0,1         # i = i + 1
    j L1                    # go to L1
L2: lw $s0, 0($sp)          # y[i] == 0: end of string.
                            # Restore old $s0
    addi $sp,$sp,4          # pop 1 word off stack
    jr $ra                  # return
```

Interchange sort function

```
void sort (int v[], int n)
{
    int i, j;
    for (i = 0; i < n; i += 1)
    {
        for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j-=1)
        {
            swap(v,j);
        }
    }
}
void swap(int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```


Sorting function

Procedure body

```
swap:  sll    $t1, $a1, 2          # reg $t1 = k * 4
       add    $t1, $a0, $t1        # reg $t1 = v + (k * 4)
                                   # reg $t1 has the address of v[k]
       lw     $t0, 0($t1)          # reg $t0 (temp) = v[k]
       lw     $t2, 4($t1)          # reg $t2 = v[k + 1]
                                   # refers to next element of v
       sw     $t2, 0($t1)          # v[k] = reg $t2
       sw     $t0, 4($t1)          # v[k+1] = reg $t0 (temp)
```

Procedure return

```
       jr     $ra                  # return to calling routine
```

Saving registers

```
sort:   addi    $sp,$sp, -20        # make room on stack for 5 registers
        sw      $ra, 16($sp)# save $ra on stack
        sw      $s3,12($sp)         # save $s3 on stack
        sw      $s2, 8($sp)# save $s2 on stack
        sw      $s1, 4($sp)# save $s1 on stack
        sw      $s0, 0($sp)# save $s0 on stack
```

Procedure body

Move parameters		move	$s2, $a0 # copy parameter $a0 into $s2 (save $a0)
		move	$s3, $a1 # copy parameter $a1 into $s3 (save $a1)
Outer loop		move	$s0, $zero# i = 0
	for1tst:	slt	$t0, $s0,$s3 #reg$t0=0if$s0Š$s3(iŠn)
		beq	$t0, $zero, exit1# go to exit1 if $s0 Š $s3 (i Š n)
Inner loop		addi	$s1, $s0, -1# j = i - 1
	for2tst:	slti	$t0, $s1,0 #reg$t0=1if$s1<0(j<0)
		bne	$t0, $zero, exit2# go to exit2 if $s1 < 0 (j < 0)
		sll	$t1, $s1, 2# reg $t1 = j * 4
		add	$t2, $s2, $t1# reg $t2 = v + (j * 4)
		lw	$t3, 0($t2)# reg $t3 = v[j]
		lw	$t4, 4($t2)# reg $t4 = v[j + 1]
		slt	$t0, $t4, $t3 # reg $t0 = 0 if $t4 Š $t3
		beq	$t0, $zero, exit2# go to exit2 if $t4 Š $t3
Pass parameters and call		move	$a0, $s2 # 1st parameter of swap is v (old $a0)
		move	$a1, $s1 # 2nd parameter of swap is j
		jal	swap # swap code shown in Figure 2.25
Inner loop		addi	$s1, $s1, -1# j -= 1
		j	for2tst # jump to test of inner loop
Outer loop	exit2:	addi	$s0, $s0, 1 # i += 1
		j	for1tst # jump to test of outer loop

Restoring registers

```
exit1:  lw      $s0, 0($sp)         # restore $s0 from stack
        lw      $s1, 4($sp)# restore $s1 from stack
        lw      $s2, 8($sp)# restore $s2 from stack
        lw      $s3,12($sp)         # restore $s3 from stack
        lw      $ra,16($sp)         # restore $ra from stack
        addi    $sp,$sp, 20         # restore stack pointer
```

Procedure return

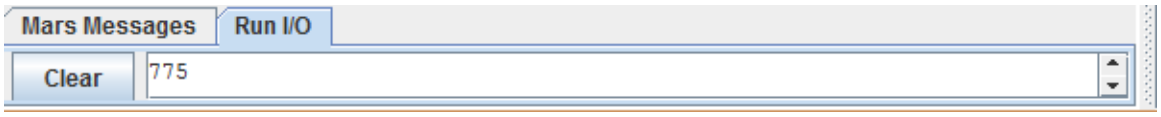
```
        jr      $ra                 # return to calling routine
```


Exercises

syscall

- **Print decimal** integer to standard output (the console).
- Argument(s):
  - $v0 = 1
  - $a0 = number to be printed
- Return value: none

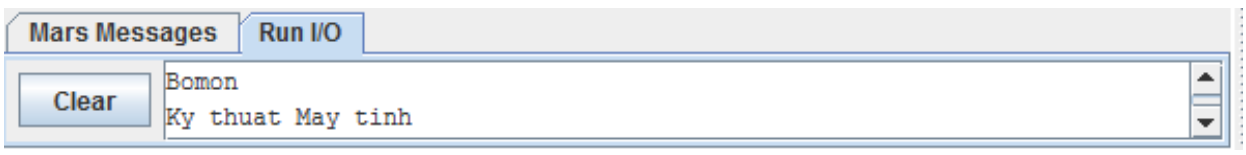
```
li   $v0, 1            # service 1 is print integer
li   $a0, 0x307        # the interger to be printed is 0x307
syscall                # execute
```



syscall

- **Print string** to standard output (the console).
- Argument(s)
  - $v0 = 4
  - $a0 = address of null terminated string to print
- Return value: none

```
.data
Message: .asciiz "Bomon \nKy thuat May tinh"
.text
   li   $v0, 4
   la   $a0, Message
   syscall
```



syscall

- **Read integer** from standard input (the console).
- Argument
  - $v0 = 5
- Return value
  - $v0 = contains integer read

```
li      $v0, 5
syscall
```


syscall

- **Read string** from standard input
- Argument(s):
  - $v0 = 8
  - $a0 = address of input buffer
  - $a1 = maximum number of characters to read
- Return value: none
- Note: for specified length n, string can be no longer than n-1.
  - If less than that, adds newline to end.
  - In either case, then pads with null byte
- String can be declared with *.space*

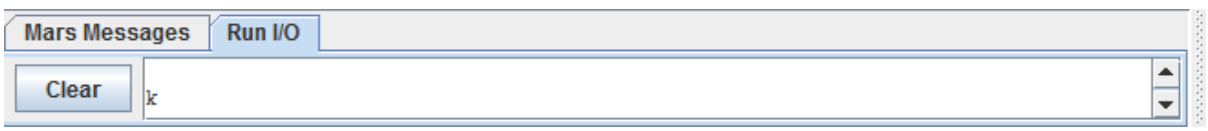
syscall

```
.data
Message: .space 100     # string with max len = 99
.text
   li  $v0, 8
   la  $a0, Message
   li  $a1, 100
   syscall
```


syscall

- **Print** a **character** to standard output.
- Arguments
  - $v0 = 11
  - $a0 = character to print (at LSB)
- Return value: none

```
li  $v0, 11
li  $a0, 'k'
syscall
```



syscall

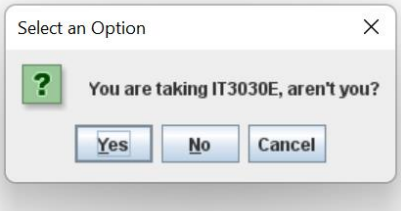
- **Read** a **character** from standard input.
- Argument(s):
  - $v0 = 12
- Return value:
  - $v0 contains the character read

syscall

- ConfirmDialog
- Argument(s):
  - $v0 = 50
  - $a0 = address of the null-terminated message string
- Return value: $a0 = value of selected option

  0: Yes 1: No 2: Cancel

```
.data
Message: .asciiz "You are taking IT3030E, aren't you?"
.text
   li   $v0, 50
   la   $a0, Message
   syscall
```

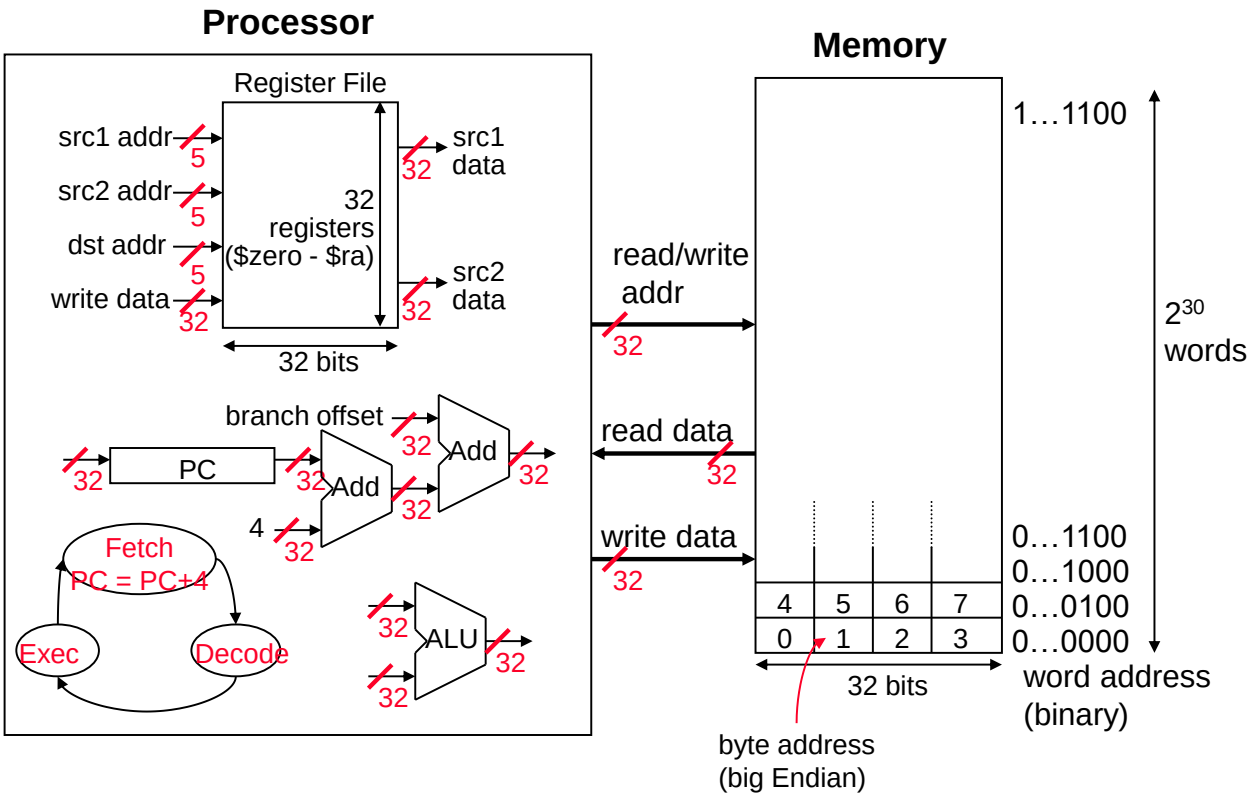


syscall

- exit: terminate the program
- Argument
  - $v0 = 10
- Return value: none

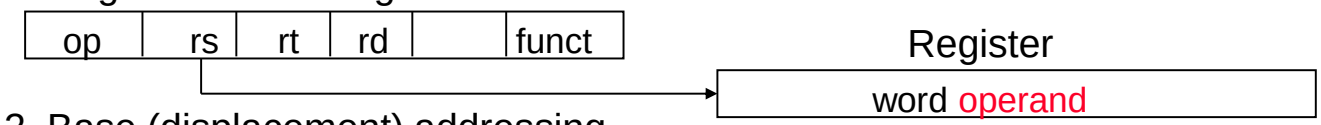
Exercise

MIPS Organization

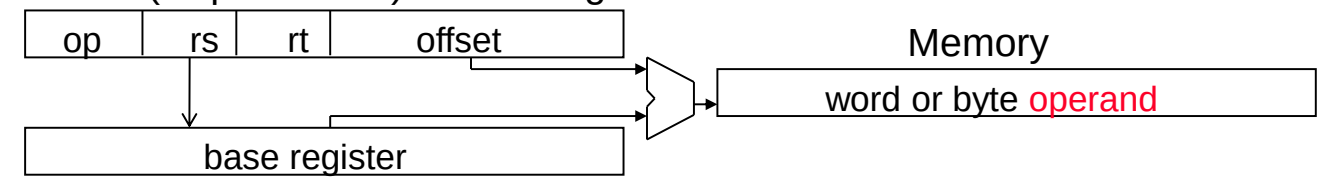


Addressing Modes Illustrated

1. Register addressing

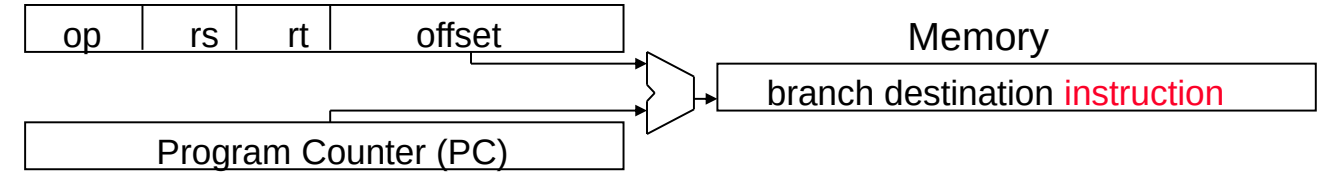


2. Base (displacement) addressing

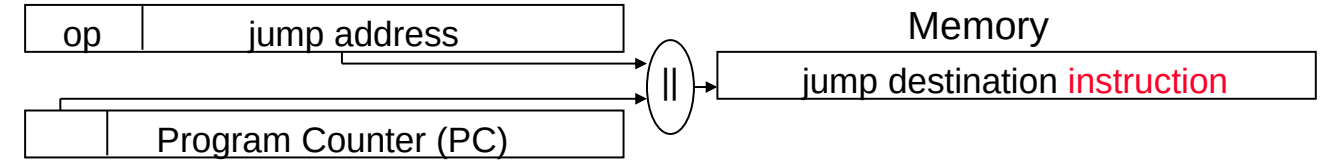


3. Immediate addressing

4. PC-relative addressing



5. Pseudo-direct addressing



Summary

- Provided one problem to be solved by computer
  - Can it be implemented?
  - Can it be programmed?
  - Which CPU is suitable?
- Metric of performance
  - How many bytes does the program occupy in memory?
  - How many instructions are executed?
  - How many clocks are required per instruction?
  - How much time is required to execute the program?

➔ Largely depend on Instruction Set Architecture (ISA)

Chapter 5: The Processor

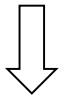
Ngo Lam Trung, Pham Ngoc Hung

[with materials from *Computer Organization and Design, 4th Edition*, Patterson & Hennessy, © 2008, MK and M.J. Irwin's presentation, PSU 2008]

Review

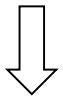
High-level language program (in C)

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



Assembly language program (for MIPS)

```
swap:
      muli $2, $5,4
      add  $2, $4,$2
      lw   $15, 0($2)
      lw   $16, 4($2)
      sw   $16, 0($2)
      sw   $15, 4($2)
      jr   $31
```



Binary machine language program (for MIPS)

```
00000000101000010000000000011000
00000000000110000001100000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000
```

Performance metric

CPU time = CPI * CC * IC

CPI: cycle per instruction
CC: clock cycle
IC: instruction count

How to improve?

- IC:
- CC:
- CPI:

In this chapter

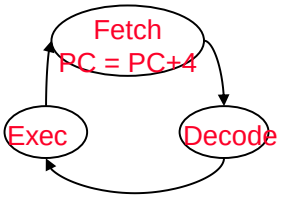
- Implementation of data path
- How to get CPI < 1

Overview

- We will examine two MIPS implementations
  - A simplified version
  - A more realistic pipelined version
- Limit to a simple subset of MIPS ISA
  - Memory reference: `lw`, `sw`
  - Arithmetic/logical: `add`, `sub`, `and`, `or`, `slt`
  - Control transfer: `beq`, `j`
- Implementation of real CPU with other instructions are similar to the simplified version (theoretically!)

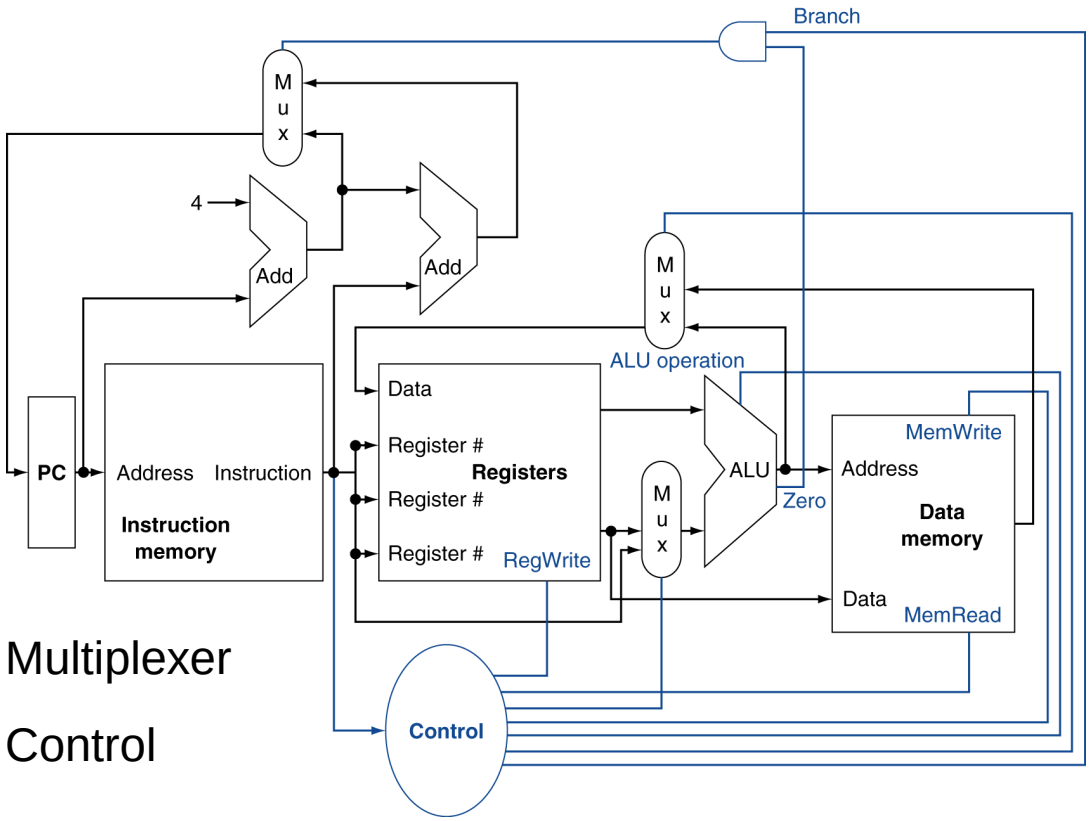
General instruction cycle

- Generic implementation
  - use the program counter (PC) to supply the instruction address and fetch the instruction from memory (and update the PC)
  - decode the instruction (and read registers)
  - execute the instruction



- All instructions (except `j`) use the ALU after reading the registers
  - ALU: Arithmetic and Logic Unit, where the arithmetic and logic operations are executed
- In this chapter: implementation of CPU that can execute the simple subset of MIPS ISA

CPU implementation with MUXes and Control

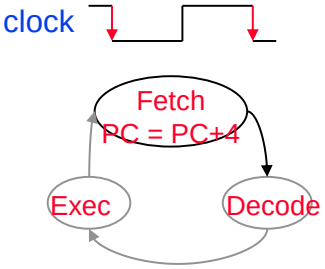


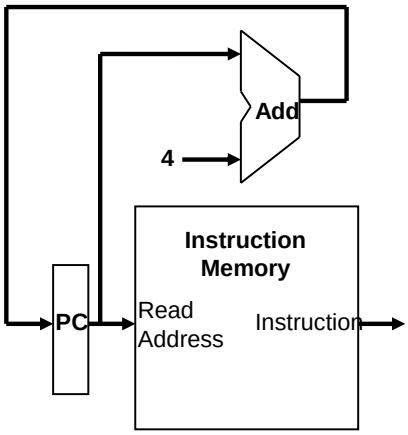
- Multiplexer
- Control

Don't panic! We'll build this incrementally.

Fetching Instructions

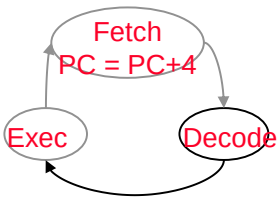
- Fetching instructions involves
  - reading the instruction from the Instruction Memory
  - updating the PC value to be the address of the next instruction in memory

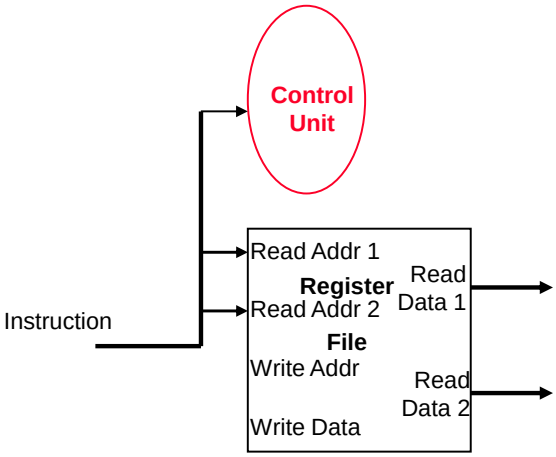




Decoding Instructions

- Decoding instructions involves
  - sending the fetched instruction's opcode and function field bits to the control unit
  - The control unit send appropriate control signals to other parts inside CPU to execute the operations corresponds to the instruction



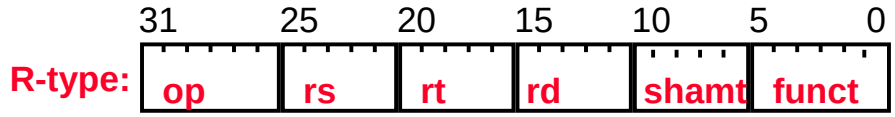


- Example: reading two values from the Register File

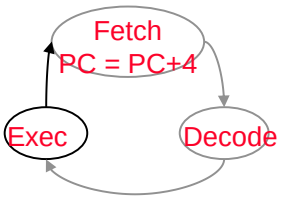
  →Register File addresses are contained in the instruction

Executing R Format Operations

- R format operations (**`add, sub, slt, and, or`**)



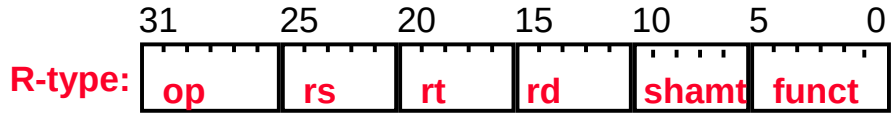
  - read two register operands rs and rt
  - perform operation (op and funct) on values in rs and rt
  - store the result back into the Register File (into location rd)



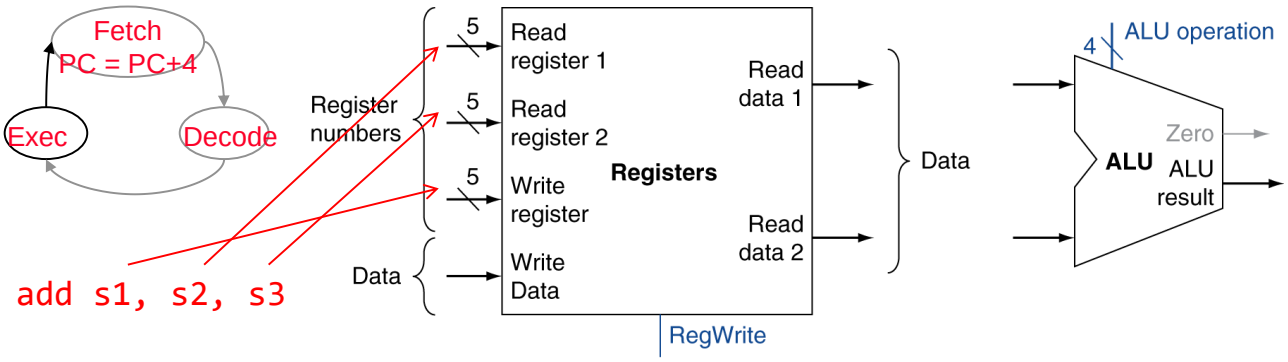

```
Example: add s1, s2, s3
- Value of s2 and s3 are sent to ALU
- ALU execute the s2 + s3 operation
- Result is store into s1
```


Executing R Format Operations

- R format operations (**`add, sub, slt, and, or`**)



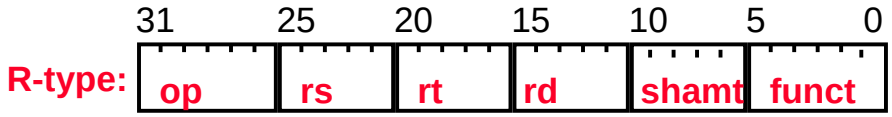
  - read two register operands rs and rt
  - perform operation (op and funct) on values in rs and rt
  - store the result back into the Register File (into location rd)



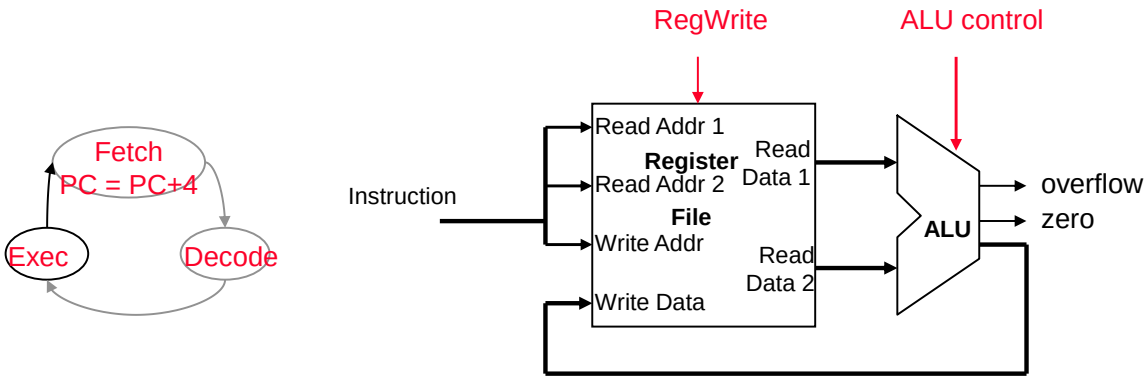
Draw connection between a and b to form the execution unit?

Executing R Format Operations

- R format operations (**`add, sub, slt, and, or`**)



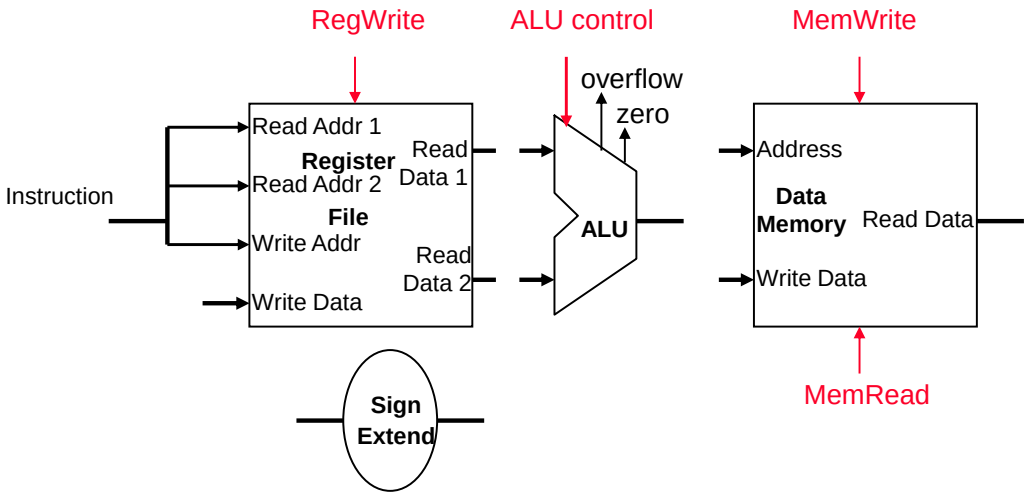
  - read two register operands rs and rt
  - perform operation (op and funct) on values in rs and rt
  - store the result back into the Register File (into location rd)



  - We need the write control signal to control when the result is written to Register File

Executing Load and Store Operations

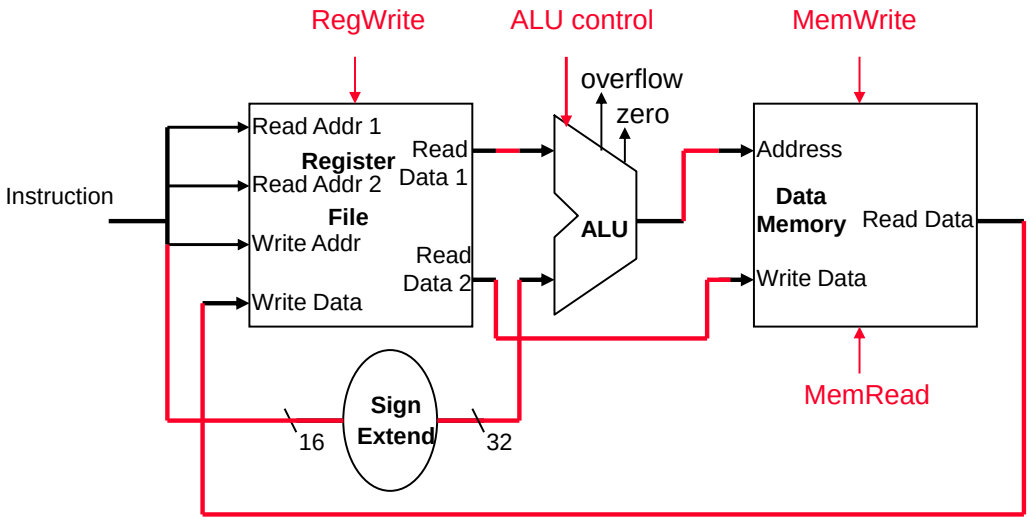
- Load and store operations involves
  - read register operands (including one base register)
  - compute memory address by adding the base to the offset
    - The 16-bit offset field in the instruction is sign-extended to 32 bit
  - store: read from the Register File, write to the Data Memory
  - load: read from the Data Memory, write to the Register File



Draw necessary connections to form execution unit?

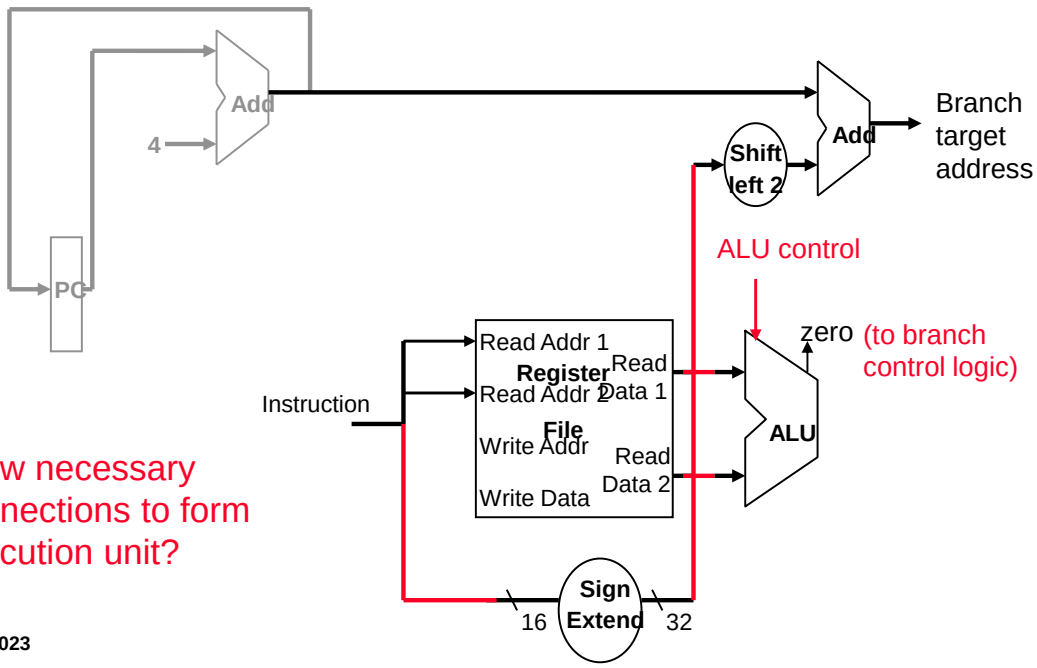
Executing Load and Store Operations

- Load and store operations involves
  - read register operands (including one base register)
  - compute memory address by adding the base to the offset
    - The 16-bit offset field in the instruction is signed-extended to 32 bit
  - store: read from the Register File, write to the Data Memory
  - load: read from the Data Memory, write to the Register File



Executing Branch Operations

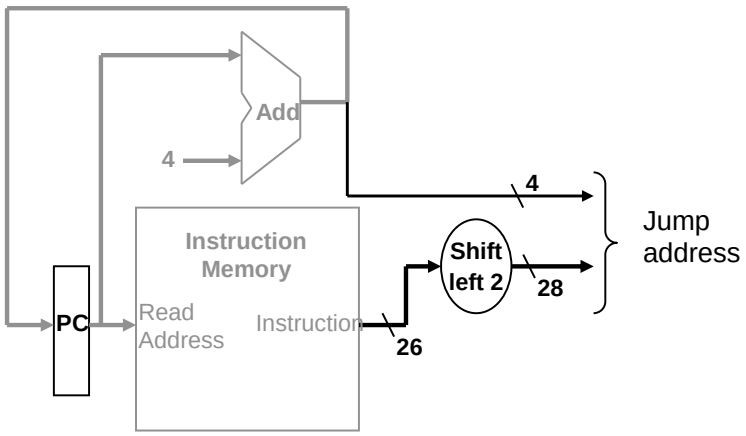
- Branch operations involves
  - read register operands
  - compare the operands (subtract, check **`zero`** ALU output)
  - compute the branch target address: adding the updated PC to the 16-bit signed-extended offset field in the instr



Draw necessary connections to form execution unit?

Executing Jump Operations

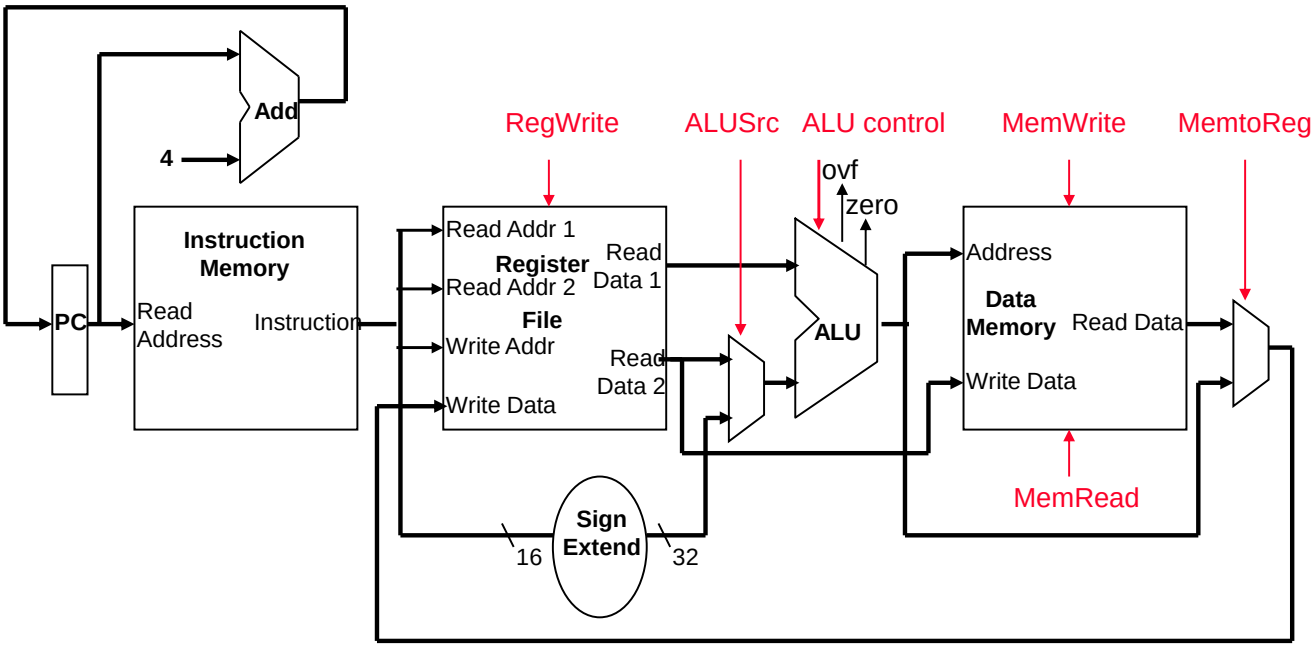
- Jump operation involves
  - keep 4 highest bits of PC
  - replace the lower 28 bits of the PC by
    - the lower 26 bits of the fetched instruction shifted left by 2 bits



Creating a Single Datapath from the Parts

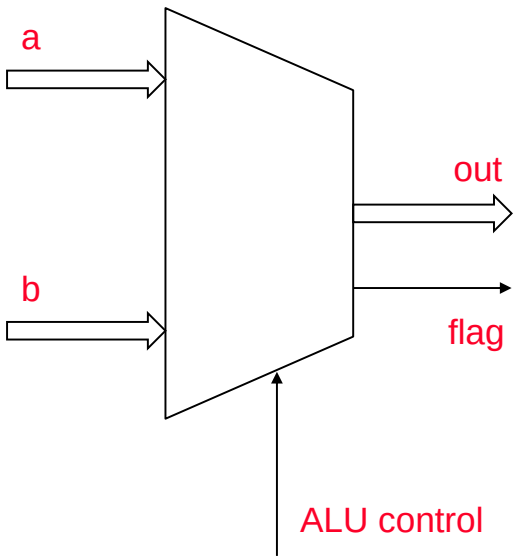
- Assemble the datapath segments and add control lines and multiplexors as needed
- Single cycle design – fetch, decode and execute each instructions in one clock cycle
  - separate Instruction Memory and Data Memory, though they are both in main memory
  - multiplexors needed at the input of shared elements with control lines to do the selection
  - write signals to control writing to the Register File and Data Memory

Fetch, R, and Memory Access Portions

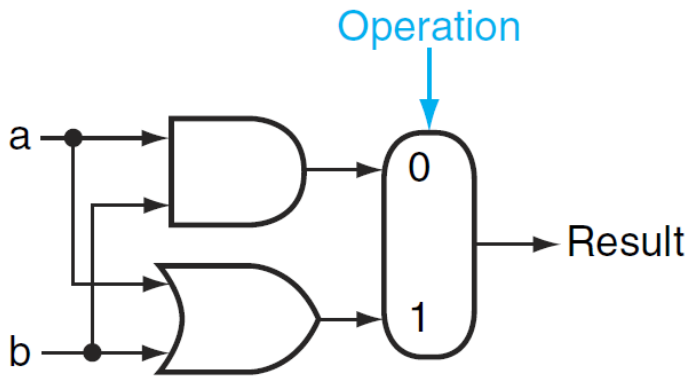


Designing ALU

- Input/output
  - Two data input: a, b
  - ALU control signal
  - Data out
  - Flags out
- Operations
  - And, or, nor
  - Add, subtract

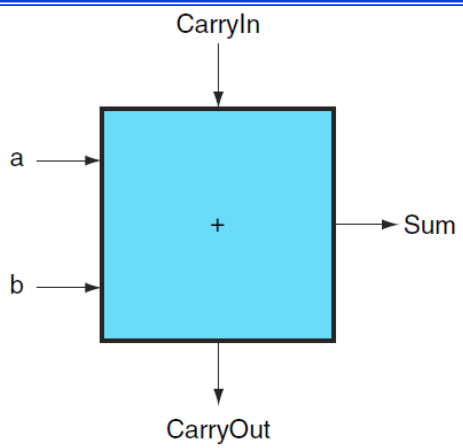


1-bit ALU with logic operation



- What do we have if
  - Operation = 0:
  - Operation = 1:

1-bit full-adder

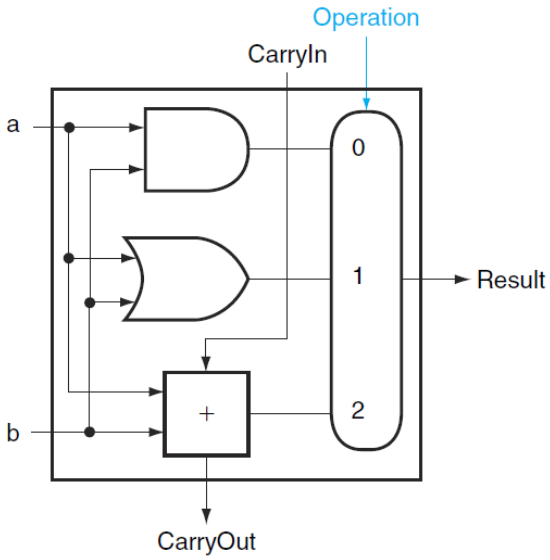


$$\text{CarryOut} = (b \cdot \text{CarryIn}) + (a \cdot \text{CarryIn}) + (a \cdot b)$$

$$\text{Sum} = (a \cdot \overline{b} \cdot \overline{\text{CarryIn}}) + (\overline{a} \cdot b \cdot \overline{\text{CarryIn}}) + (\overline{a} \cdot \overline{b} \cdot \text{CarryIn}) + (a \cdot b \cdot \text{CarryIn})$$

➔ We already designed this in prev. chapter

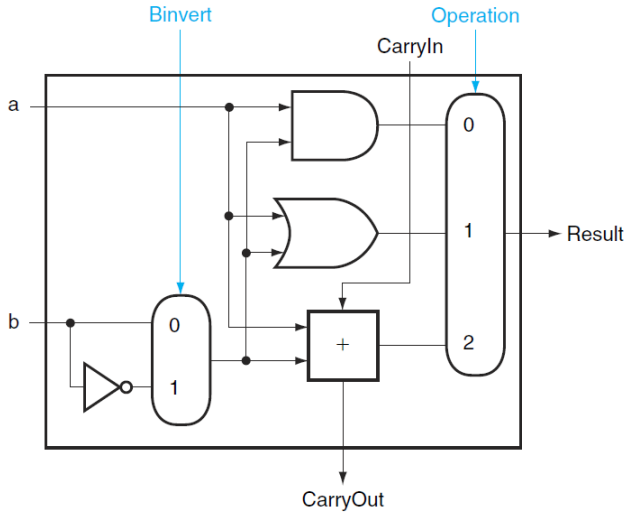
1-bit ALU with AND, OR, ADD



- Operation = 00:
- Operation = 01:
- Operation = 10:

How about 1-bit ALU with AND, OR, ADD, SUB?

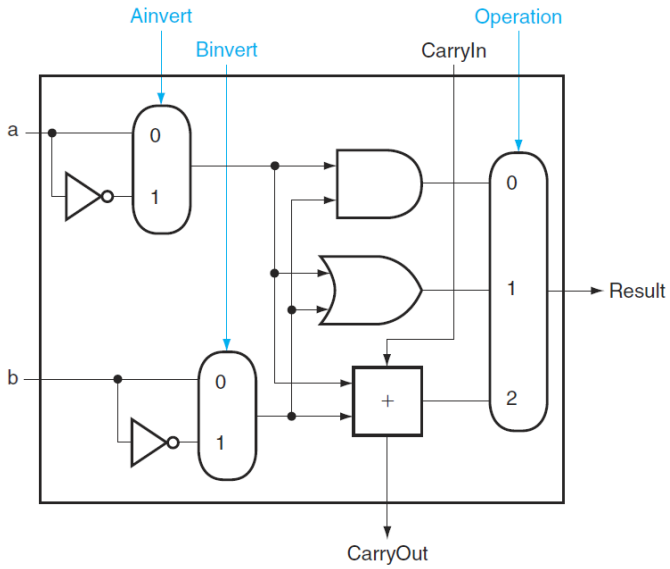
- a-b = a + (-b) = a + (2's complement of b)



- For SUB operation
  - Operation =
  - Binvert =
  - CarryIn =

How to add NOR operation?

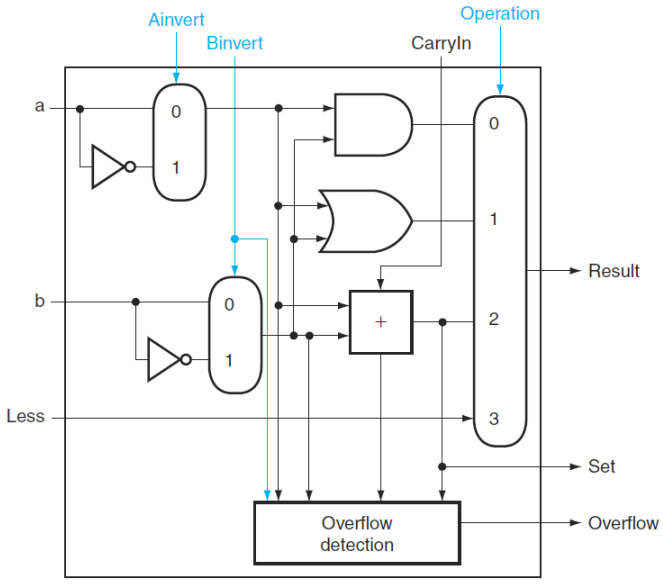
- $\overline{a+b} = \bar{a}.\bar{b}$



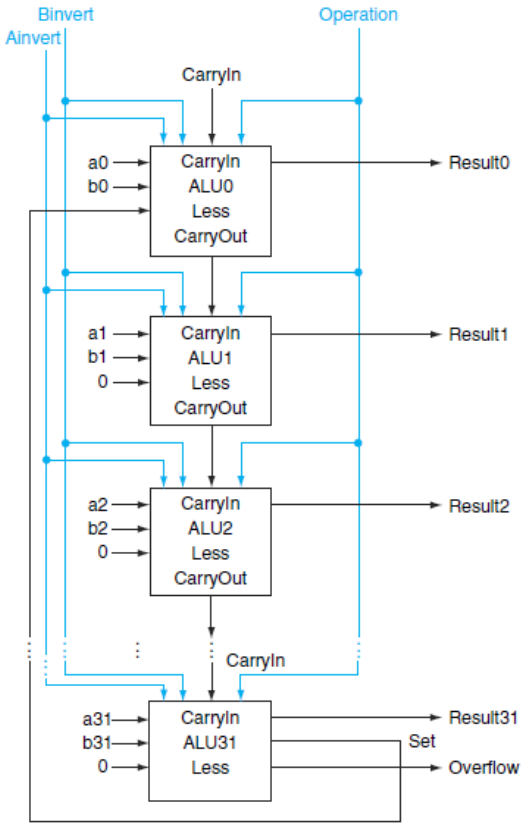
- Find control signal for NOR operation:

Adding SLT operation

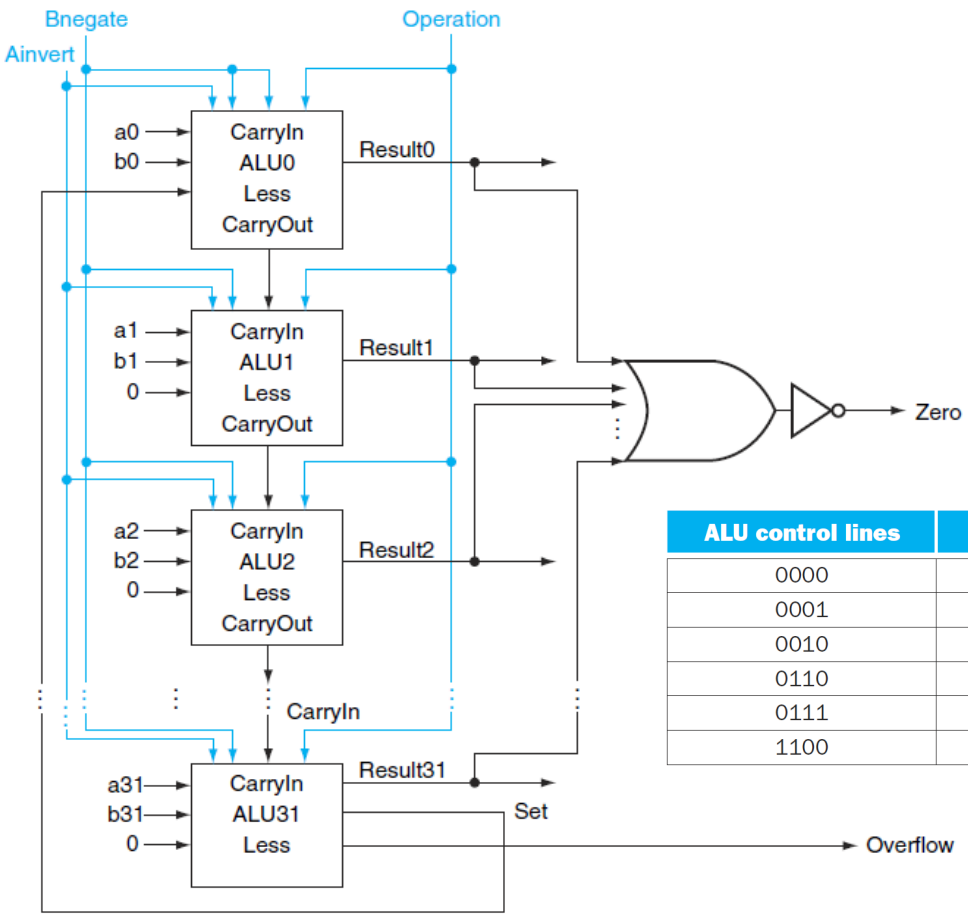
- Check highest bit of (a-b)
  - 1: a < b → set
  - 0: a >= b → clear



Building 32-bit ALU from 1-bit ALUs

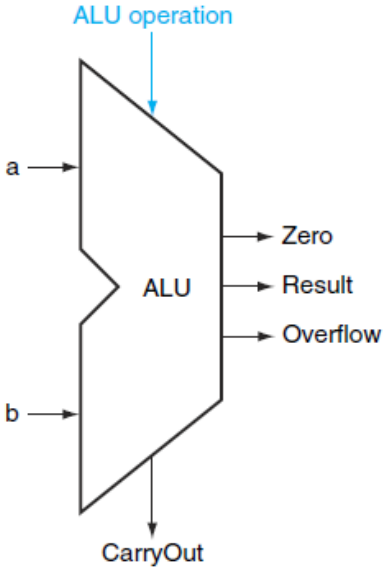


Final ALU with AND, OR, NOR, ADD, SUB, SLT



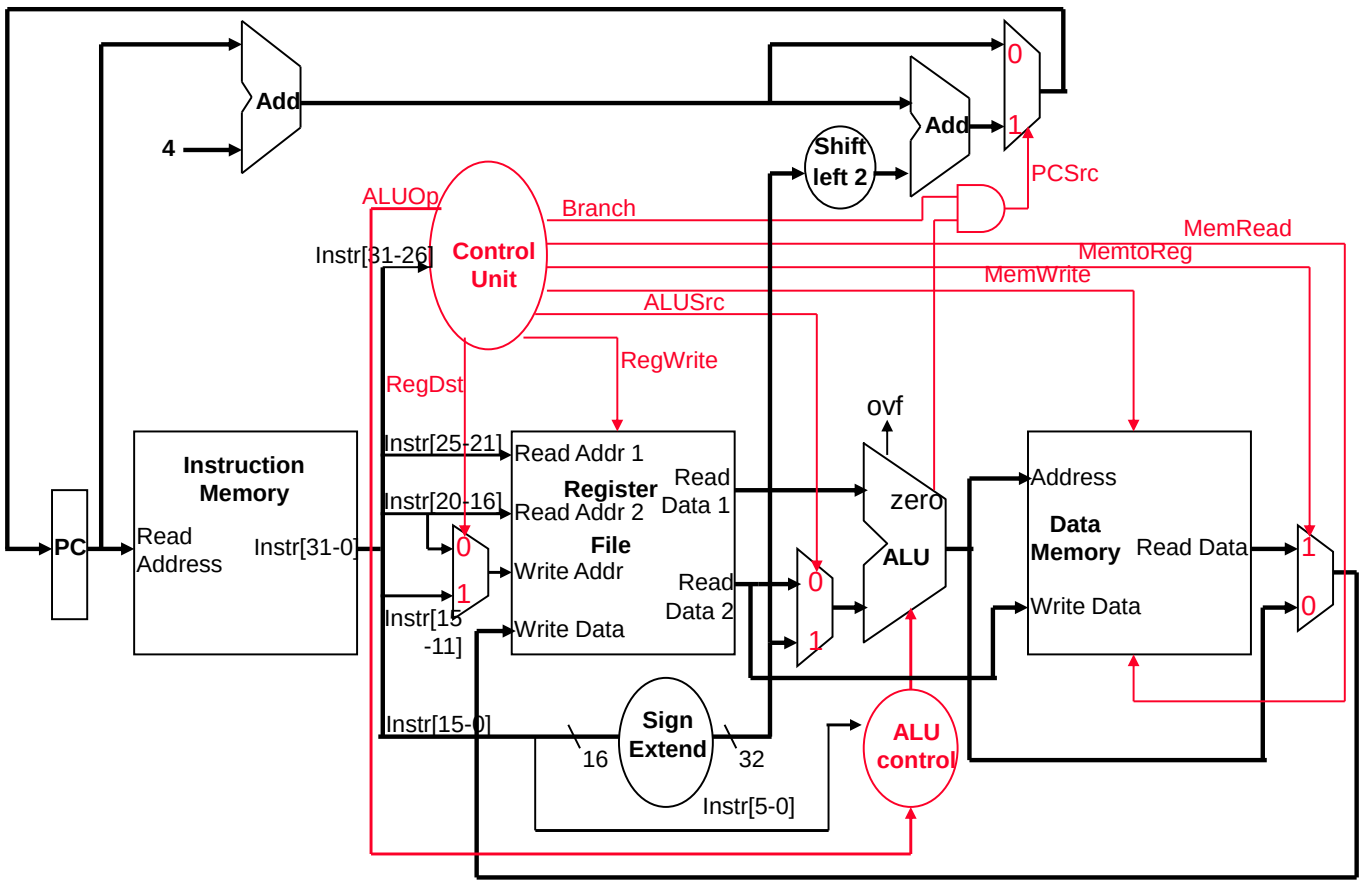
ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR

ALU symbol and interconnection

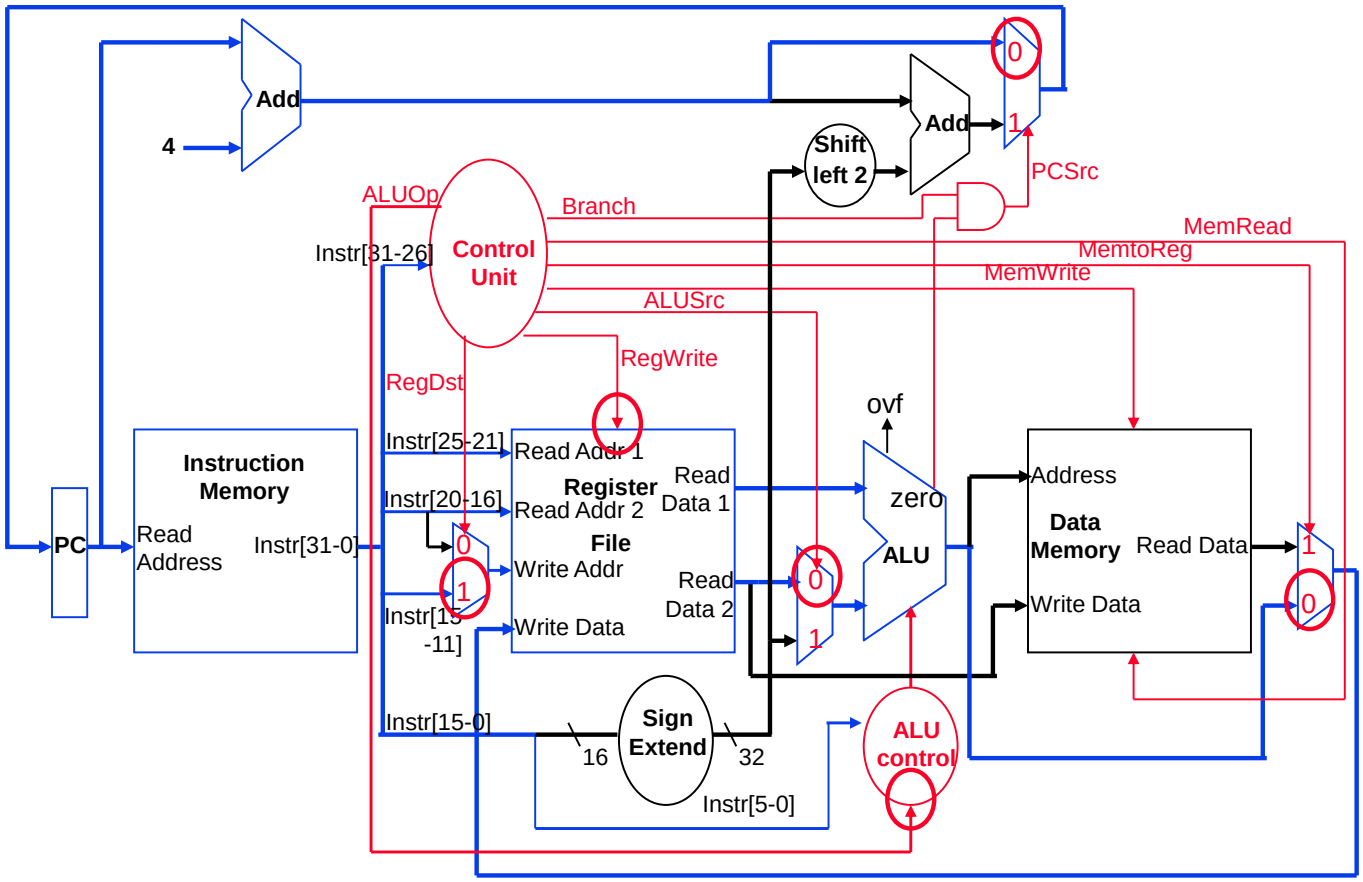


ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR

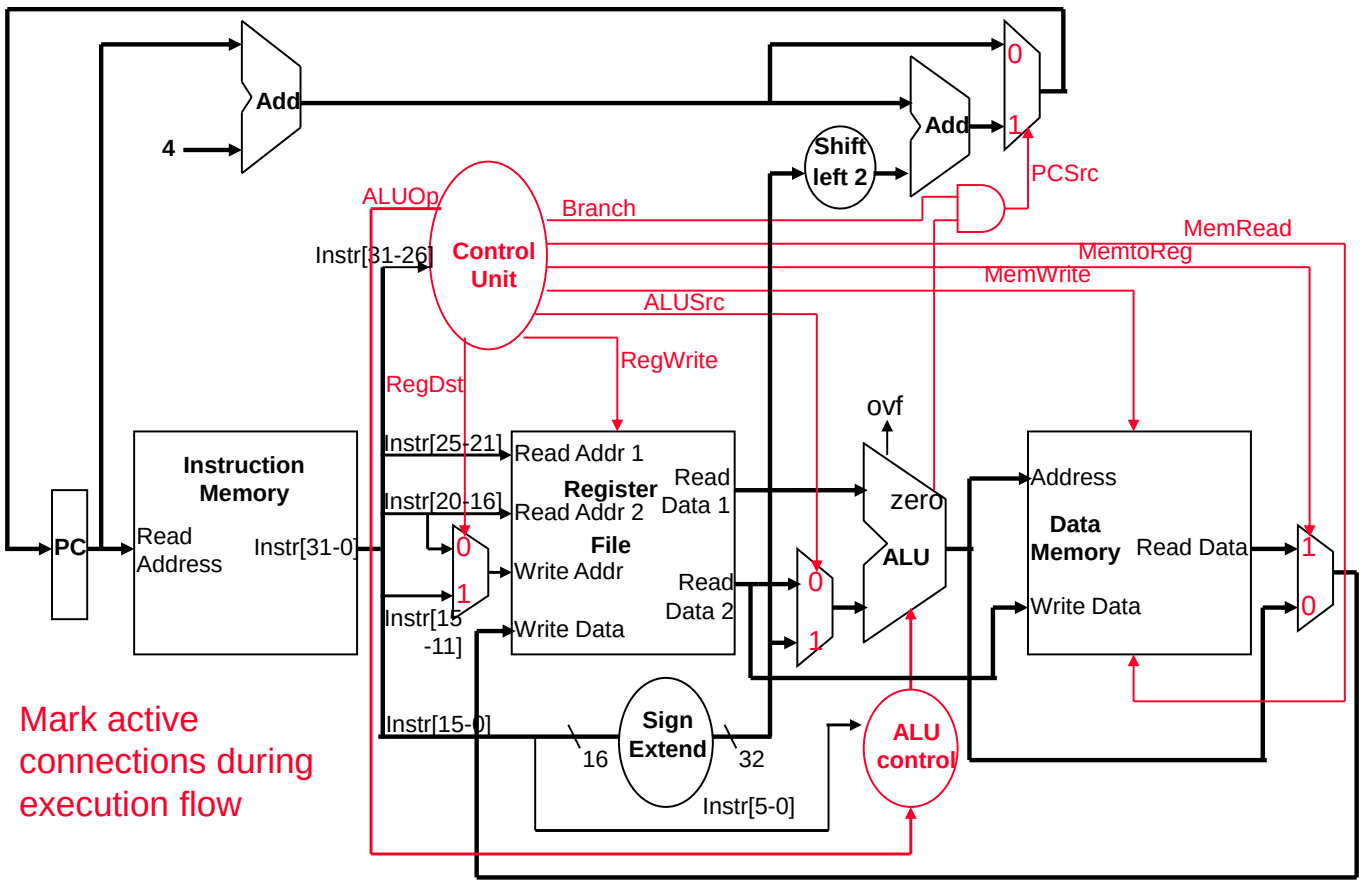
Adding Control Unit to build single datapath



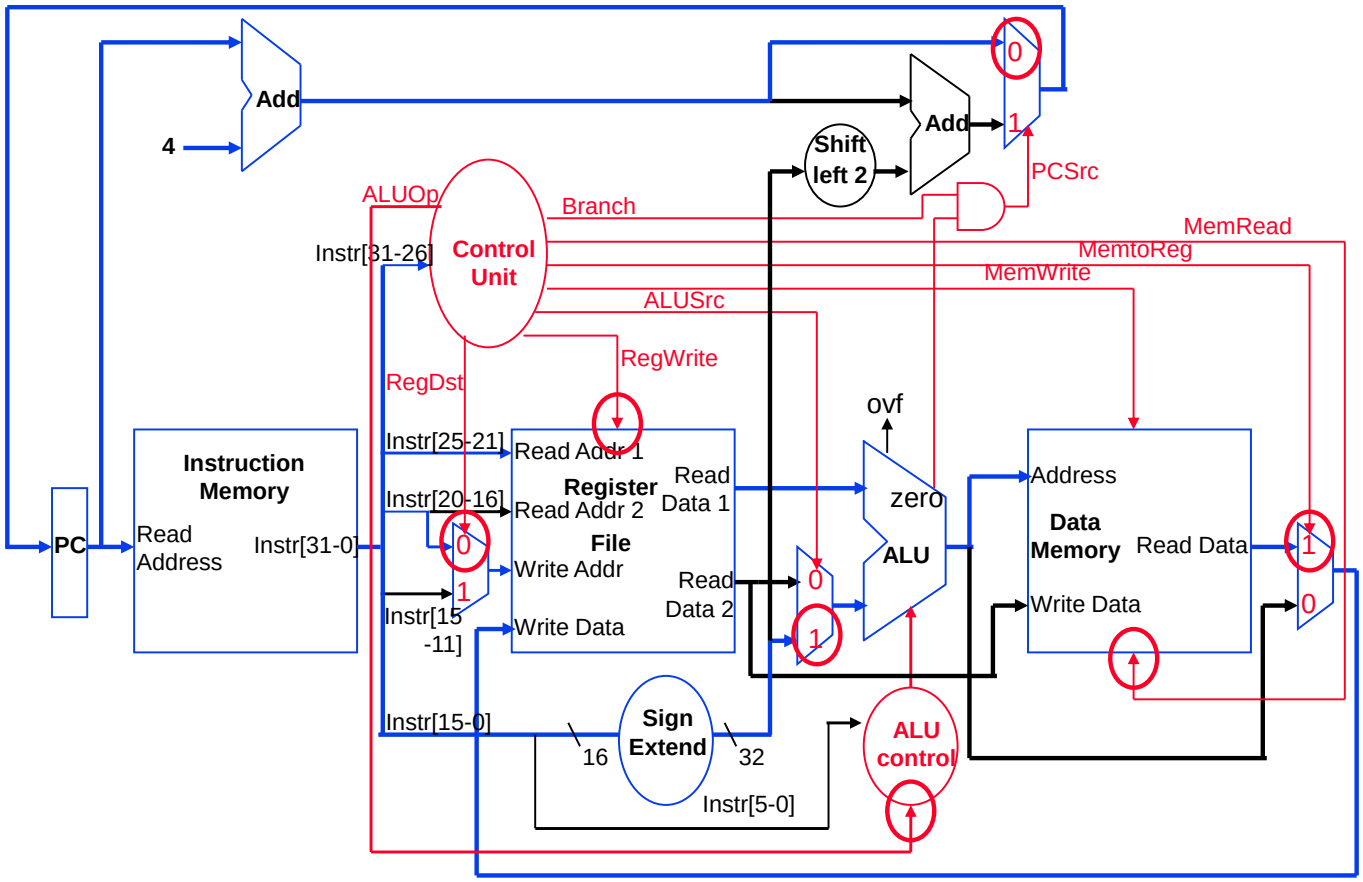
R-type Instruction Data/Control Flow



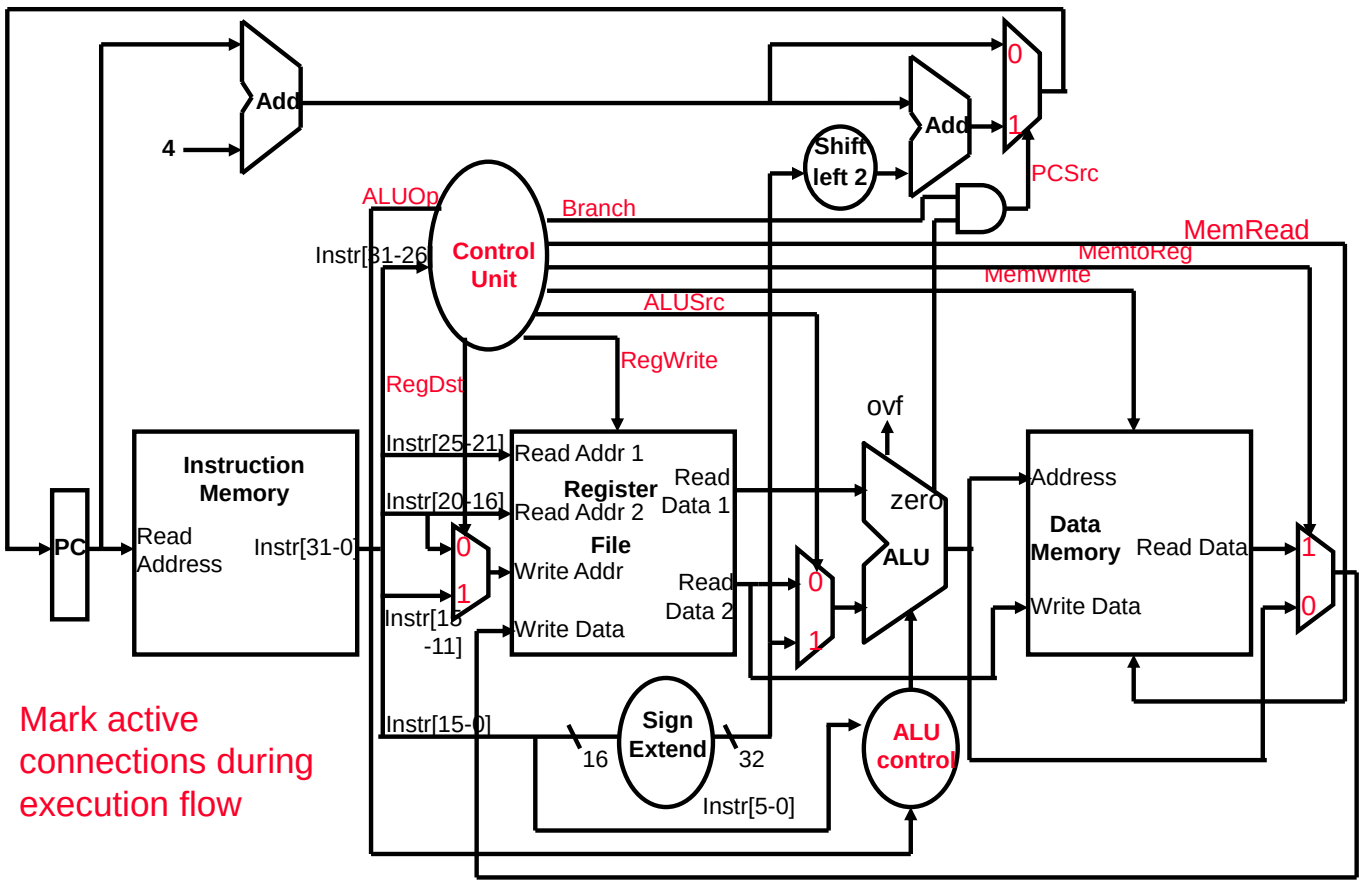
Load Word Instruction Data/Control Flow



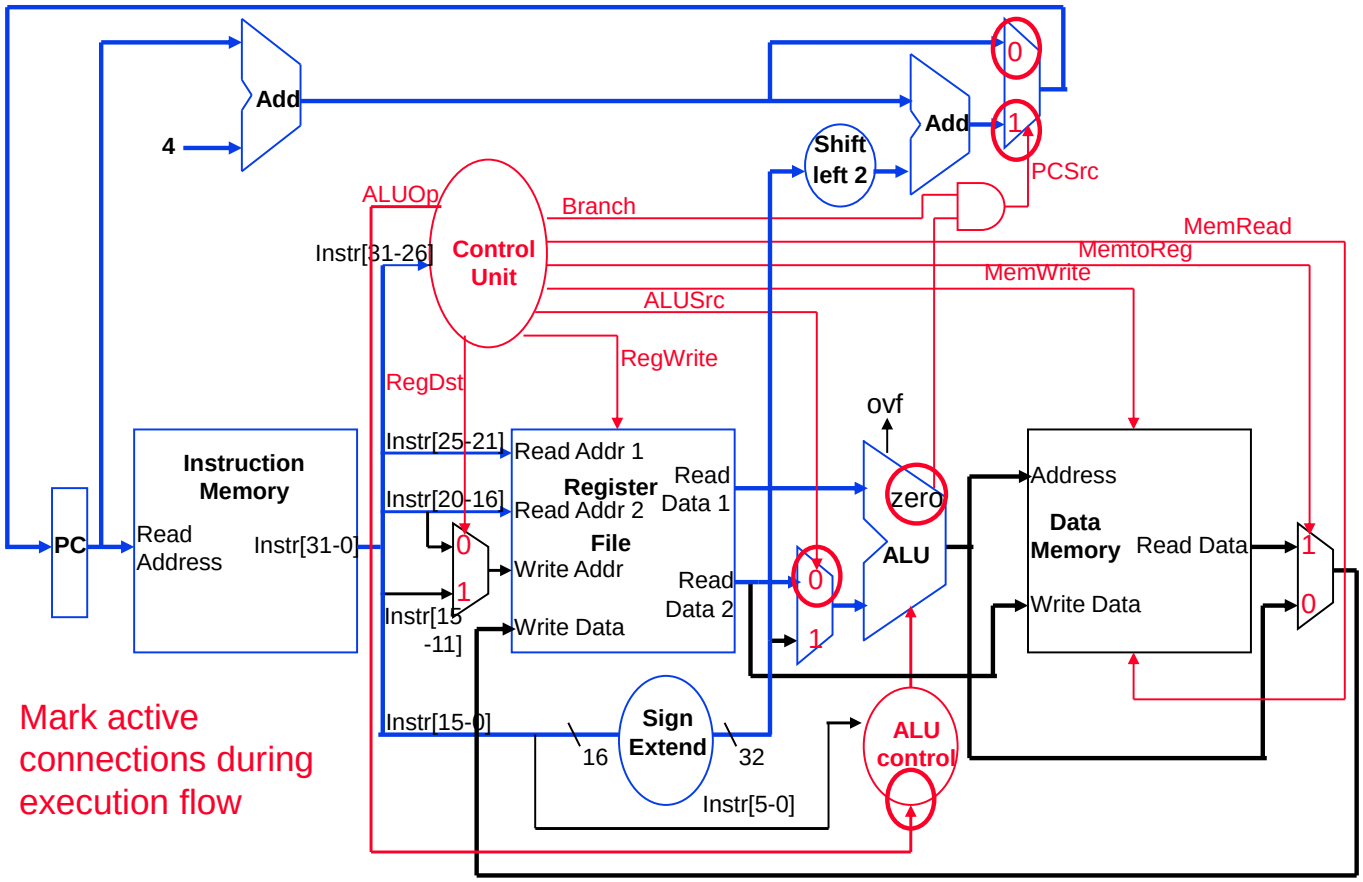
Load Word Instruction Data/Control Flow



Branch Instruction Data/Control Flow

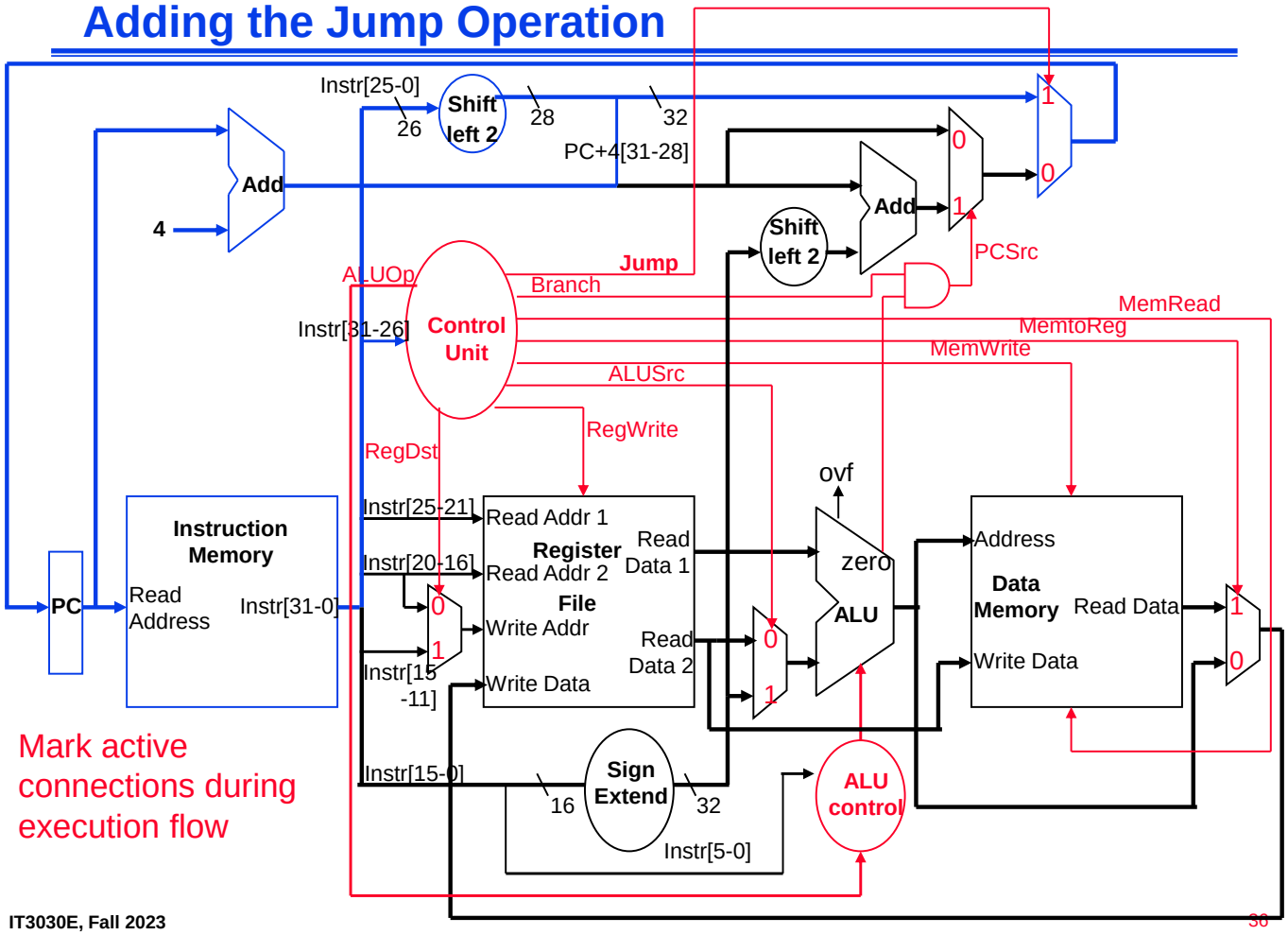


Branch Instruction Data/Control Flow



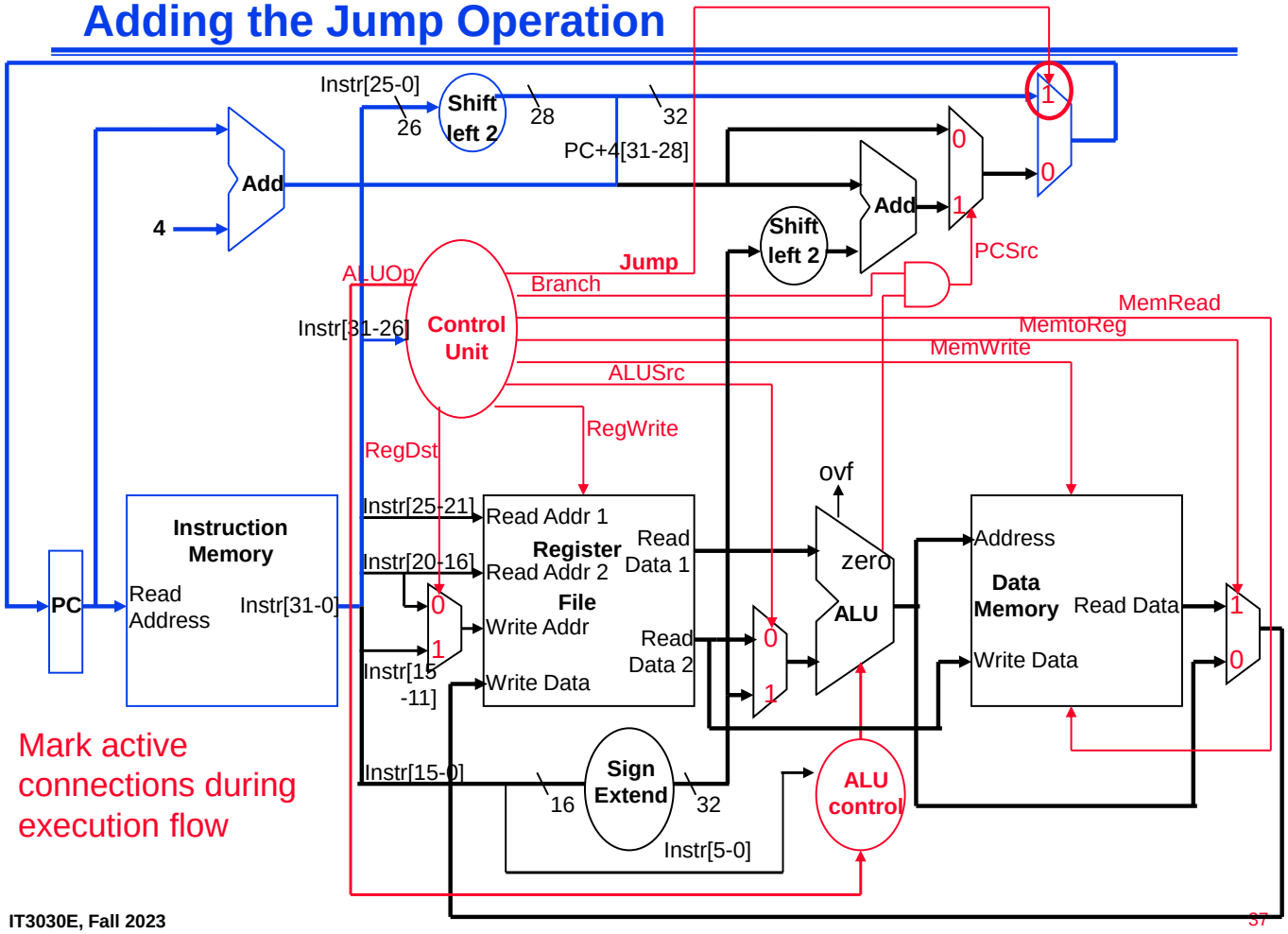
Mark active connections during execution flow

Adding the Jump Operation



Mark active connections during execution flow

Adding the Jump Operation



Mark active connections during execution flow

Instruction Times (Critical Paths)

- What is the clock cycle time assuming negligible delays for muxes, control unit, sign extend, PC access, shift left 2, wires, setup and hold times except:
  - Instruction and Data Memory (200 ps)
  - ALU and adders (200 ps)
  - Register File access (reads or writes) (100 ps)

Instr.	I Mem	Reg Rd	ALU Op	D Mem	Reg Wr	Total
R-type						
load						
store						
beq						
jump						

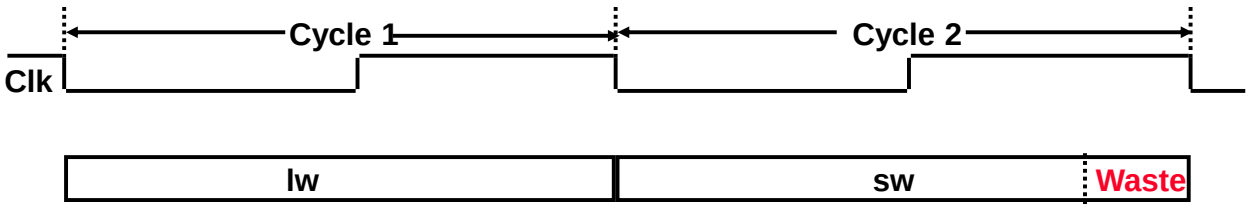
Instruction Critical Paths for Single cycle CPU

- What is the clock cycle time assuming negligible delays for muxes, control unit, sign extend, PC access, shift left 2, wires, setup and hold times except:
  - Instruction and Data Memory (200 ps)
  - ALU and adders (200 ps)
  - Register File access (reads or writes) (100 ps)

Instr.	I Mem	Reg Rd	ALU Op	D Mem	Reg Wr	Total
R-type	200	100	200		100	600
load	200	100	200	200	100	800
store	200	100	200	200		700
beq	200	100	200			500
jump	200					200

Single Cycle Disadvantages & Advantages

- Uses the clock cycle inefficiently – the clock cycle must be timed to accommodate the slowest instruction
  - especially problematic for more complex instructions like floating point multiply



- May be wasteful of area since some functional units (e.g., adders) must be duplicated since they can not be shared during a clock cycle

but

- Is simple and easy to understand

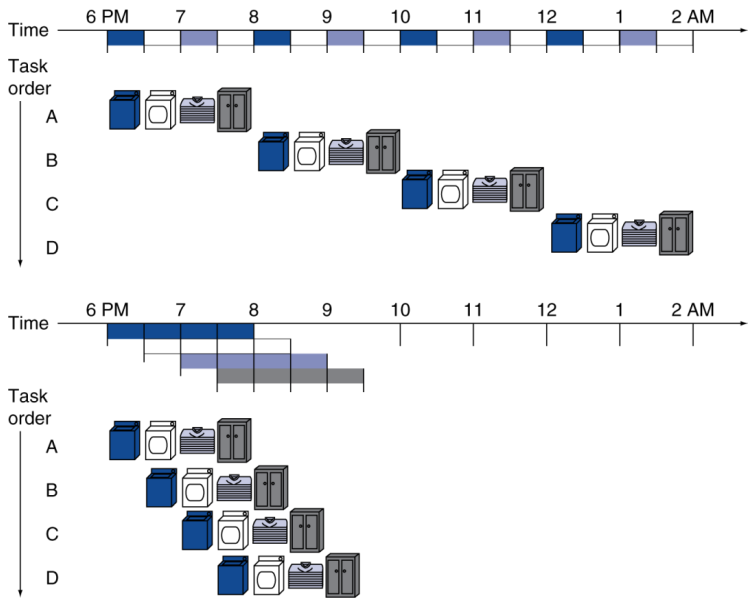
How Can We Make The Computer Faster?

- Divide instruction cycles into smaller cycles
- Executing instructions in parallel
  - With only one CPU?
- Pipelining:
  - Start fetching and executing the next instruction before the current one has completed
  - Overlapping execution

Pipeline in real life

A more serious example: laundry work

- Pipelined laundry boots performance up to 4 times



- With 4 loads

  T_{normal} = 4*2 = 8 hours

  $T_{pipeline}$ = 3.5 hours

- With n loads

  T_{normal} = n*2 hours

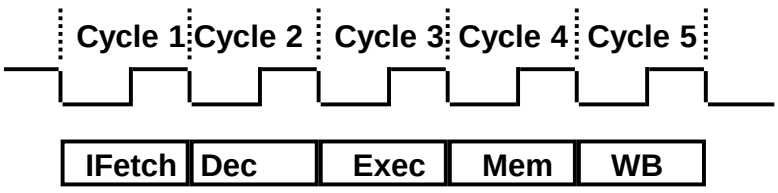
  $T_{pipeline}$ = (3+n)/2 hours

4 stages: washing, drying, ironing, folding

When n → ∞ : T_{normal} → 4*$T_{pipeline}$

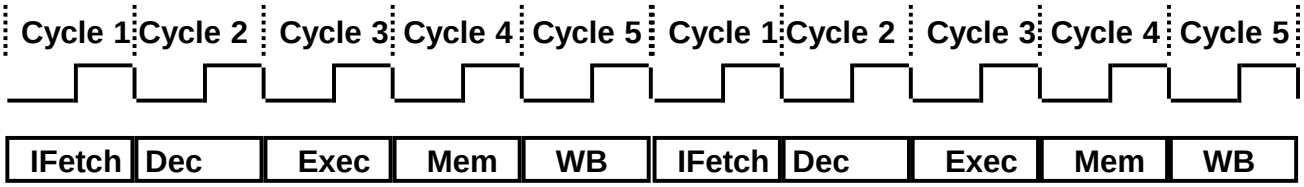
MIPS Pipeline

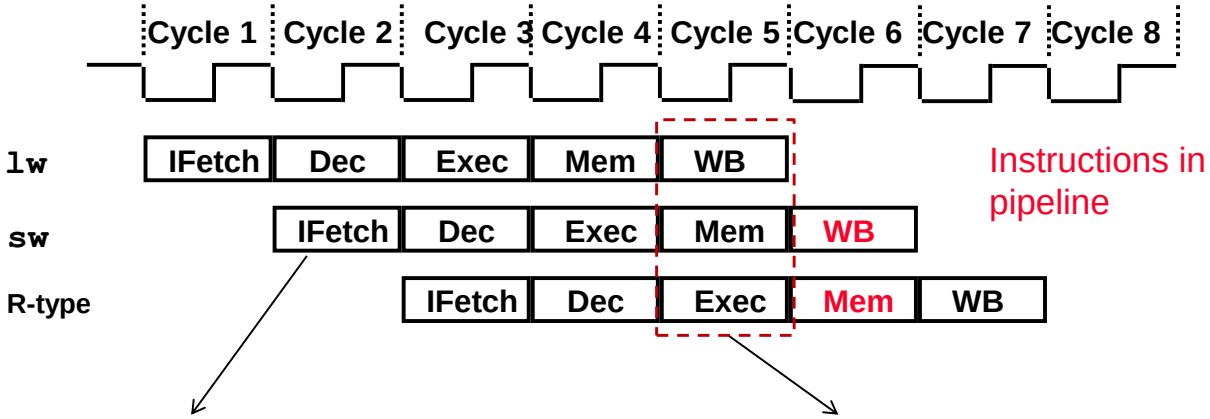
- Five stages, one step per stage
  - IFetch: Instruction Fetch and Update PC
  - Dec: Registers Fetch and Instruction Decode
  - Exec: Execute R-type; calculate memory address
  - Mem: Read/write the data from/to the Data Memory
  - WB: Write the result data into the register file



Execution time for a single instruction is always 5 cycles, regardless of instruction operation

Instruction pipeline

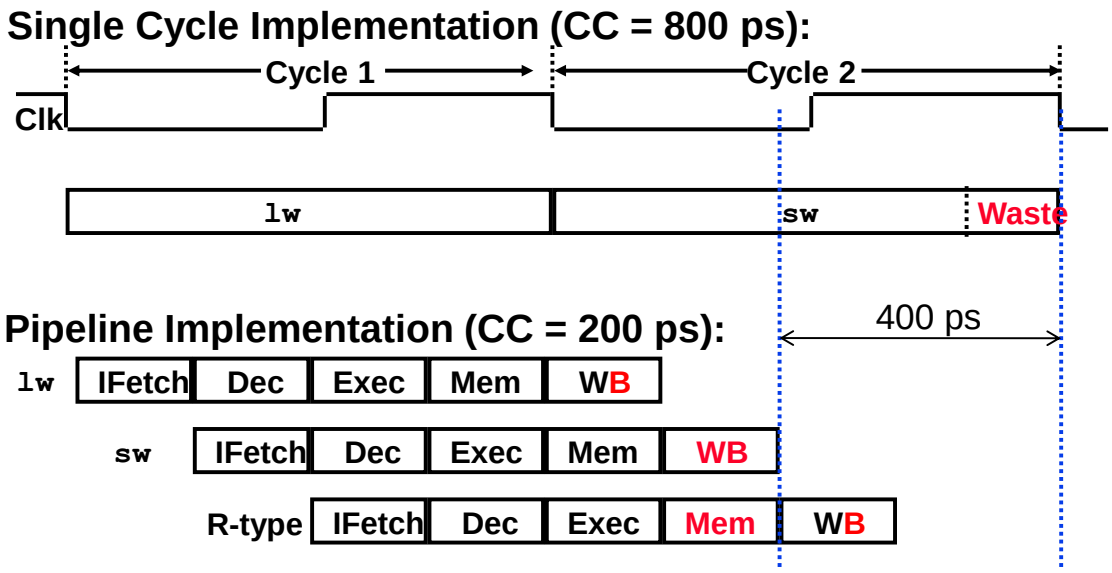




Start fetching and executing the next instruction before the current one has completed

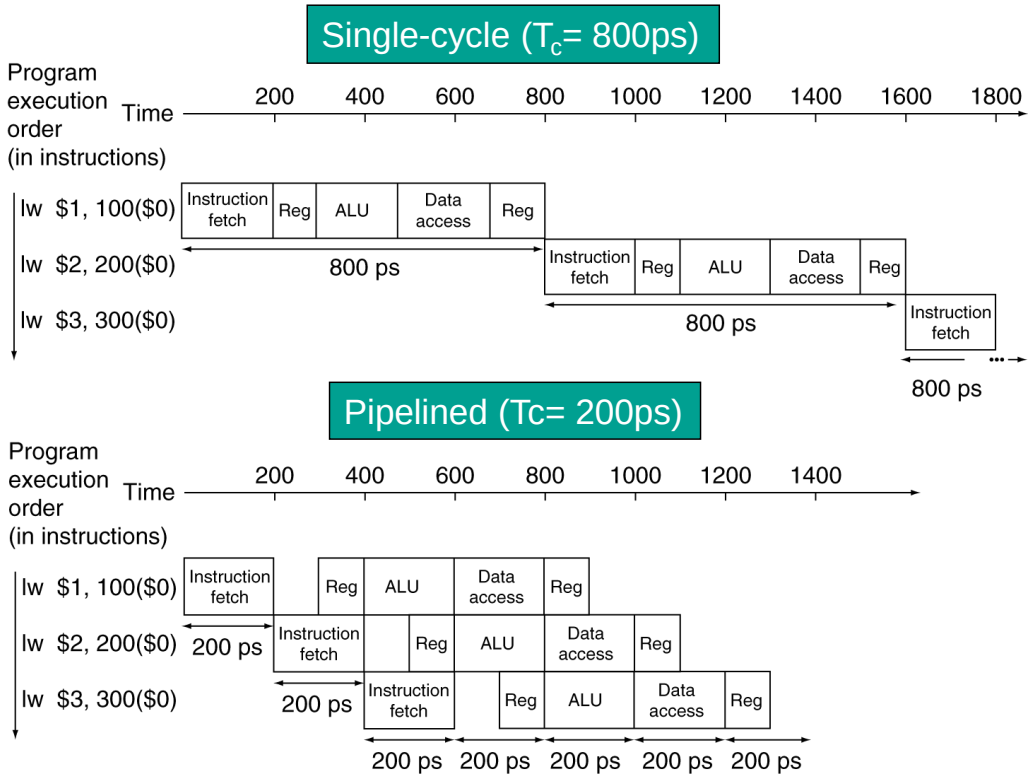
More than one instruction are executed at a time

Single Cycle versus Pipeline



- To complete an entire instruction in the pipelined case takes 1000 ps (as compared to 800 ps for the single cycle case). Why ?
- How long does each take to complete 1,000,000 adds ?

Example with lw instructions



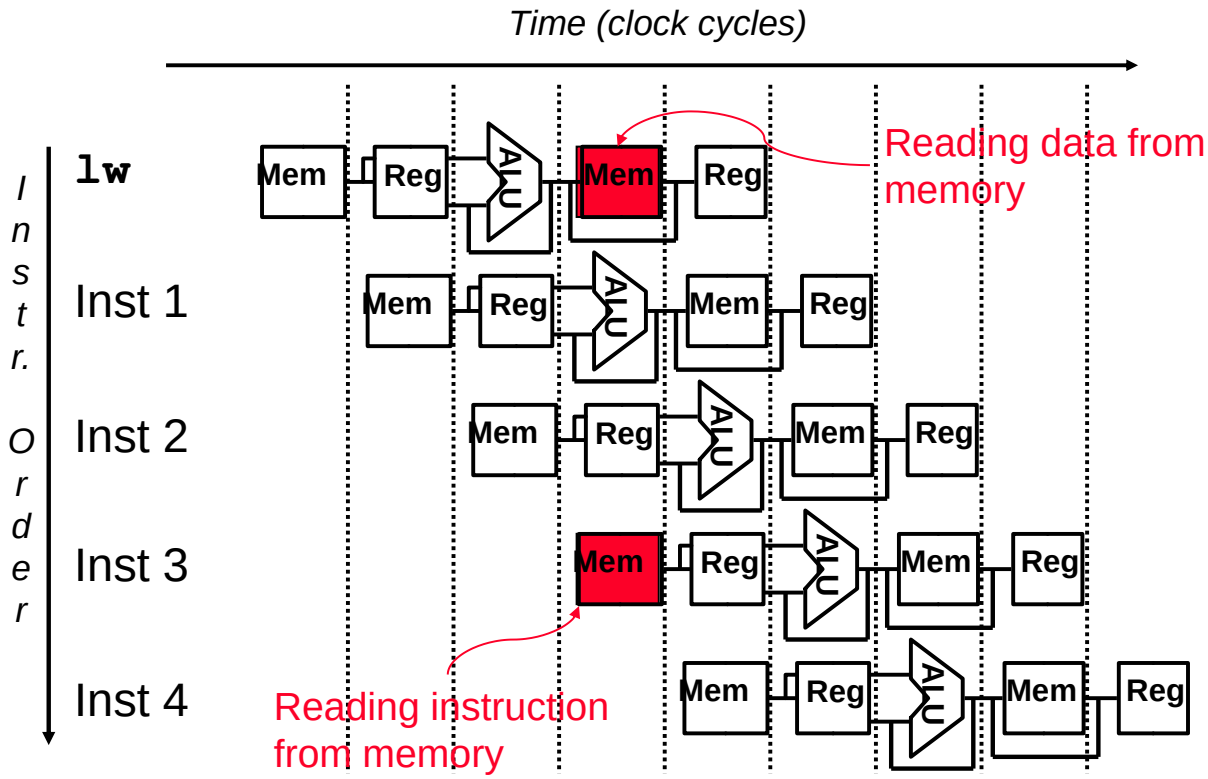
Pipeline hazards

- Pipeline can lead us into troubles!!!
- Hazards: situations that prevent starting the next instruction in the next cycle
  - structural hazards: attempt to use the same resource by two different instructions at the same time
  - data hazards: attempt to use data before it is ready
    - An instruction's source operand(s) are produced by a prior instruction still in the pipeline
  - control hazards: attempt to make a decision about program control flow before the condition has been evaluated and the new PC target address calculated
    - branch and jump instructions, exceptions
- In most cases, hazard can be solved simply by waiting
  - but we need better solutions to take advantages of pipeline

Structural hazard

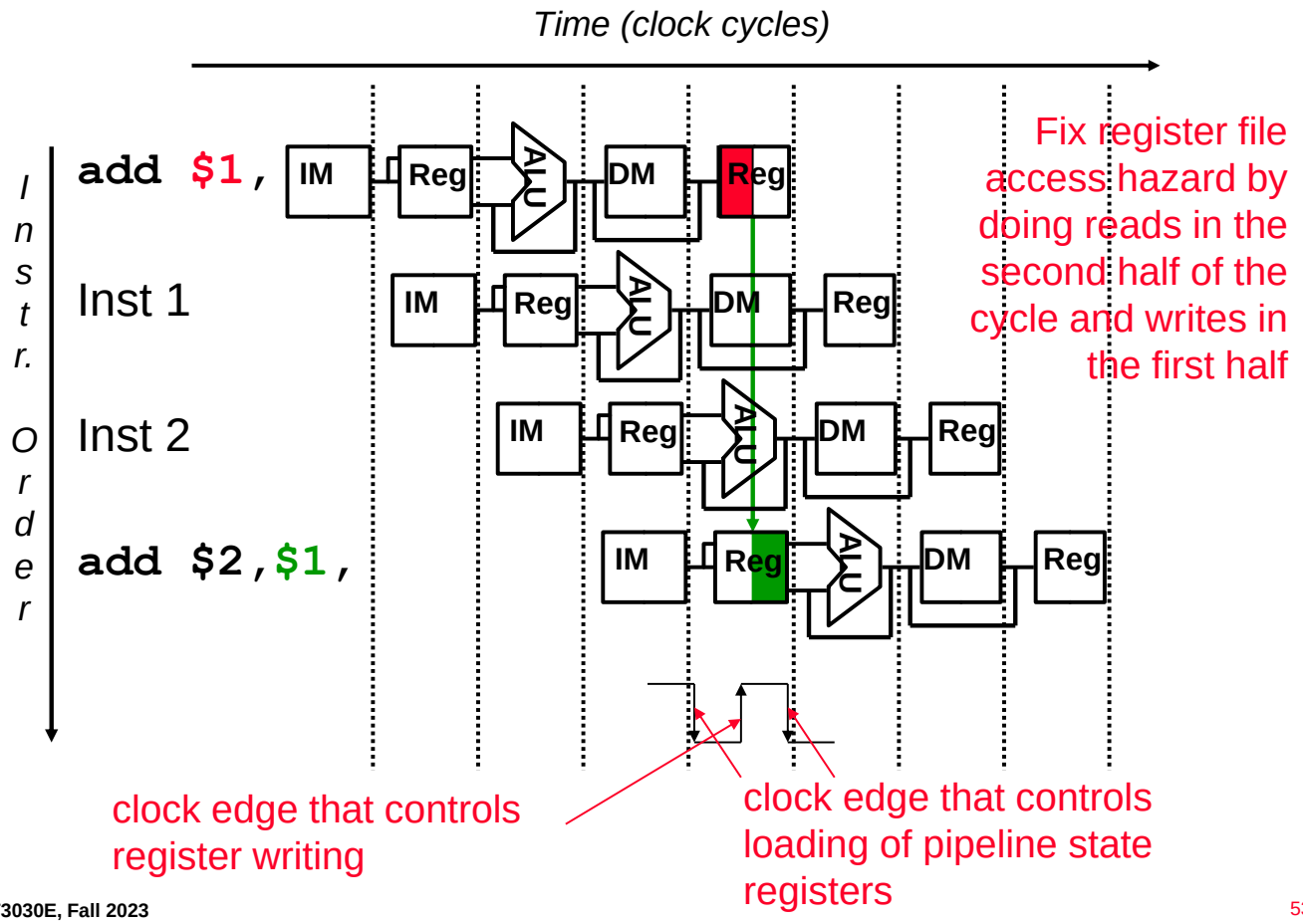
- Conflict for use of a resource
- In MIPS pipeline with a single memory
  - Load/store requires data access
  - Instruction fetch would have to *stall* for that cycle
    - Would cause a pipeline “bubble”
- Hence, pipelined datapath require separate instruction/data memories
  - Or separate instruction/data caches

A Single Memory Would Be a Structural Hazard



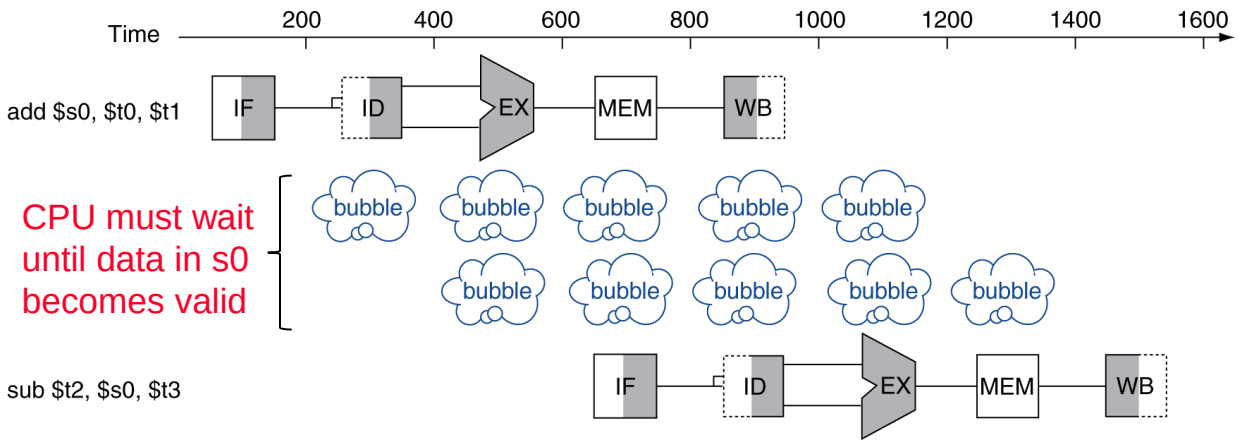
- Fix with separate instr and data memories (I$ and D$)

How About Register File Access?



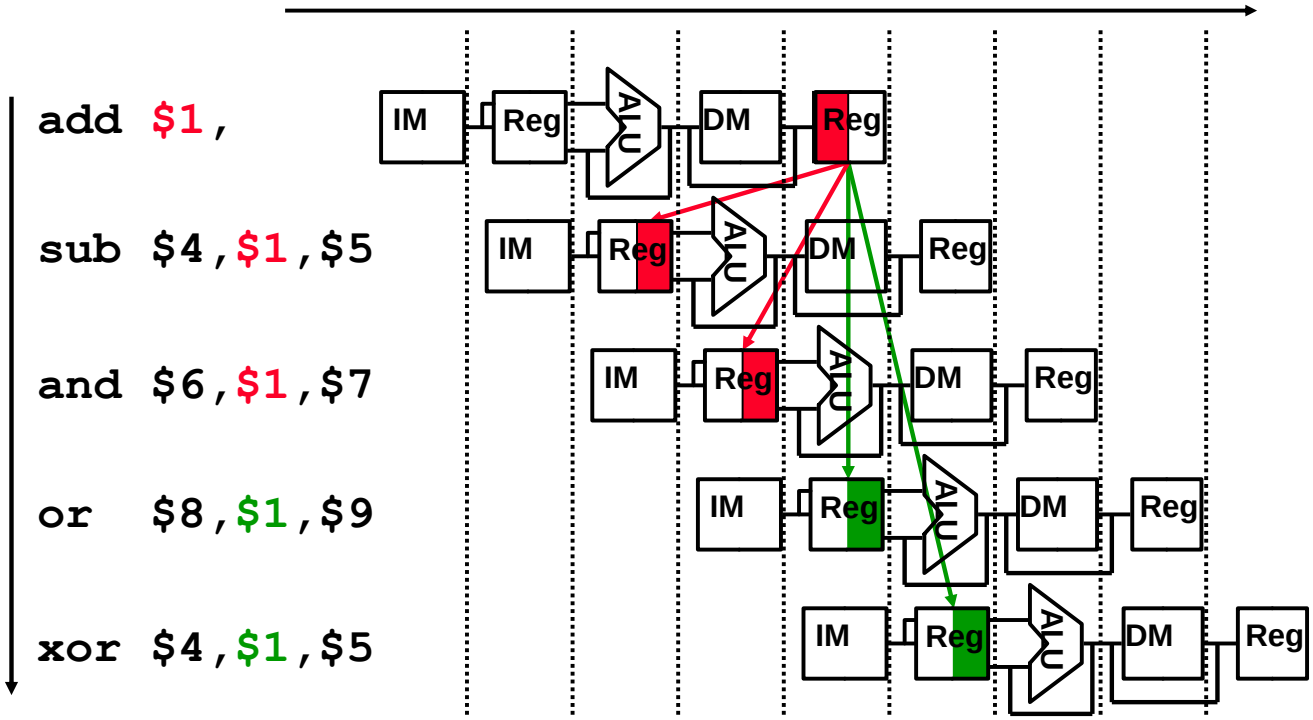
Data hazard

- An instruction depends on completion of data access by a previous instruction
  - `add $s0, $t0, $t1`
    `sub $t2, $s0, $t3`



Example

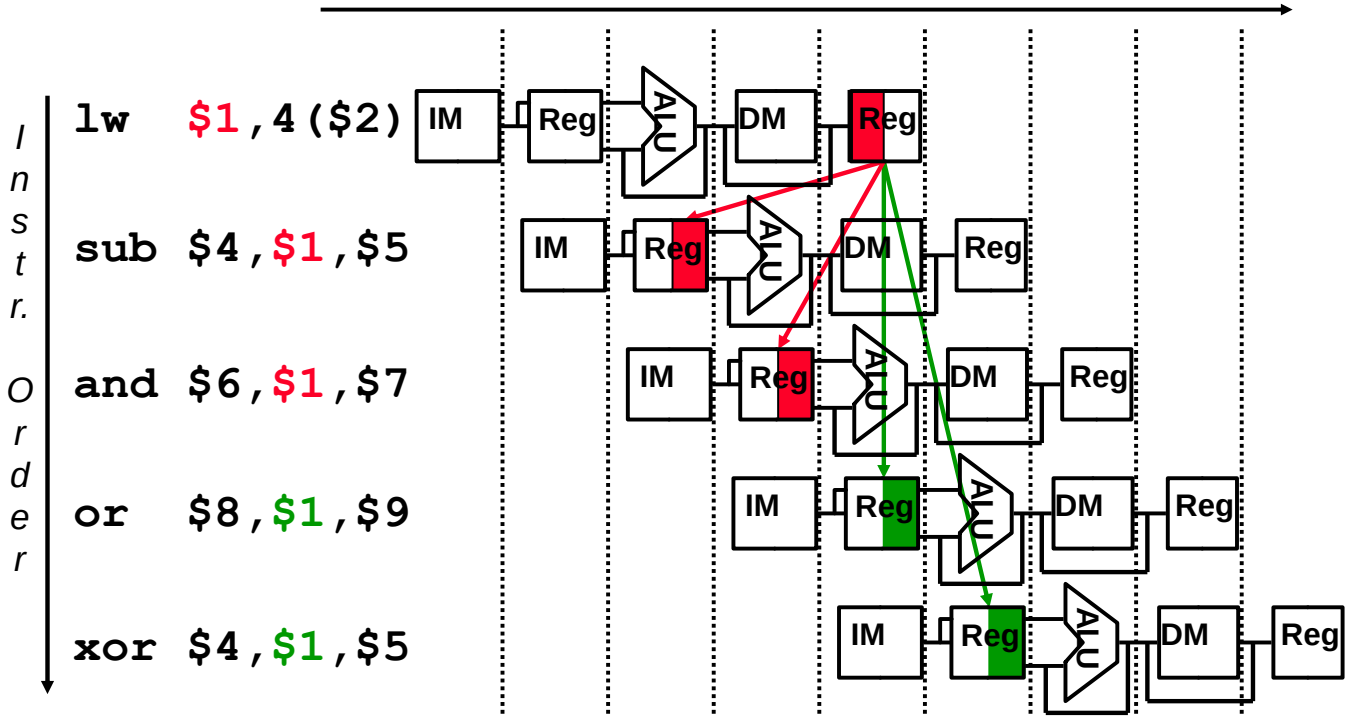
- Dependencies backward in time cause hazards



- Read before write data hazard

Example

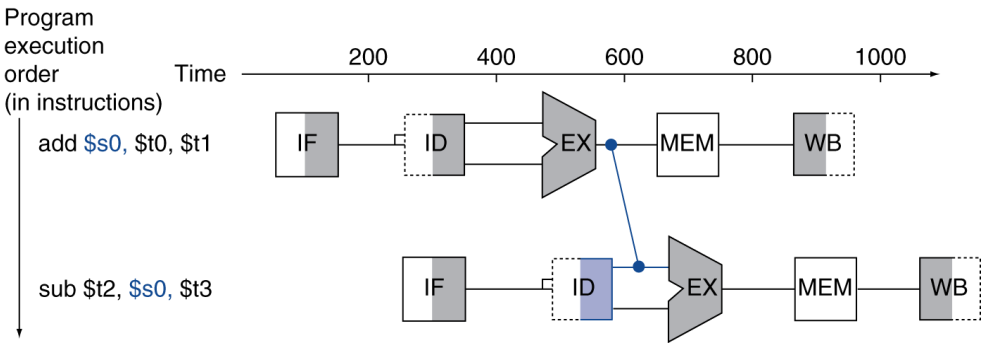
- Dependencies backward in time cause hazards



- Load-use data hazard

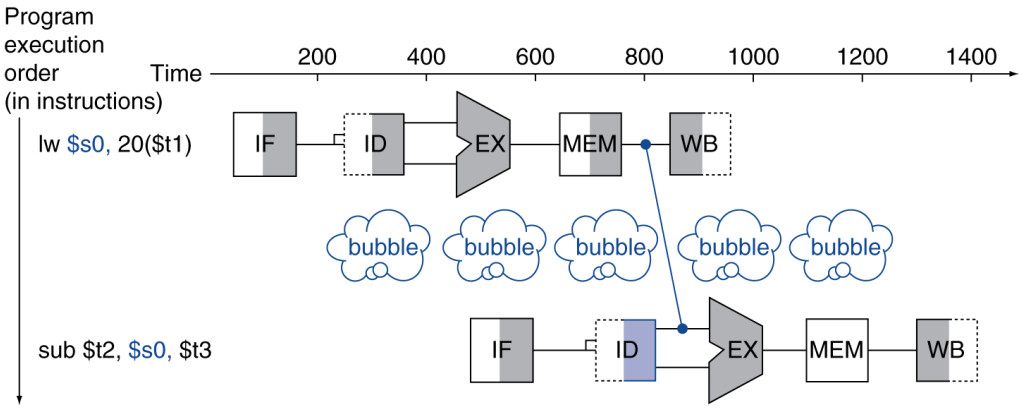
Solving hazard with forwarding

- Use result when it is computed
  - Don't wait for it to be stored in a register
  - Requires extra connections in the datapath
- Forward from EX to EX (output to input)



Load-Use Data Hazard

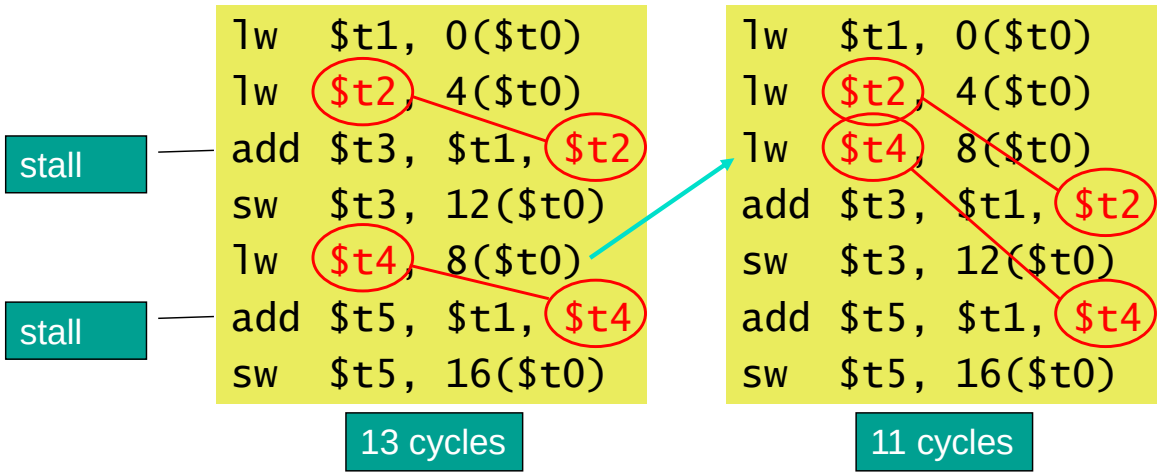
- One cycle stall is necessary
- Forward from MEM (output) to EX (input)



Code Scheduling to Avoid Stalls

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

```
A = B + E;
C = B + F;
```

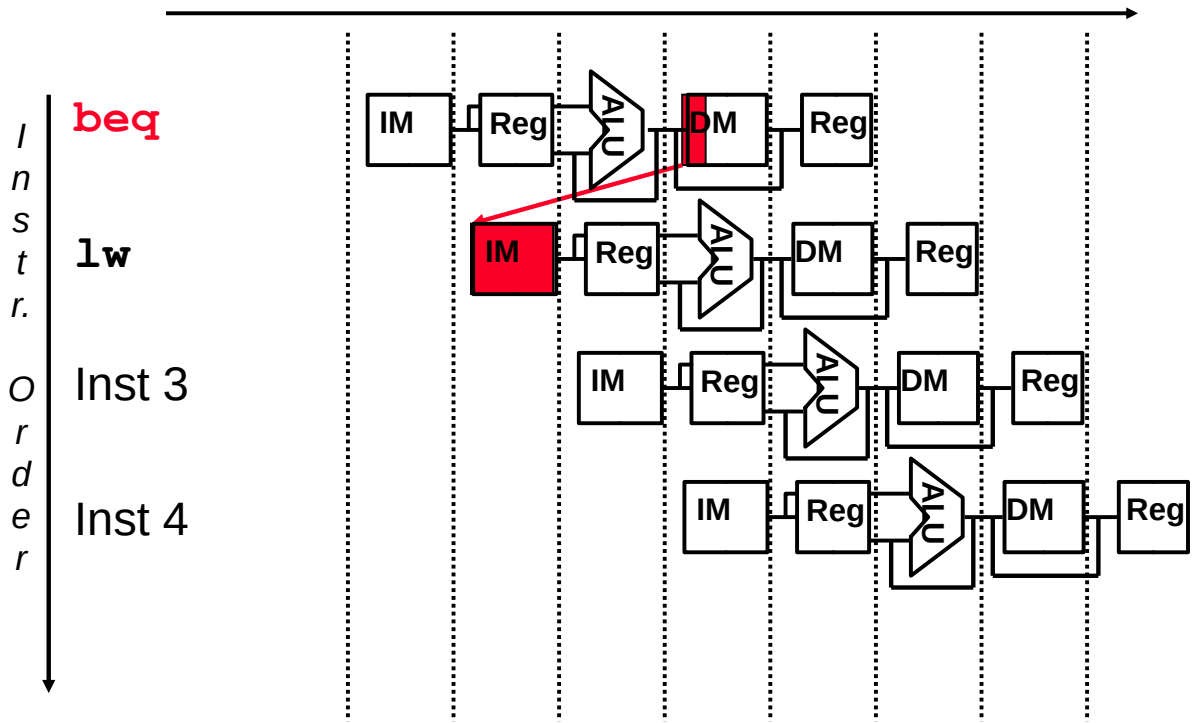


Control Hazards

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

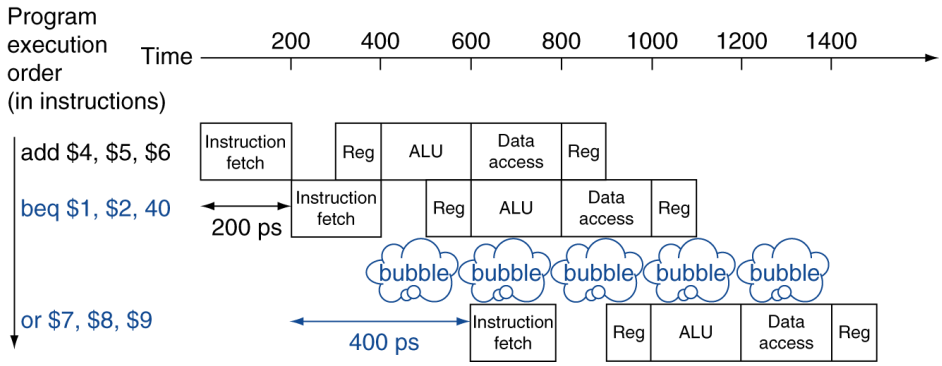
Branch Instructions Cause Control Hazards

- Dependencies backward in time cause hazards



Stall on Branch

- Naïve approach: Wait until branch outcome determined before fetching next instruction



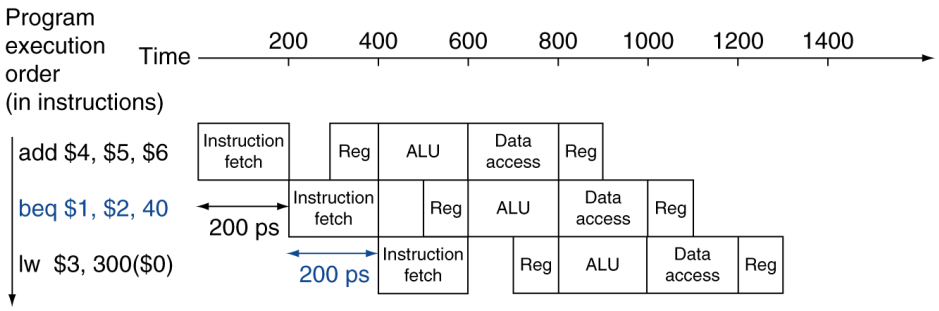
Performance affect: assume that 17% of instructions in program are branches, if each branch take one cycle for the stall, then performance will be 17% slower. (CPI = 1.17)

Branch Prediction

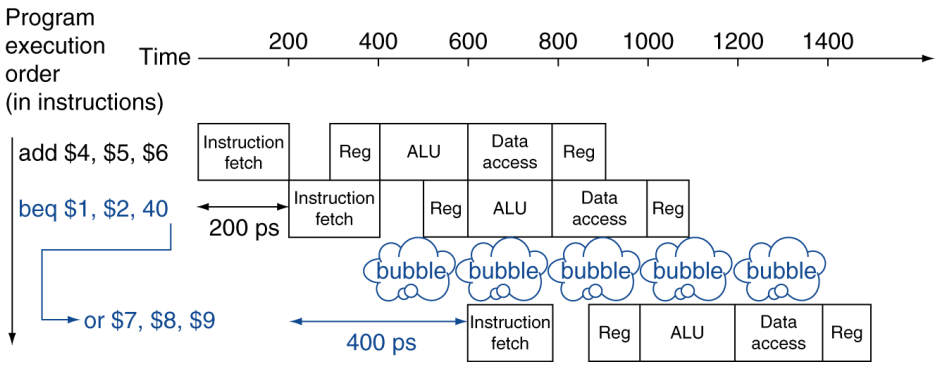
- Predict outcome of branch
- Only stall if prediction is wrong
- In MIPS pipeline
  - Can predict branches not taken
  - Fetch instruction after branch, with no delay

MIPS with Predict Not Taken

Prediction correct



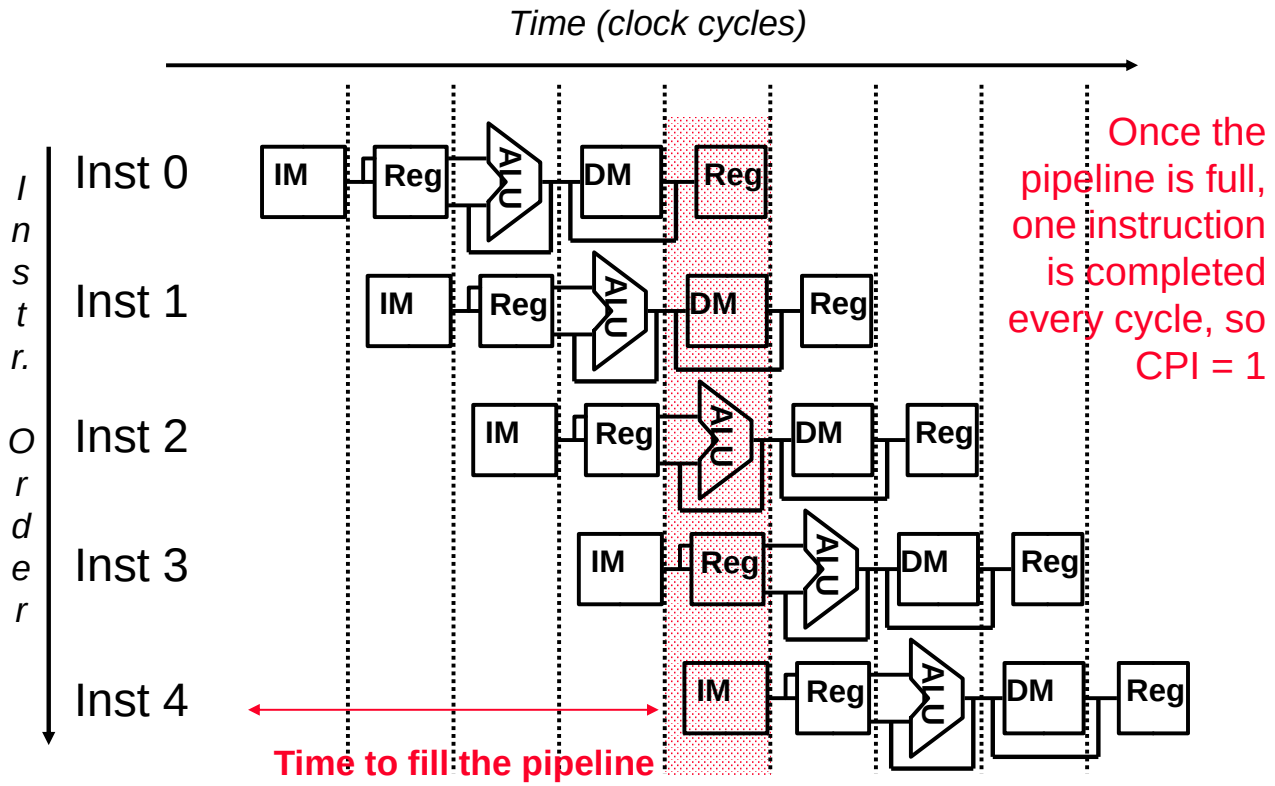
Prediction incorrect



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
  - As good as > 90% accuracy

Summary: Pipeline Operation



Exercise 1

Assume that the following instructions are executed in a 5-stage-pipelined MIPS CPU. Draw the timeline of each intruction

IF ID EX MEM WB

add $s0, $s1, $s2

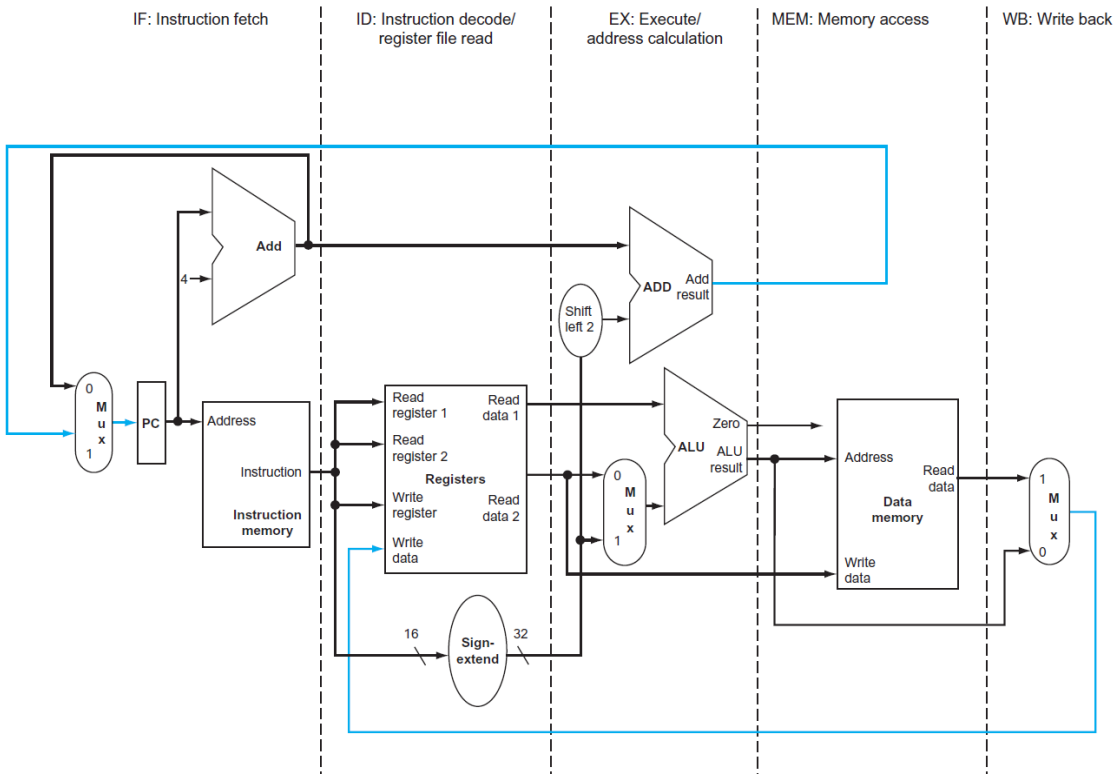
lw $t0, 0($t1)

sw $t2, 0($t3)

bne $s0, $s1, EXIT

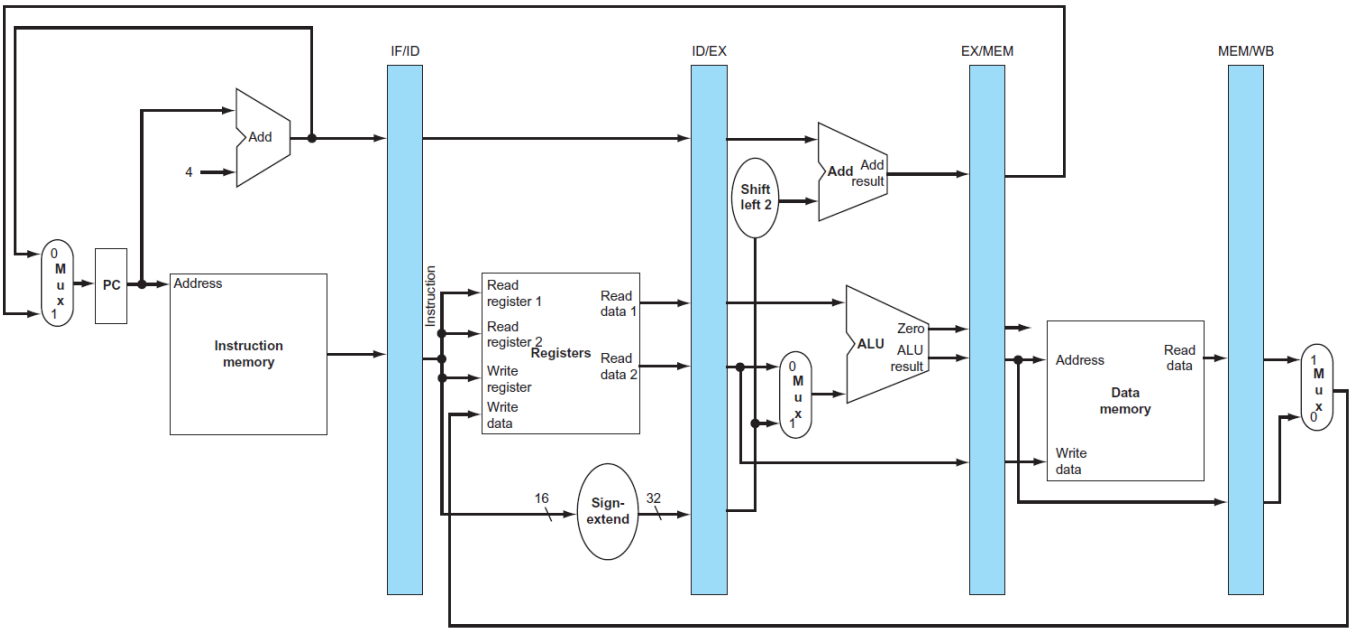
Pipelined datapath

- How to share/isolate data between different stages?



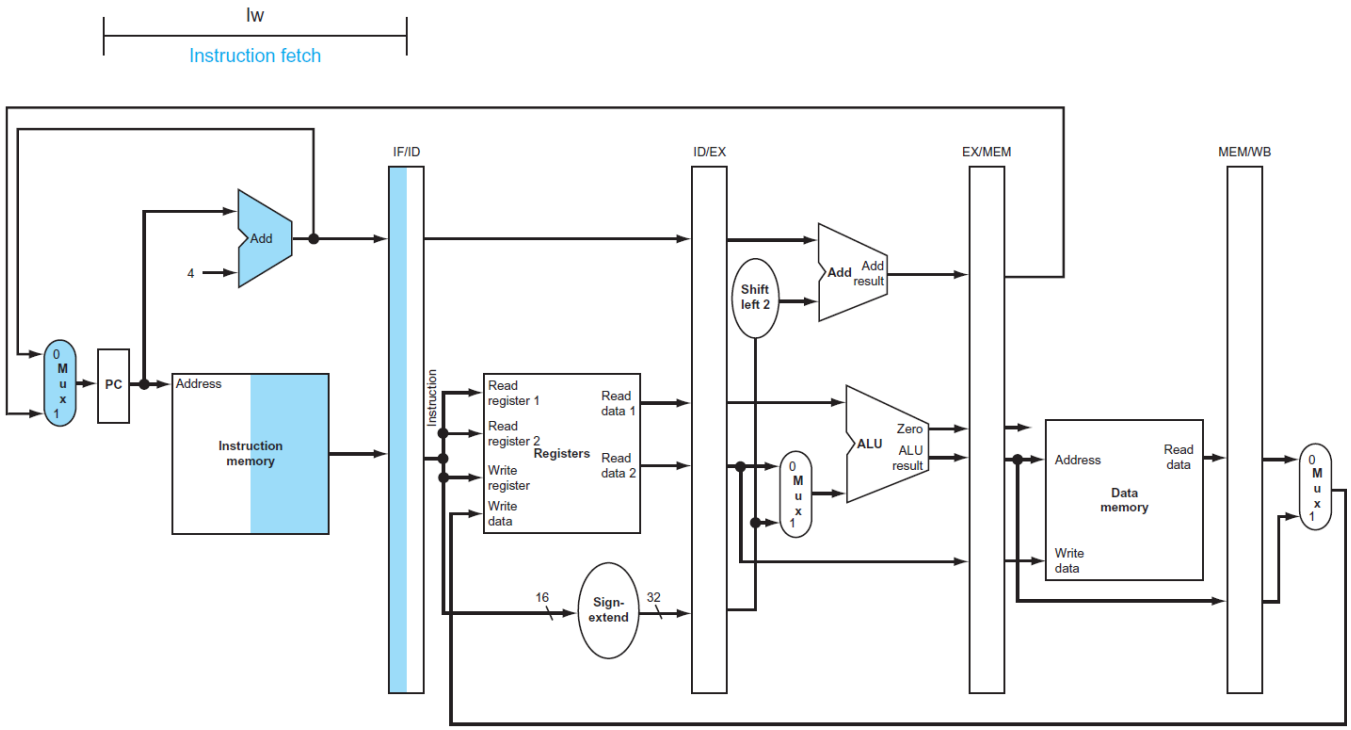
Pipelined datapath

- Adding pipeline registers

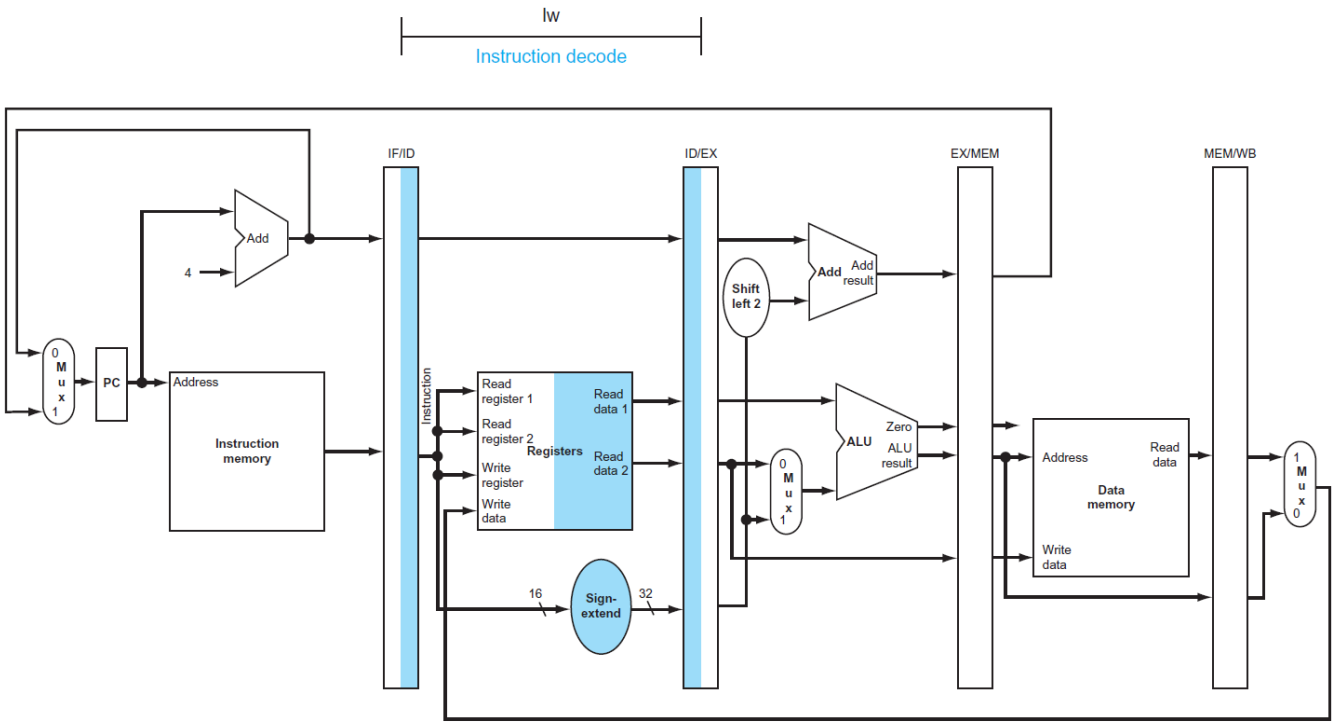


- 5 stages, why only 4 registers?

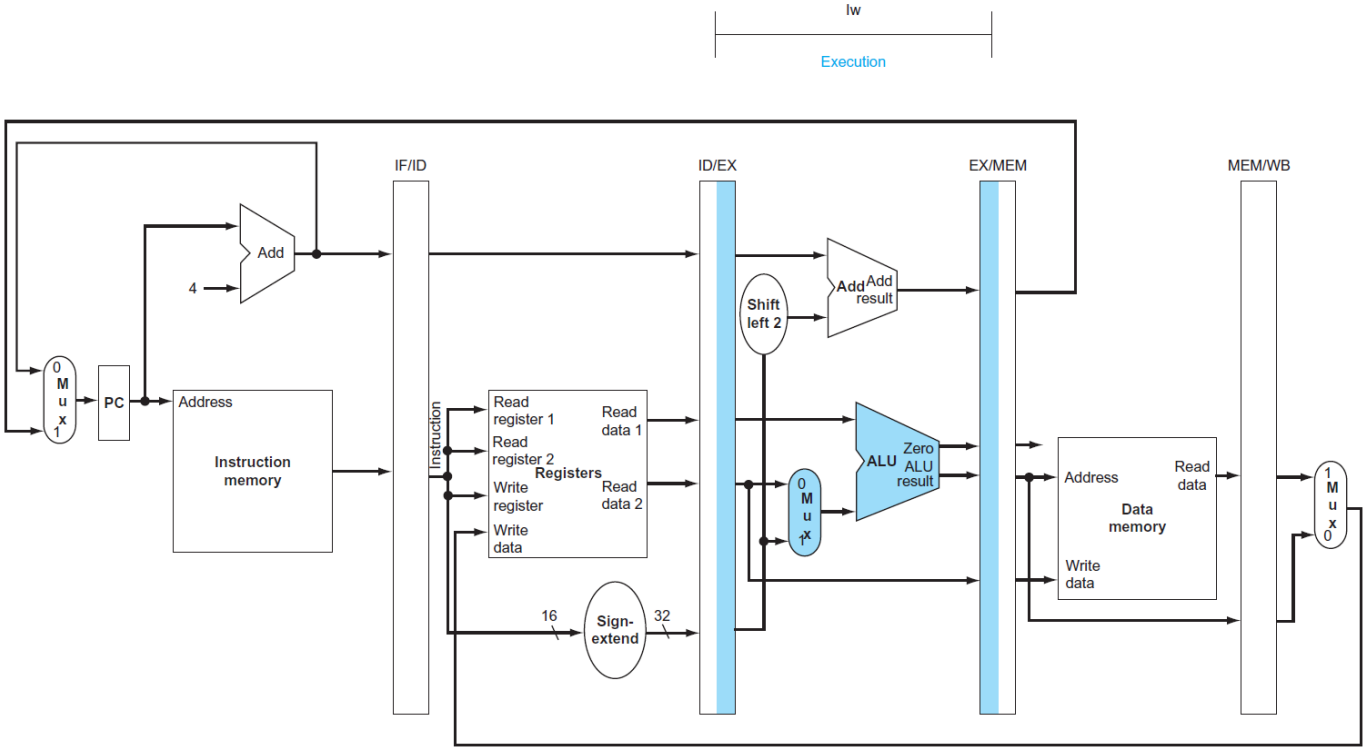
Instruction fetch in pipeline



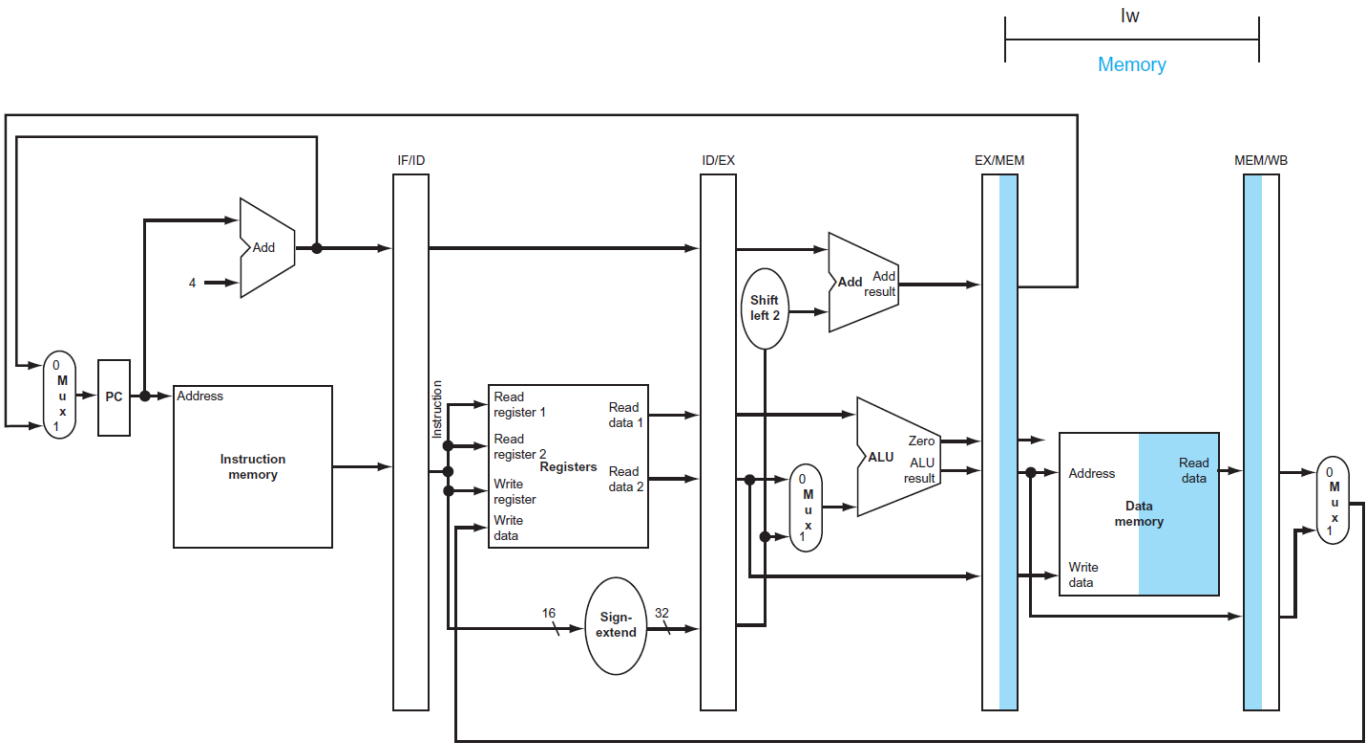
Instruction decode



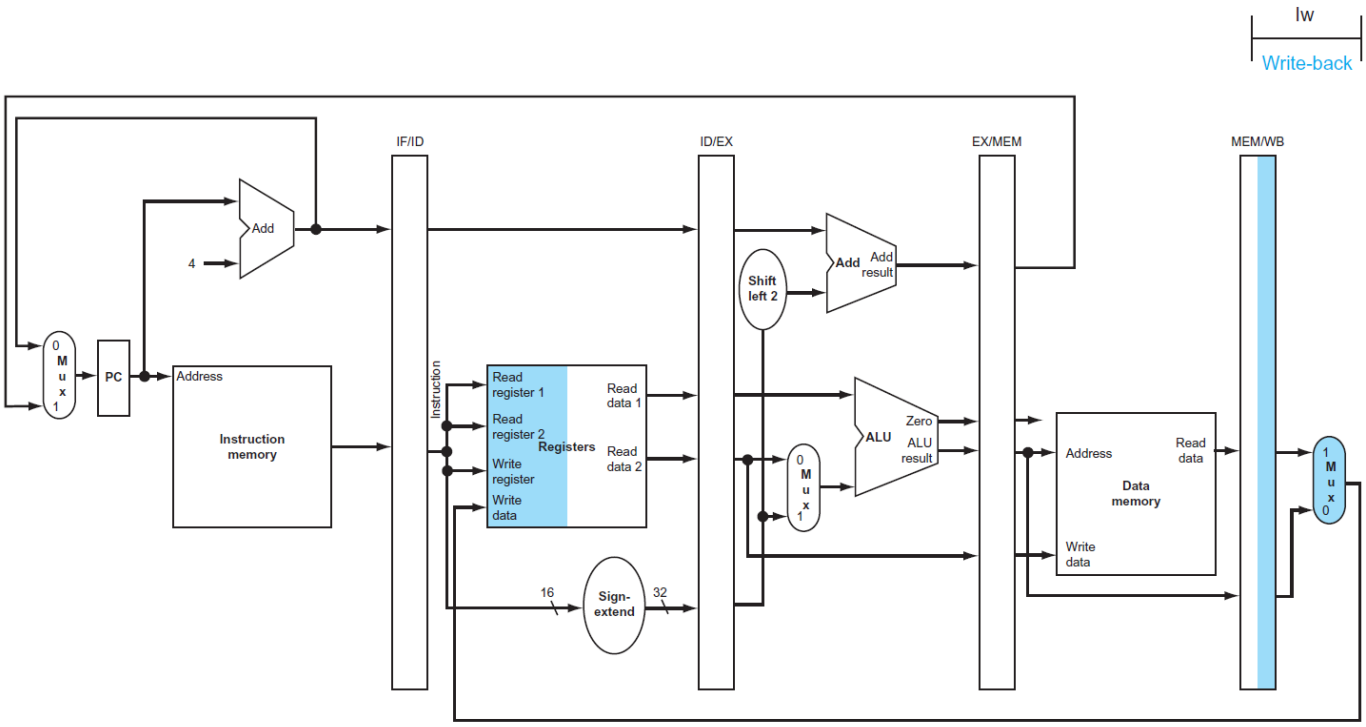
Execution



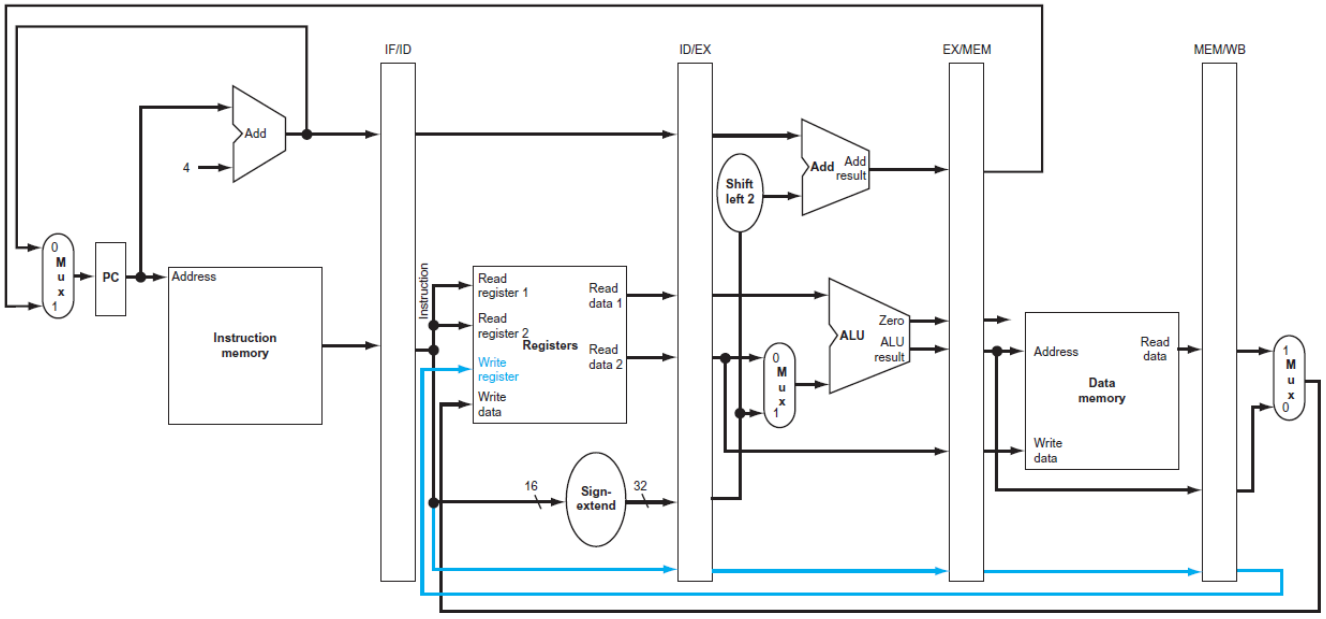
Memory access



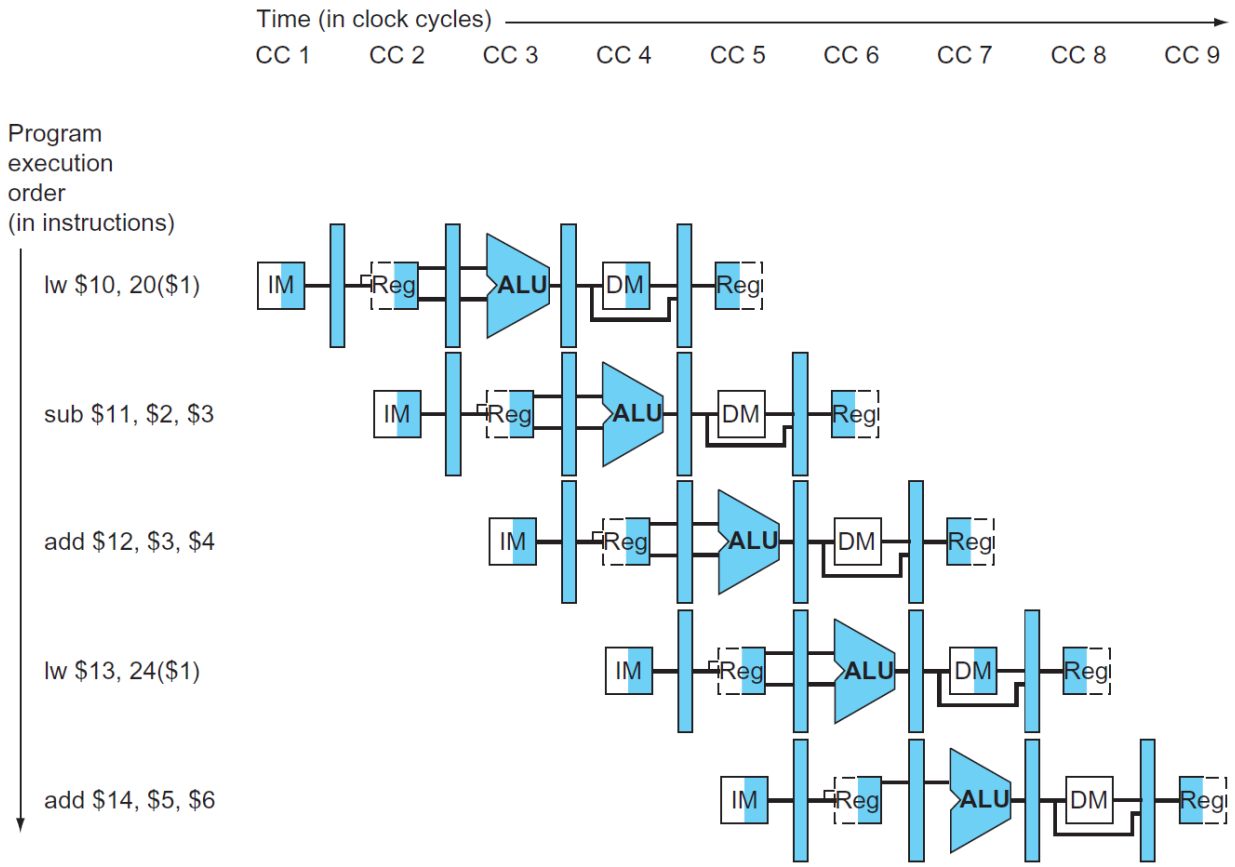
Write-back



Correction to support lw instruction

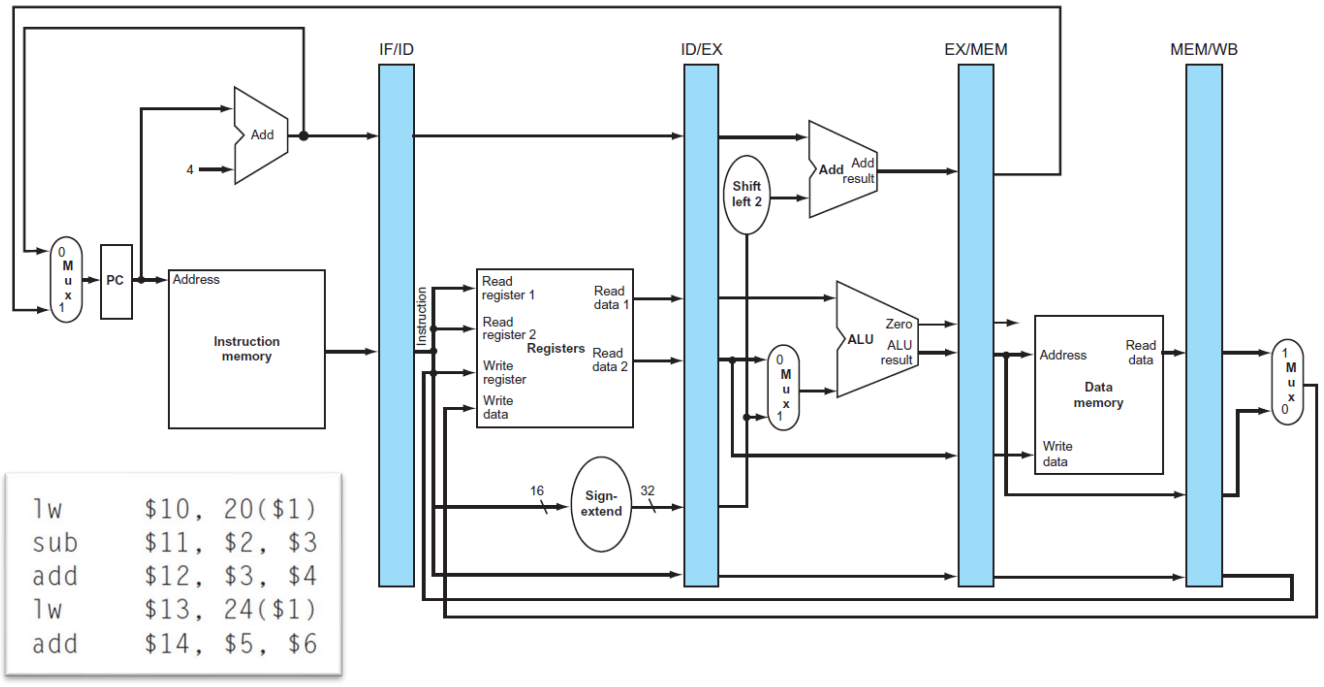


Pipeline diagram



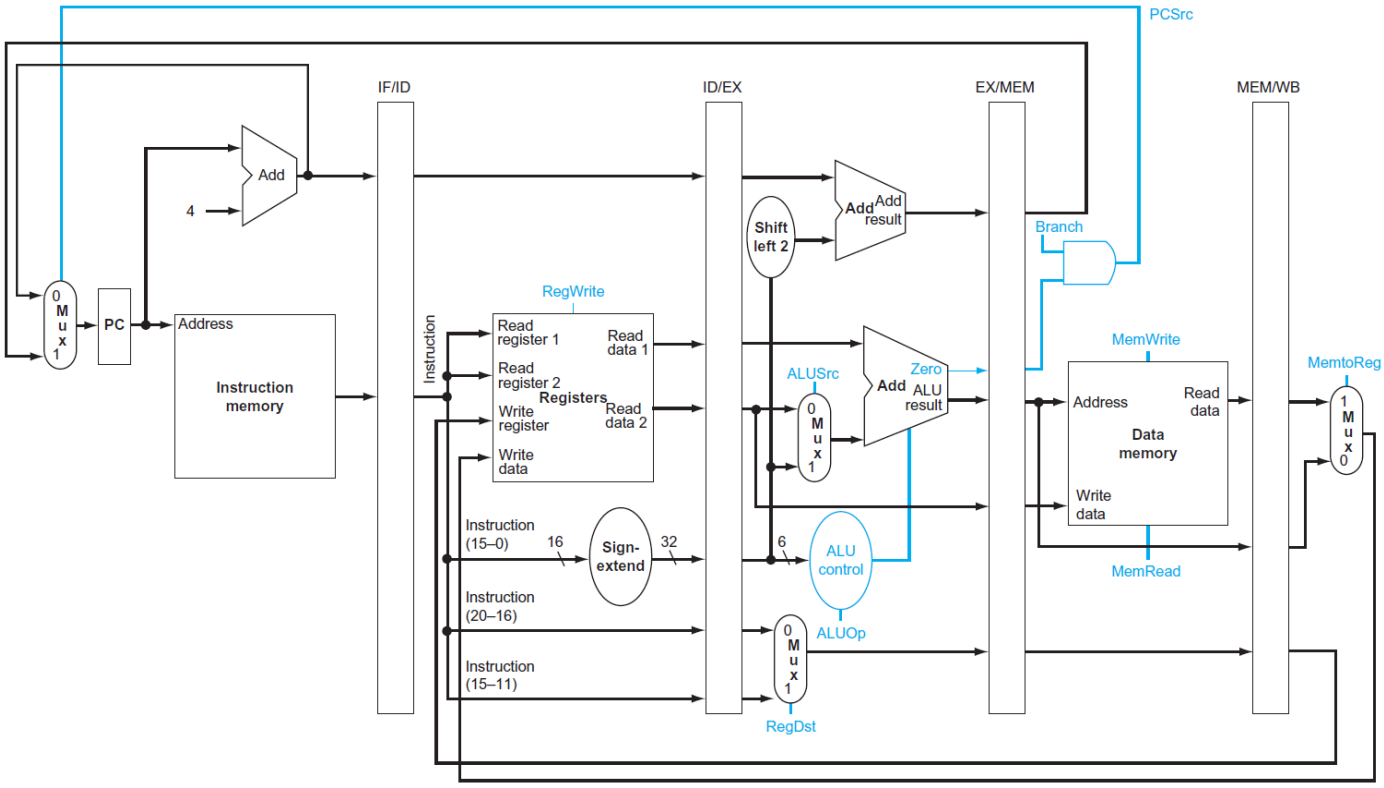
Pipeline diagram

add $14, $5, $6	lw $13, 24 ($1)	add $12, $3, $4	sub $11, $2, $3	lw $10, 20($1)
Instruction fetch	Instruction decode	Execution	Memory	Write-back

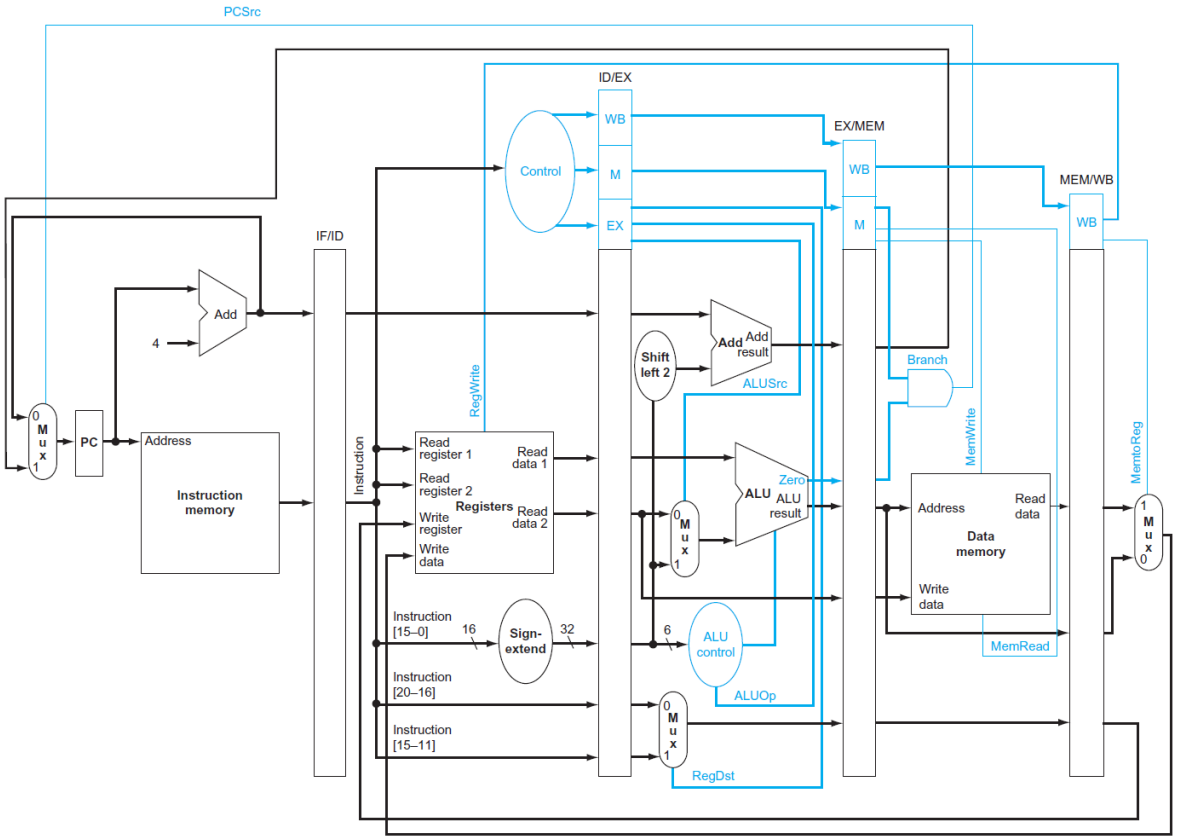



```
lw      $10, 20($1)
sub     $11, $2, $3
add     $12, $3, $4
lw      $13, 24($1)
add     $14, $5, $6
```

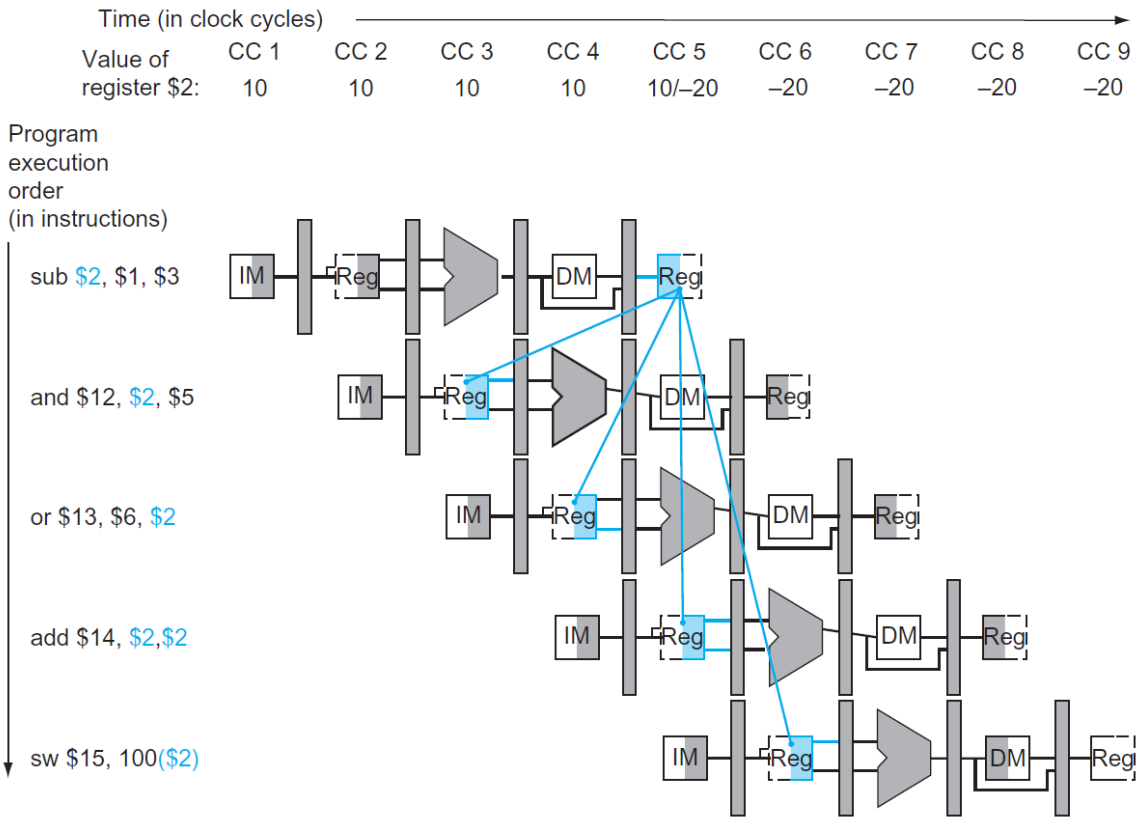

Control signals in pipeline



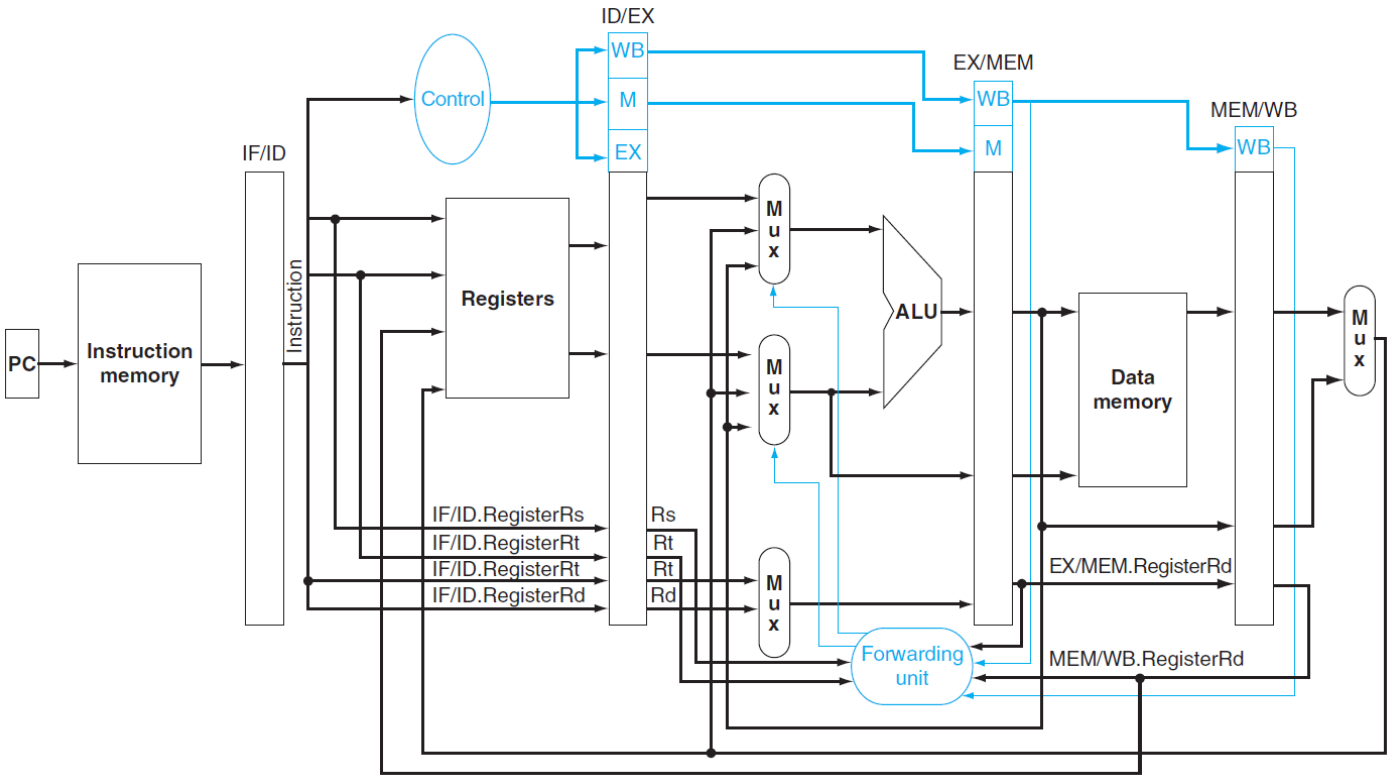
Pipelined control



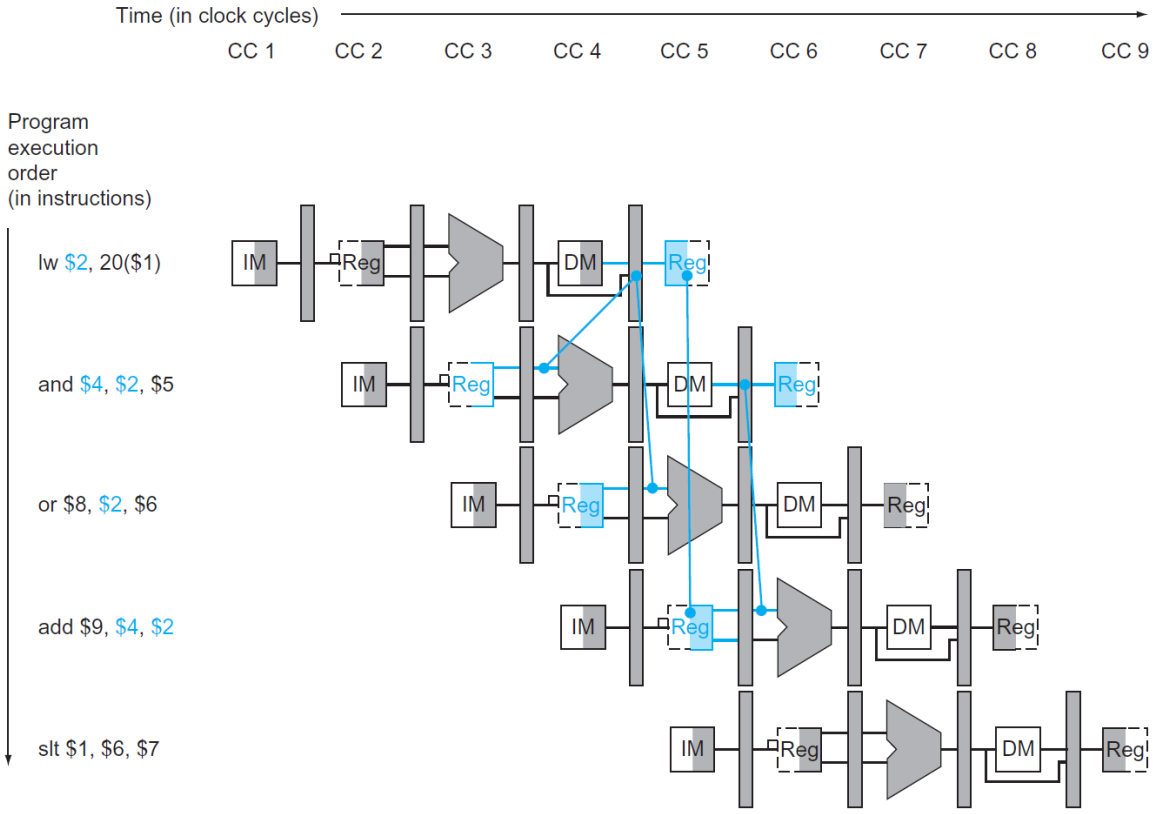
Solving read-after-write hazard



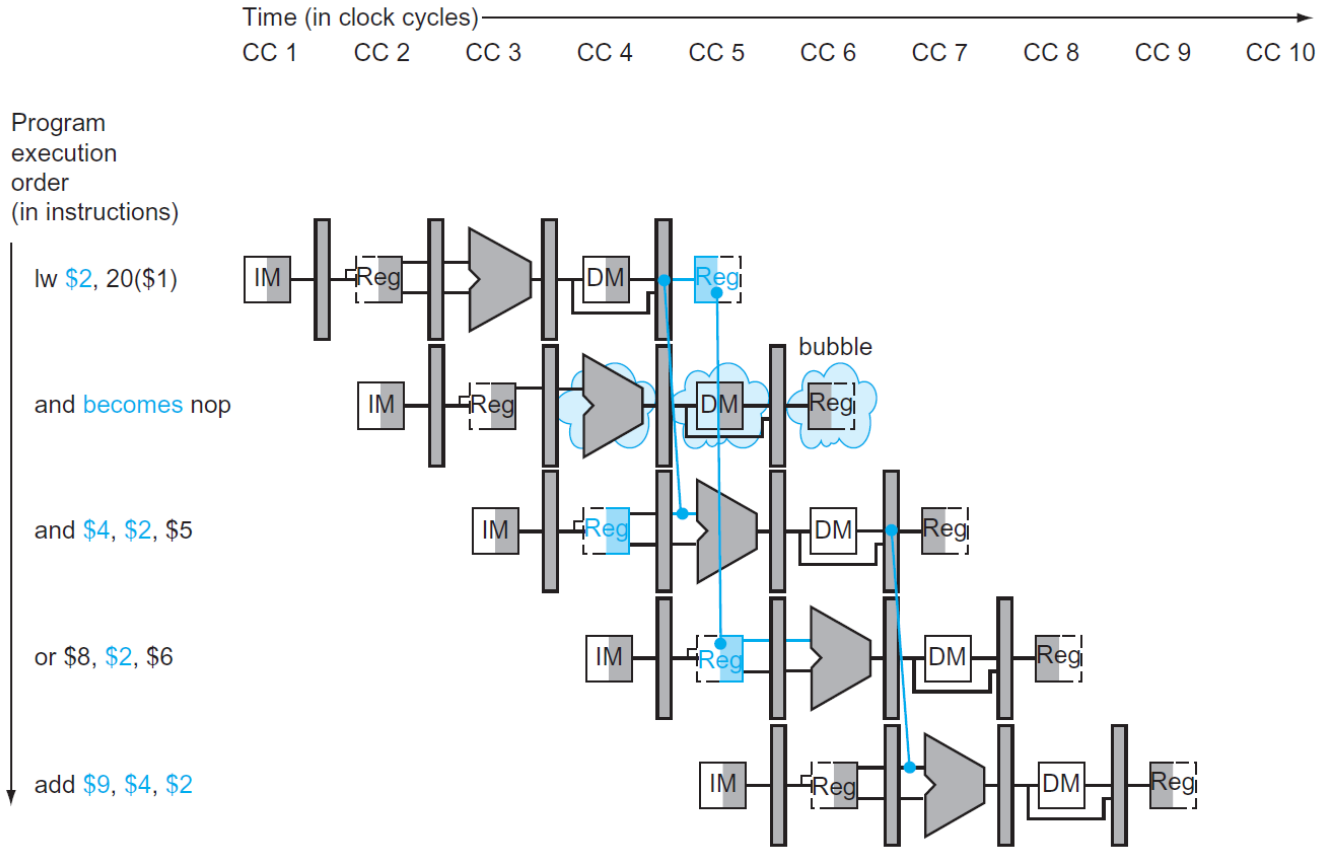
Solving read-after-write hazard



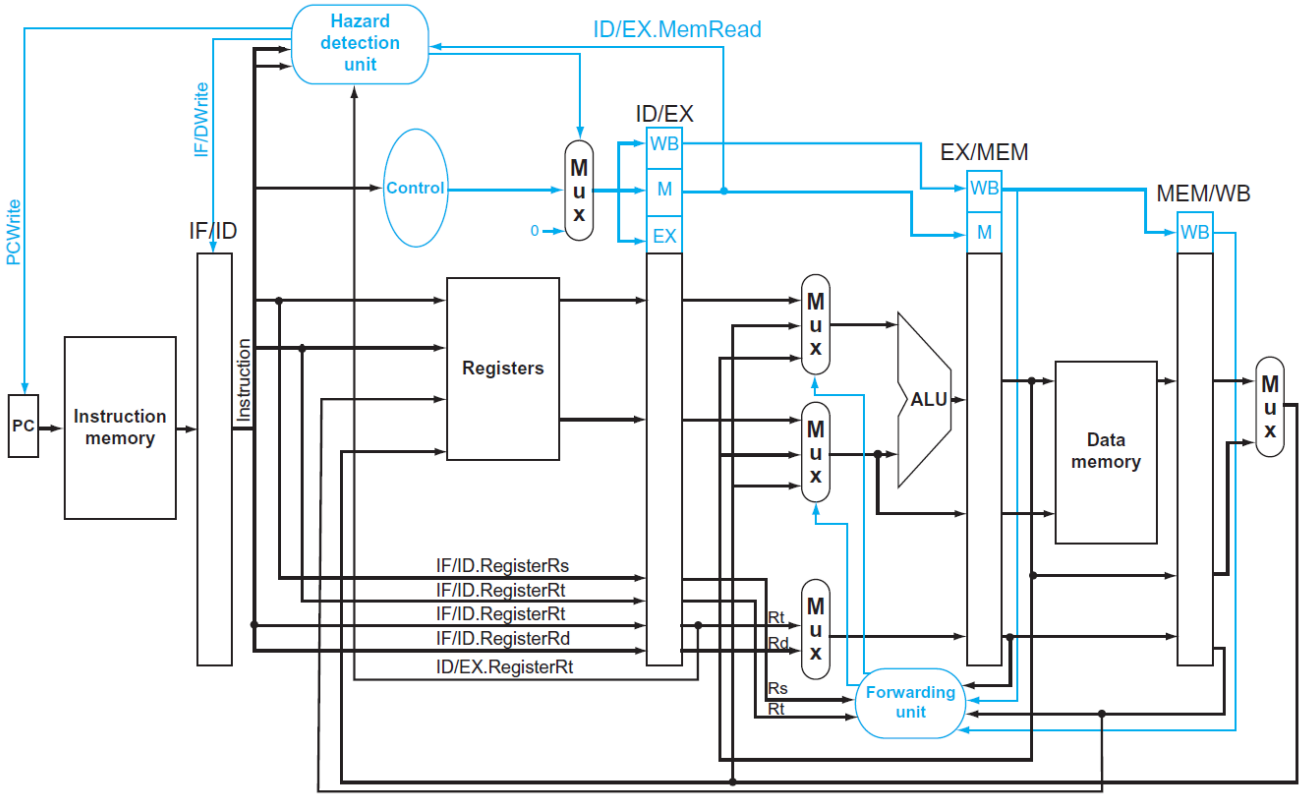
Solving load-use hazard



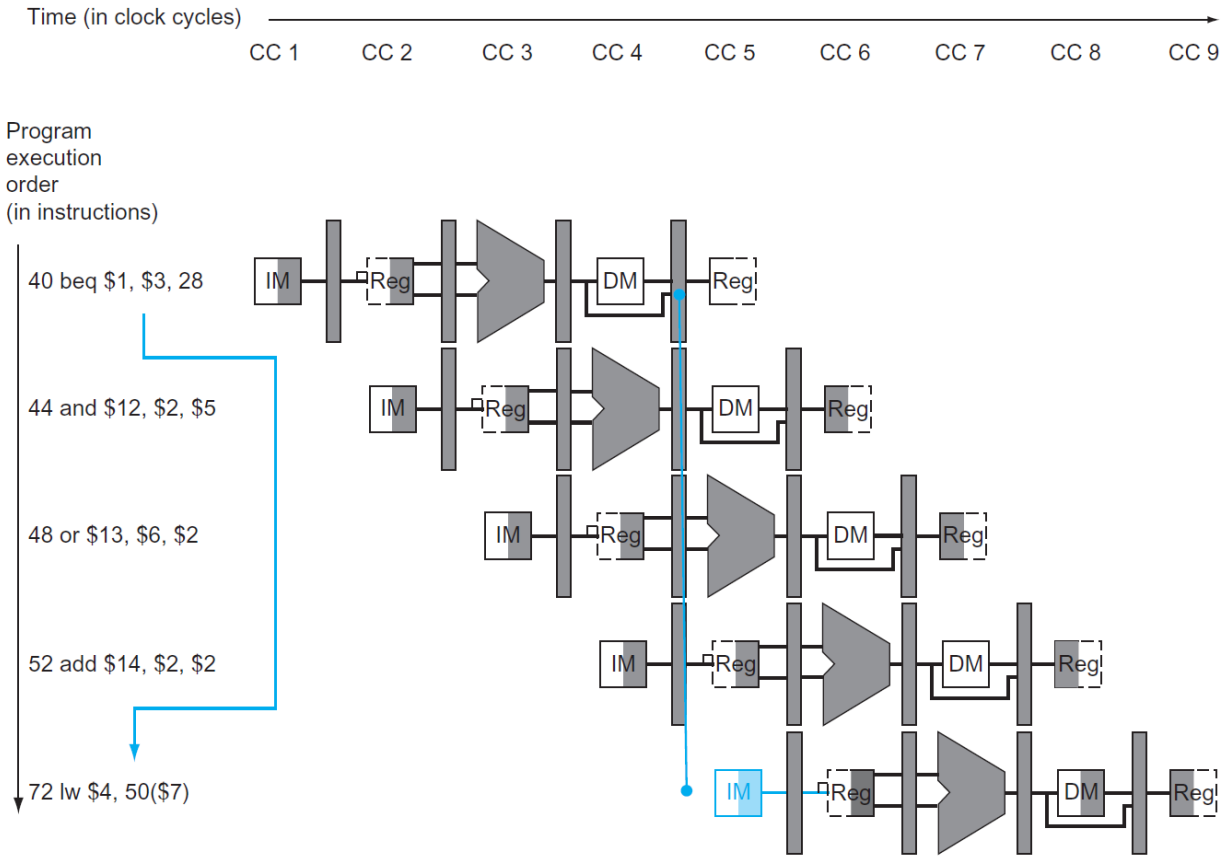
Solving load-use hazard



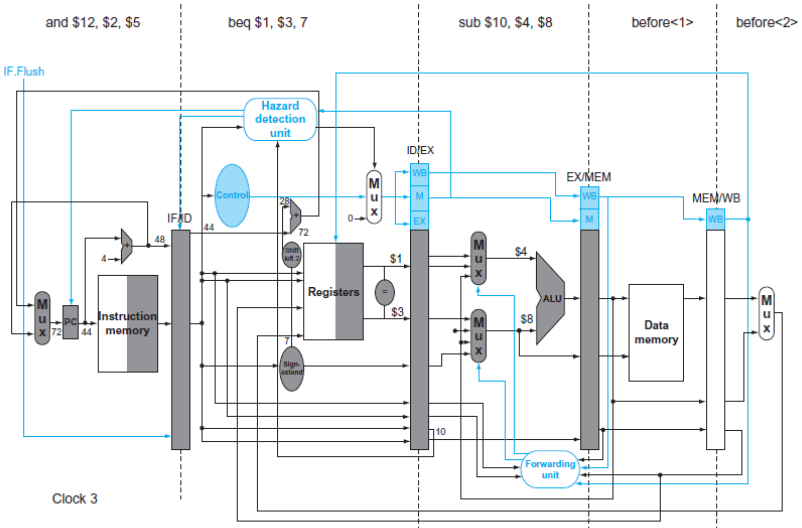
Solving load-use hazard

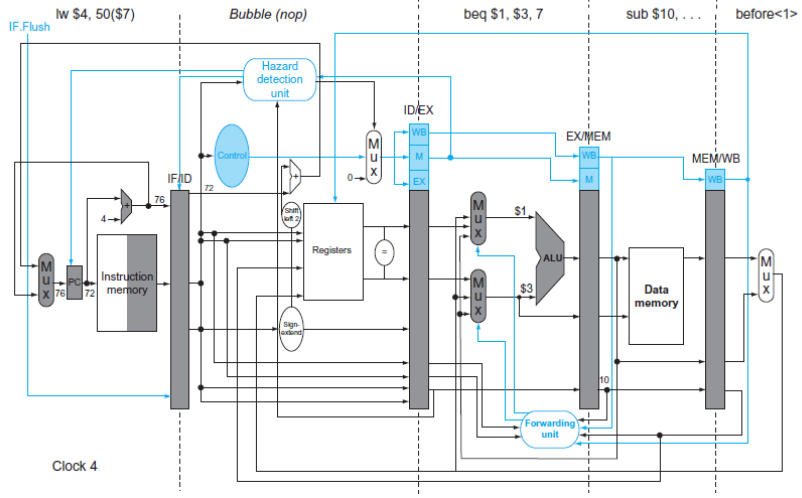


Solving control hazard



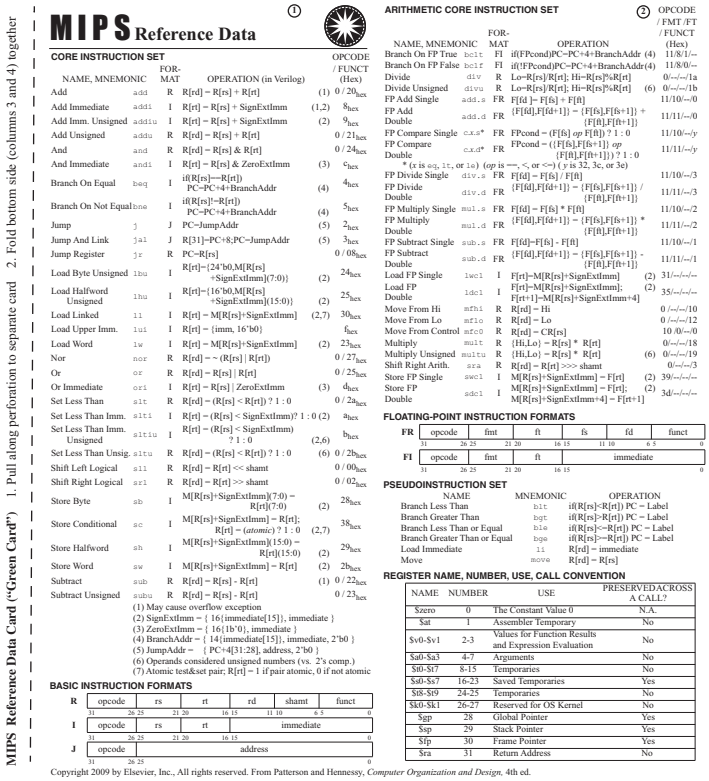
Pipeline when Branch taken





Summary

- All modern-day processors use pipelining
- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Potential speedup: a CPI of 1 and a fast CC
- Must detect and resolve hazards
  - Stalling negatively affects CPI (makes CPI less than the ideal of 1)



MIPS Reference Data Card ("Green Card") 1. Pull along perforation to separate card 2. Fold bottom side (columns 3 and 4) together

MIPS Reference Data ①

CORE INSTRUCTION SET

NAME, MNEMONIC		FOR-MAT	OPERATION (in Verilog)		OPCODE / FUNCT (Hex)
Add	add	R	R[rd] = R[rs] + R[rt]	(1)	0 / 20_{hex}
Add Immediate	addi	I	R[rt] = R[rs] + SignExtImm	(1,2)	8_{hex}
Add Imm. Unsigned	addiu	I	R[rt] = R[rs] + SignExtImm	(2)	9_{hex}
Add Unsigned	addu	R	R[rd] = R[rs] + R[rt]		0 / 21_{hex}
And	and	R	R[rd] = R[rs] & R[rt]		0 / 24_{hex}
And Immediate	andi	I	R[rt] = R[rs] & ZeroExtImm	(3)	c_{hex}
Branch On Equal	beq	I	if(R[rs]==R[rt]) PC=PC+4+BranchAddr	(4)	4_{hex}
Branch On Not Equal	bne	I	if(R[rs]!=R[rt]) PC=PC+4+BranchAddr	(4)	5_{hex}
Jump	j	J	PC=JumpAddr	(5)	2_{hex}
Jump And Link	jal	J	R[31]=PC+8;PC=JumpAddr	(5)	3_{hex}
Jump Register	jr	R	PC=R[rs]		0 / 08_{hex}
Load Byte Unsigned	lbu	I	R[rt]={24'b0,M[R[rs]+SignExtImm](7:0)}	(2)	24_{hex}
Load Halfword Unsigned	lhu	I	R[rt]={16'b0,M[R[rs]+SignExtImm](15:0)}	(2)	25_{hex}
Load Linked	ll	I	R[rt] = M[R[rs]+SignExtImm]	(2,7)	30_{hex}
Load Upper Imm.	lui	I	R[rt] = {imm, 16'b0}		f_{hex}
Load Word	lw	I	R[rt] = M[R[rs]+SignExtImm]	(2)	23_{hex}
Nor	nor	R	R[rd] = ~ (R[rs] | R[rt])		0 / 27_{hex}
Or	or	R	R[rd] = R[rs] | R[rt]		0 / 25_{hex}
Or Immediate	ori	I	R[rt] = R[rs] | ZeroExtImm	(3)	d_{hex}
Set Less Than	slt	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0		0 / $2a_{hex}$
Set Less Than Imm.	slti	I	R[rt] = (R[rs] < SignExtImm)? 1 : 0	(2)	a_{hex}
Set Less Than Imm. Unsigned	sltiu	I	R[rt] = (R[rs] < SignExtImm) ? 1 : 0	(2,6)	b_{hex}
Set Less Than Unsig.	sltu	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0	(6)	0 / $2b_{hex}$
Shift Left Logical	sll	R	R[rd] = R[rt] << shamt		0 / 00_{hex}
Shift Right Logical	srl	R	R[rd] = R[rt] >> shamt		0 / 02_{hex}
Store Byte	sb	I	M[R[rs]+SignExtImm](7:0) = R[rt](7:0)	(2)	28_{hex}
Store Conditional	sc	I	M[R[rs]+SignExtImm] = R[rt]; R[rt] = (*atomic*) ? 1 : 0	(2,7)	38_{hex}
Store Halfword	sh	I	M[R[rs]+SignExtImm](15:0) = R[rt](15:0)	(2)	29_{hex}
Store Word	sw	I	M[R[rs]+SignExtImm] = R[rt]	(2)	$2b_{hex}$
Subtract	sub	R	R[rd] = R[rs] - R[rt]	(1)	0 / 22_{hex}
Subtract Unsigned	subu	R	R[rd] = R[rs] - R[rt]		0 / 23_{hex}

(1) May cause overflow exception
(2) SignExtImm = { 16{immediate[15]}, immediate }
(3) ZeroExtImm = { 16{1'b0}, immediate }
(4) BranchAddr = { 14{immediate[15]}, immediate, 2'b0 }
(5) JumpAddr = { PC+4[31:28], address, 2'b0 }
(6) Operands considered unsigned numbers (vs. 2's comp.)
(7) Atomic test&set pair; R[rt] = 1 if pair atomic, 0 if not atomic

BASIC INSTRUCTION FORMATS

R	opcode	rs	rt	rd	shamt	funct
	31 26	25 21	20 16	15 11	10 6	5 0
I	opcode	rs	rt	immediate		
	31 26	25 21	20 16	15 0		
J	opcode	address				
	31 26	25 0				

ARITHMETIC CORE INSTRUCTION SET ②

NAME, MNEMONIC		FOR-MAT	OPERATION		OPCODE / FMT /FT / FUNCT (Hex)
Branch On FP True	bc1t	FI	if(FPcond)PC=PC+4+BranchAddr	(4)	11/8/1/--
Branch On FP False	bc1f	FI	if(!FPcond)PC=PC+4+BranchAddr	(4)	11/8/0/--
Divide	div	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]		0/--/--/1a
Divide Unsigned	divu	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]	(6)	0/--/--/1b
FP Add Single	add.s	FR	F[fd] = F[fs] + F[ft]		11/10/--/0
FP Add Double	add.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} + {F[ft],F[ft+1]}		11/11/--/0
FP Compare Single	c.x.s*	FR	FPcond = (F[fs] *op* F[ft]) ? 1 : 0		11/10/--/*y*
FP Compare Double	c.x.d*	FR	FPcond = ({F[fs],F[fs+1]} *op* {F[ft],F[ft+1]}) ? 1 : 0		11/11/--/*y*

* (x is `eq`, `lt`, or `le`) (*op* is ==, <, or <=) (*y* is 32, 3c, or 3e)

FP Divide Single	div.s	FR	F[fd] = F[fs] / F[ft]		11/10/--/3
FP Divide Double	div.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} / {F[ft],F[ft+1]}		11/11/--/3
FP Multiply Single	mul.s	FR	F[fd] = F[fs] * F[ft]		11/10/--/2
FP Multiply Double	mul.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} * {F[ft],F[ft+1]}		11/11/--/2
FP Subtract Single	sub.s	FR	F[fd]=F[fs] - F[ft]		11/10/--/1
FP Subtract Double	sub.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} - {F[ft],F[ft+1]}		11/11/--/1
Load FP Single	lwc1	I	F[rt]=M[R[rs]+SignExtImm]	(2)	31/--/--/--
Load FP Double	ldc1	I	F[rt]=M[R[rs]+SignExtImm]; F[rt+1]=M[R[rs]+SignExtImm+4]	(2)	35/--/--/--
Move From Hi	mfhi	R	R[rd] = Hi		0 /--/--/10
Move From Lo	mflo	R	R[rd] = Lo		0 /--/--/12
Move From Control	mfc0	R	R[rd] = CR[rs]		10 /0/--/0
Multiply	mult	R	{Hi,Lo} = R[rs] * R[rt]		0/--/--/18
Multiply Unsigned	multu	R	{Hi,Lo} = R[rs] * R[rt]	(6)	0/--/--/19
Shift Right Arith.	sra	R	R[rd] = R[rt] >>> shamt		0/--/--/3
Store FP Single	swc1	I	M[R[rs]+SignExtImm] = F[rt]	(2)	39/--/--/--
Store FP Double	sdc1	I	M[R[rs]+SignExtImm] = F[rt]; M[R[rs]+SignExtImm+4] = F[rt+1]	(2)	3d/--/--/--

FLOATING-POINT INSTRUCTION FORMATS

FR	opcode	fmt	ft	fs	fd	funct
	31 26	25 21	20 16	15 11	10 6	5 0
FI	opcode	fmt	ft	immediate		
	31 26	25 21	20 16	15 0		

PSEUDOINSTRUCTION SET

NAME	MNEMONIC	OPERATION
Branch Less Than	`blt`	if(R[rs]<R[rt]) PC = Label
Branch Greater Than	`bgt`	if(R[rs]>R[rt]) PC = Label
Branch Less Than or Equal	`ble`	if(R[rs]<=R[rt]) PC = Label
Branch Greater Than or Equal	`bge`	if(R[rs]>=R[rt]) PC = Label
Load Immediate	`li`	R[rd] = immediate
Move	`move`	R[rd] = R[rs]

REGISTER NAME, NUMBER, USE, CALL CONVENTION

NAME	NUMBER	USE	PRESERVED ACROSS A CALL?
$zero	0	The Constant Value 0	N.A.
$at	1	Assembler Temporary	No
$v0-$v1	2-3	Values for Function Results and Expression Evaluation	No
$a0-$a3	4-7	Arguments	No
$t0-$t7	8-15	Temporaries	No
$s0-$s7	16-23	Saved Temporaries	Yes
$t8-$t9	24-25	Temporaries	No
$k0-$k1	26-27	Reserved for OS Kernel	No
$gp	28	Global Pointer	Yes
$sp	29	Stack Pointer	Yes
$fp	30	Frame Pointer	Yes
$ra	31	Return Address	No

OPCODES, BASE CONVERSION, ASCII SYMBOLS

③

MIPS opcode (31:26)	(1) MIPS funct (5:0)	(2) MIPS funct (5:0)	Binary	Decimal	Hexadecimal	ASCII Character	Decimal	Hexadecimal	ASCII Character
(1)	`sll`	`add.f`	00 0000	0	0	NUL	64	40	@
		`sub.f`	00 0001	1	1	SOH	65	41	A
`j`	`srl`	`mul.f`	00 0010	2	2	STX	66	42	B
`jal`	`sra`	`div.f`	00 0011	3	3	ETX	67	43	C
`beq`	`sllv`	`sqrt.f`	00 0100	4	4	EOT	68	44	D
`bne`		`abs.f`	00 0101	5	5	ENQ	69	45	E
`blez`	`srlv`	`mov.f`	00 0110	6	6	ACK	70	46	F
`bgtz`	`srav`	`neg.f`	00 0111	7	7	BEL	71	47	G
`addi`	`jr`		00 1000	8	8	BS	72	48	H
`addiu`	`jalr`		00 1001	9	9	HT	73	49	I
`slti`	`movz`		00 1010	10	a	LF	74	4a	J
`sltiu`	`movn`		00 1011	11	b	VT	75	4b	K
`andi`	`syscall`	`round.w.f`	00 1100	12	c	FF	76	4c	L
`ori`	`break`	`trunc.w.f`	00 1101	13	d	CR	77	4d	M
`xori`		`ceil.w.f`	00 1110	14	e	SO	78	4e	N
`lui`	`sync`	`floor.w.f`	00 1111	15	f	SI	79	4f	O
	`mfhi`		01 0000	16	10	DLE	80	50	P
(2)	`mthi`		01 0001	17	11	DC1	81	51	Q
	`mflo`	`movs.f`	01 0010	18	12	DC2	82	52	R
	`mtlo`	`movn.f`	01 0011	19	13	DC3	83	53	S
			01 0100	20	14	DC4	84	54	T
			01 0101	21	15	NAK	85	55	U
			01 0110	22	16	SYN	86	56	V
			01 0111	23	17	ETB	87	57	W
	`mult`		01 1000	24	18	CAN	88	58	X
	`multu`		01 1001	25	19	EM	89	59	Y
	`div`		01 1010	26	1a	SUB	90	5a	Z
	`divu`		01 1011	27	1b	ESC	91	5b	[
			01 1100	28	1c	FS	92	5c	\
			01 1101	29	1d	GS	93	5d	]
			01 1110	30	1e	RS	94	5e	^
			01 1111	31	1f	US	95	5f	_
`lb`	`add`	`cvt.s.f`	10 0000	32	20	Space	96	60	`
`lh`	`addu`	`cvt.d.f`	10 0001	33	21	!	97	61	a
`lwl`	`sub`		10 0010	34	22	"	98	62	b
`lw`	`subu`		10 0011	35	23	#	99	63	c
`lbu`	`and`	`cvt.w.f`	10 0100	36	24	$	100	64	d
`lhu`	`or`		10 0101	37	25	%	101	65	e
`lwr`	`xor`		10 0110	38	26	&	102	66	f
	`nor`		10 0111	39	27	'	103	67	g
`sb`			10 1000	40	28	(	104	68	h
`sh`			10 1001	41	29	)	105	69	i
`swl`	`slt`		10 1010	42	2a	*	106	6a	j
`sw`	`sltu`		10 1011	43	2b	+	107	6b	k
			10 1100	44	2c	,	108	6c	l
			10 1101	45	2d	-	109	6d	m
`swr`			10 1110	46	2e	.	110	6e	n
`cache`			10 1111	47	2f	/	111	6f	o
`ll`	`tge`	`c.f.f`	11 0000	48	30	0	112	70	p
`lwc1`	`tgeu`	`c.un.f`	11 0001	49	31	1	113	71	q
`lwc2`	`tlt`	`c.eq.f`	11 0010	50	32	2	114	72	r
`pref`	`tltu`	`c.ueq.f`	11 0011	51	33	3	115	73	s
	`teq`	`c.olt.f`	11 0100	52	34	4	116	74	t
`ldc1`		`c.ult.f`	11 0101	53	35	5	117	75	u
`ldc2`	`tne`	`c.ole.f`	11 0110	54	36	6	118	76	v
		`c.ule.f`	11 0111	55	37	7	119	77	w
`sc`		`c.sf.f`	11 1000	56	38	8	120	78	x
`swc1`		`c.ngle.f`	11 1001	57	39	9	121	79	y
`swc2`		`c.seq.f`	11 1010	58	3a	:	122	7a	z
		`c.ngl.f`	11 1011	59	3b	;	123	7b	{
		`c.lt.f`	11 1100	60	3c	<	124	7c	|
`sdc1`		`c.nge.f`	11 1101	61	3d	=	125	7d	}
`sdc2`		`c.le.f`	11 1110	62	3e	>	126	7e	~
		`c.ngt.f`	11 1111	63	3f	?	127	7f	DEL

(1) opcode(31:26) = 0
(2) opcode(31:26) = 17_{ten} (11_{hex}); if fmt(25:21)=16_{ten} (10_{hex}) *f* = s (single); if fmt(25:21)=17_{ten} (11_{hex}) *f* = d (double)

IEEE 754 FLOATING-POINT STANDARD

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

where Single Precision Bias = 127, Double Precision Bias = 1023.

④

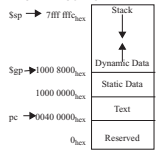
IEEE 754 Symbols

Exponent	Fraction	Object
0	0	± 0
0	≠0	± Denorm
1 to MAX - 1	anything	± Fl. Pt. Num.
MAX	0	±∞
MAX	≠0	NaN

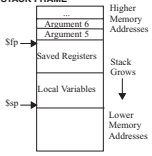
IEEE Single Precision and Double Precision Formats:

S.P. MAX = 255, D.P. MAX = 2047

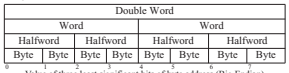
MEMORY ALLOCATION



STACK FRAME

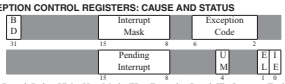


DATA ALIGNMENT



Value of three least significant bits of byte address (Big Endian)

EXCEPTION CONTROL REGISTERS: CAUSE AND STATUS



BD = Branch Delay, UM = User Mode, EL = Exception Level, IE = Interrupt Enable

EXCEPTION CODES

Number	Name	Cause of Exception	Number	Name	Cause of Exception
0	Int	Interrupt (hardware)	9	Bp	Breakpoint Exception
4	AdEL	Address Error Exception (load or instruction fetch)	10	RI	Reserved Instruction Exception
5	AdES	Address Error Exception (store)	11	CpU	Coprocessor Unimplemented
6	IBE	Bus Error on Instruction Fetch	12	Ov	Arithmetic Overflow Exception
7	DBE	Bus Error on Load or Store	13	Tr	Trap
8	Sys	Syscall Exception	15	FPE	Floating Point Exception

SIZE PREFIXES (10^x for Disk, Communication; 2^x for Memory)

SI Size	Prefix	Symbol	IEC Size	Prefix	Symbol
10^3	Kilo-	K	2^{10}	Kibi-	Ki
10^6	Mega-	M	2^{20}	Mebi-	Mi
10^9	Giga-	G	2^{30}	Gibi-	Gi
10^{12}	Tera-	T	2^{40}	Tebi-	Ti
10^{15}	Peta-	P	2^{50}	Pebi-	Pi
10^{18}	Exa-	E	2^{60}	Exbi-	Ei
10^{21}	Zetta-	Z	2^{70}	Zebi-	Zi
10^{24}	Yotta-	Y	2^{80}	Yobi-	Yi

MIPS Reference Data Card ("Green Card") 1. Pull along perforation to separate card 2. Fold bottom side (columns 3 and 4) together