Computer Systems
Week 2 (part 2)
Instructions, Assembly Language, and Machine Code

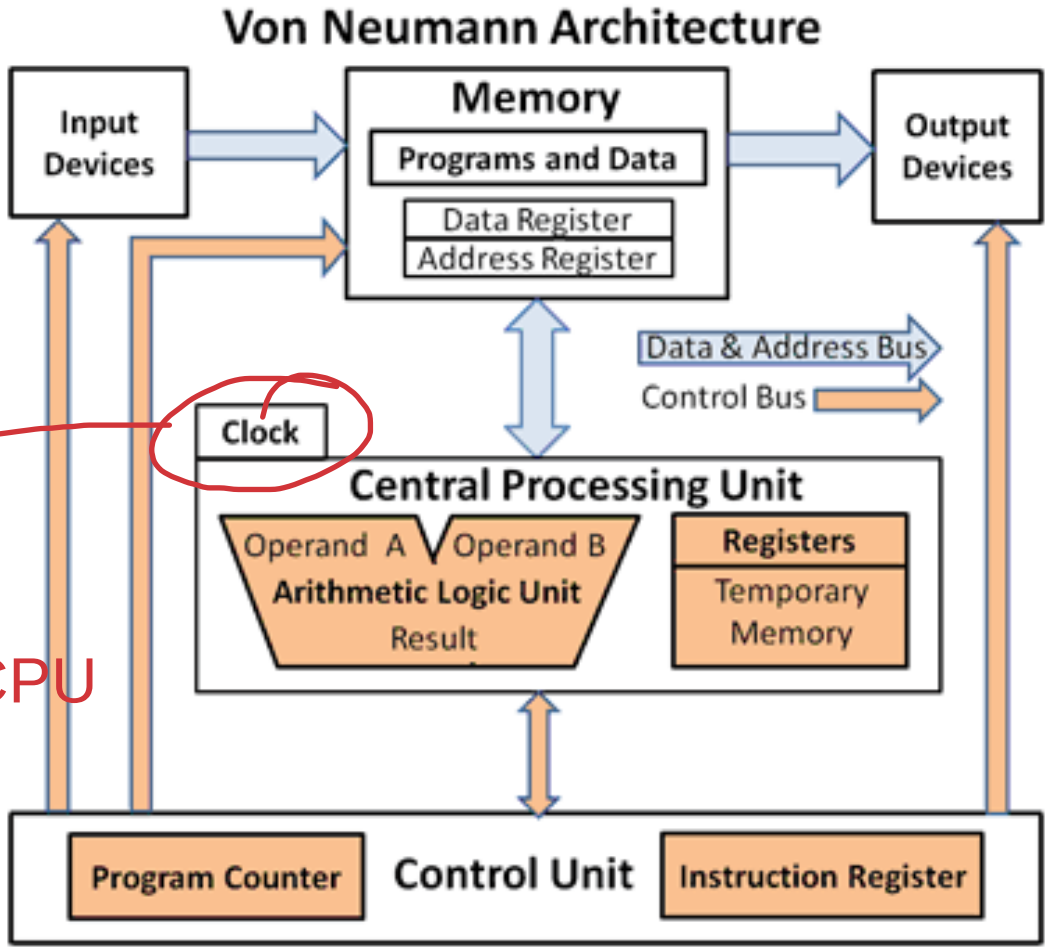
Lecture Objectives

To develop fundmental understading of computer **architecture & organisation**, **instruction sets** and **assembly language** programming.

Lecture Outline

- The von Neumann Architecture
- CPU Cycles and Instruction Pipelining
- The Harvard Architecture
- Case Study – MIPS R4000
  - Structure
  - Key Features
  - Assembly Languages
  - Instruction Sets
  - MIPS Instruction Types

The von Neumann Architecture (1/4)



Drives the CPU

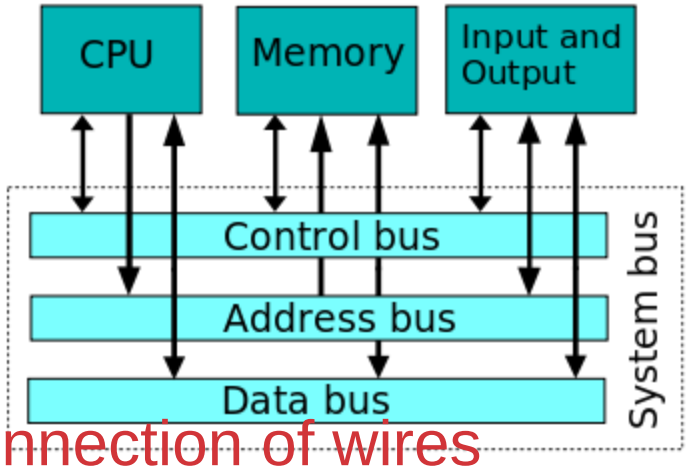
Source: https://www.mpoweruk.com/computer_architecture.htm

The von Neumann Architecture (2/4)

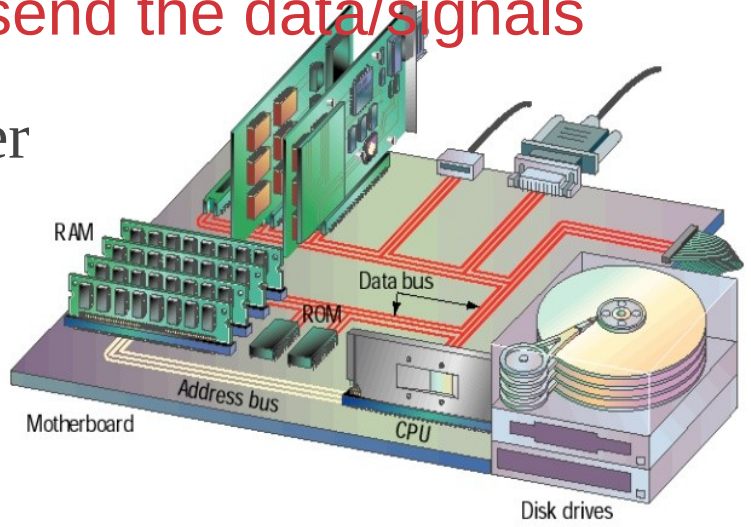
- The Central Processing Unit (CPU), which consists of
  - Control Unit
  - Arithmetic and Logical Unit (ALU)
  - Registers
    - **Program Counter** – Address of the Next Instruction
    - **Instruction Register** – Instruction currently being executed or decoded
    - **Address Register** – Either stores the memory address from which data will be fetched, or the address to which data will be sent and stored
    - **Accumulator (Register)** – short-term, intermediate storage of arithmetic and logic data computations

The von Neumann Architecture (3/4)

- System Bus
  - **Control Bus** – carries commands from the CPU and returns status signals from the devices
  - **Data Bus** – carries the actual data being processed
  - **Address Bus** – carries memory addresses from the processor to other components
- Memory (RAM)
- Input / Output Units

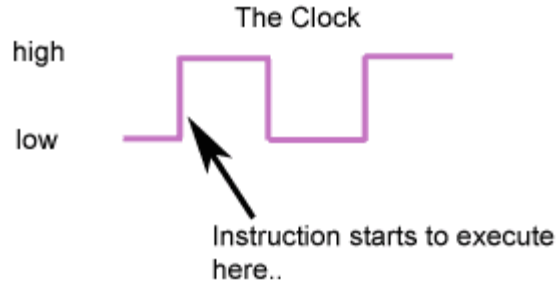


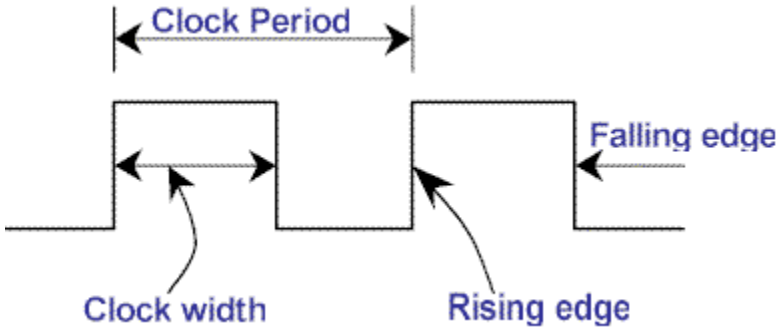
Bus is a connection of wires used to send the data/signals



The von Neumann Architecture (4/4)

- Clock Cycle
  - A signal that oscillates between high and low - used to synchronise (coordinate) actions.
  - The number of cycles that a CPU uses per second is used to determine its speed.
    - This is measured in megahertz (MHz) and gigahertz (GHz)
  - The rate of the Fetch/Execute Cycle is determined by the computer's clock.





One Cycle per Clock Tick

- A computer with a 1 GHz clock has one billionth of a second—one nanosecond—between clock ticks to run the Fetch/Execute Cycle.
  - Modern computers try to start an instruction on each clock tick.
- They pass off completing the instruction to other circuitry. This process is called **pipelining** and frees the fetch unit to start the next instruction before the last one is done.
  - It is not quite true that 1,000 instructions are executed in 1,000 ticks

Schematic Fetch/Execute Cycle (Pipelining)

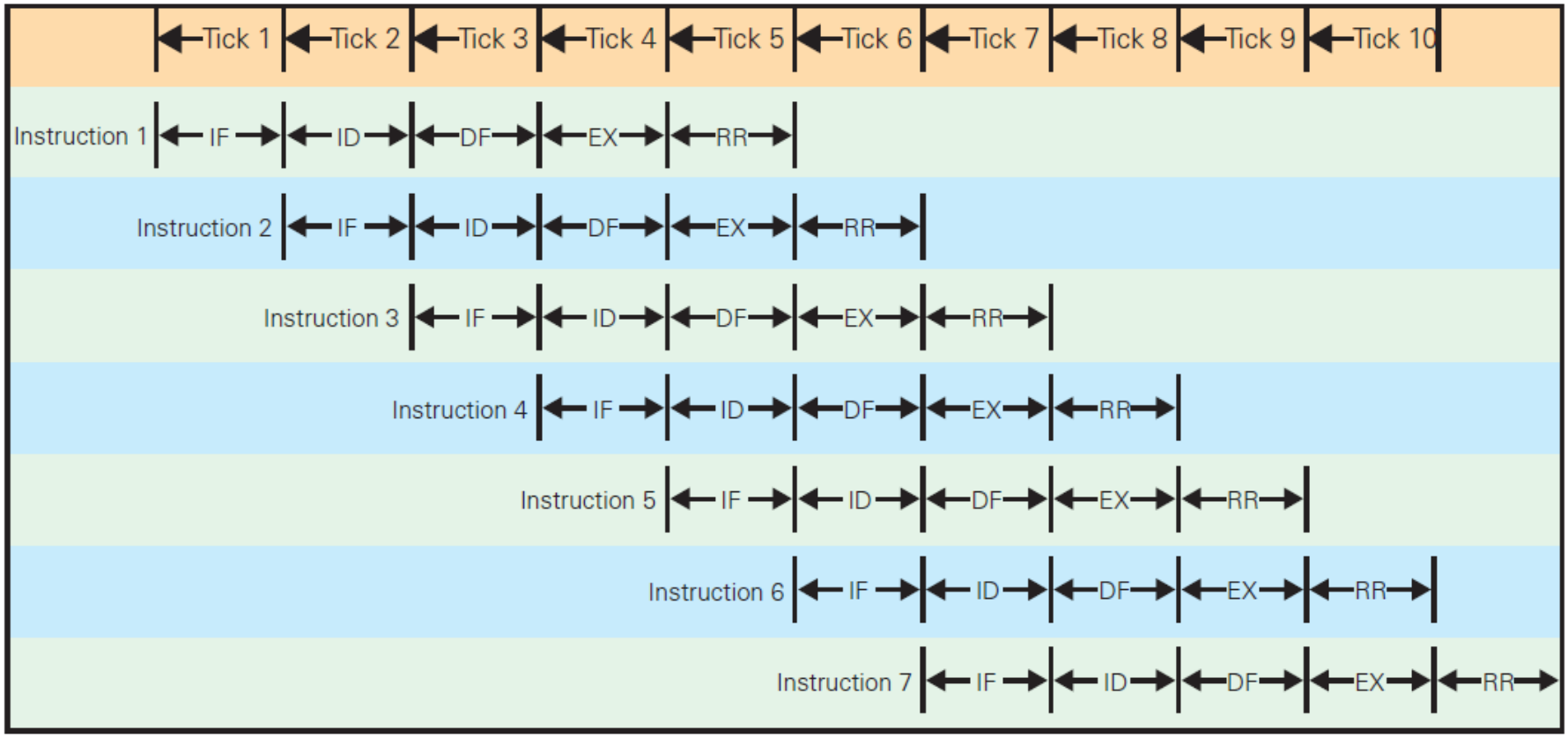
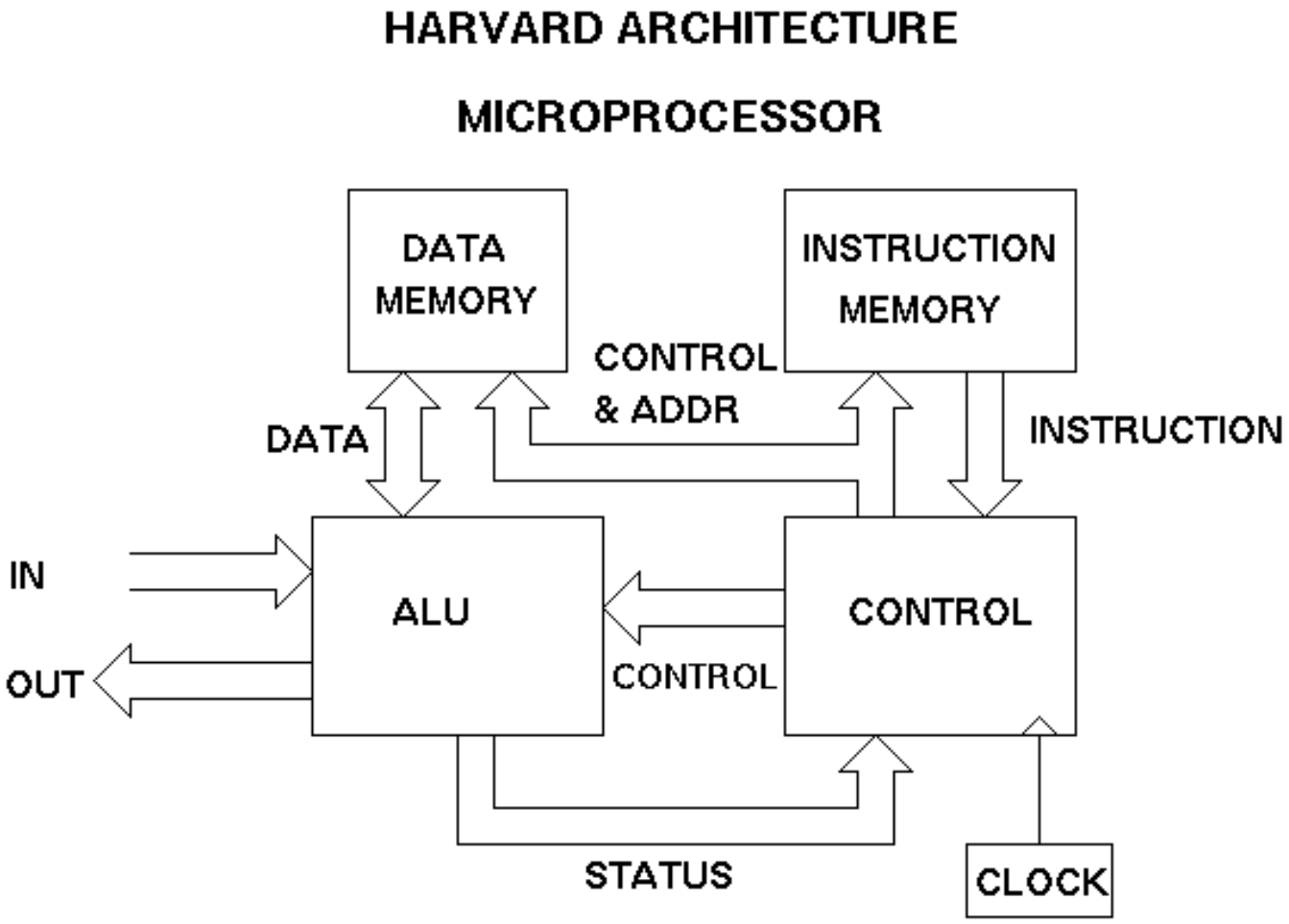


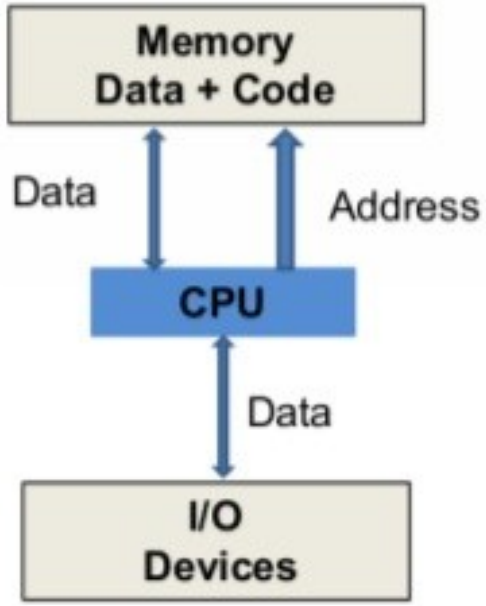
Figure 9.12 Schematic diagram of a pipelined Fetch/Execute Cycle. On each tick, the IF (Instruction Fetch) circuit starts a new instruction, and then passes it along to the ID (Instruction Decode) unit; the ID unit works on the instruction it receives, and when it finishes, it passes it along to the DF (Data Fetch) circuit, and so on. When the pipeline is filled, five instructions are in process at once, and one instruction is finished on each clock tick, making the computer appear to be running at one instruction per tick.

The Harvard Architecture

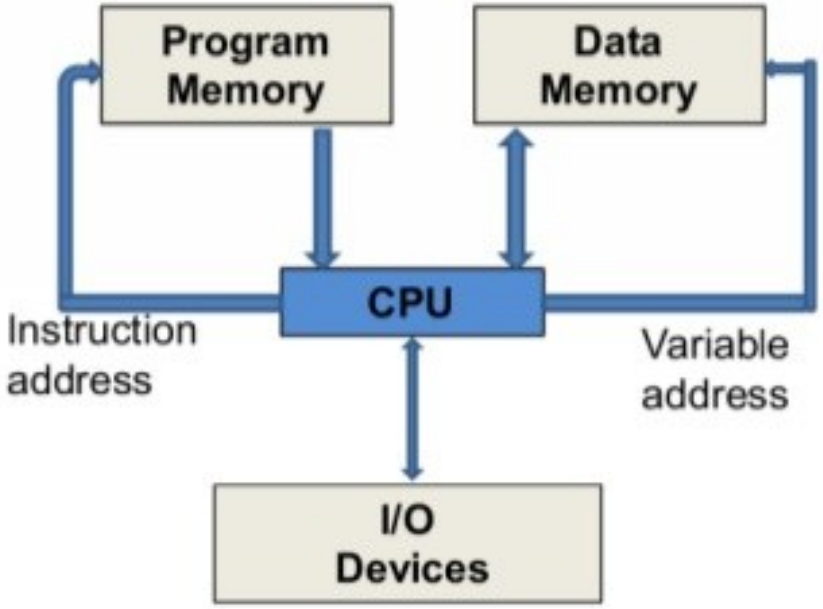


Comparison – von Neumann vs Harvard

Often labeled as
Read only



Von Neumann Machine

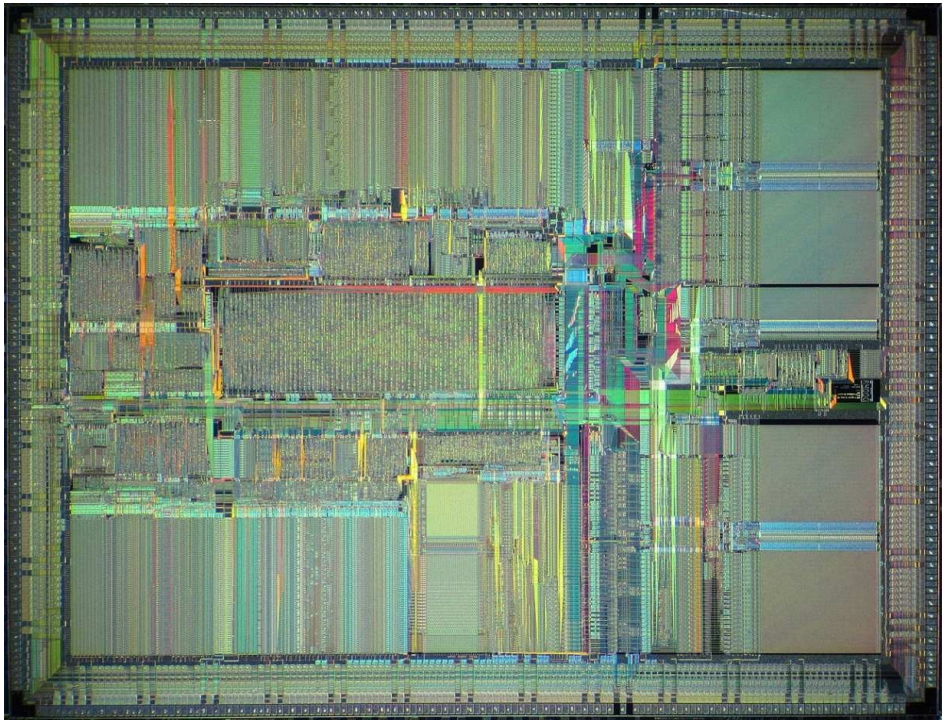


Harvard Machine

Harvard architechture is designed
To differentiate data and instructions in order to prevent

Case Study - MIPS R4000

- Similar to von Neumann but separate interfaces to instructions and data.
- Modified ***Harvard*** architecture



CPU – top view

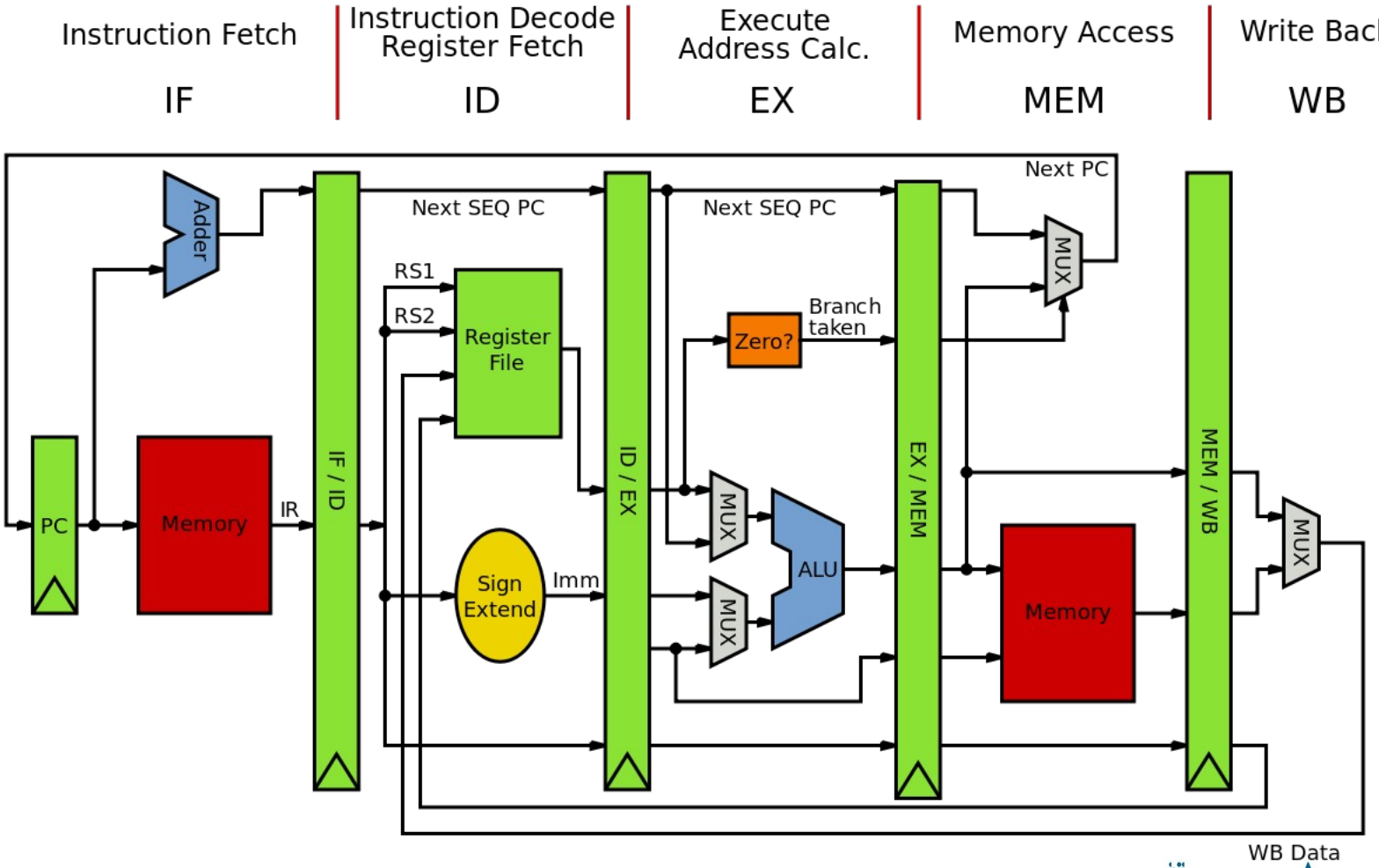
SGI Onyx

Case Study - MIPS R4000

- First 64-bit architecture
- Integrated caches (on-chip, off-chip secondary cache)
- Integrated Floating Point Unit (FPU)
- Implemented in 1992
  - Deep pipeline
  - 1.4M transistors Million instructions per secon
  - Initially 100 MHz (>50 MIPs)
- Simple, well-behaved programming model
- Compact set of instructions with regular format (MIPS III)
- “Easy” to translate from high-level code to machine language

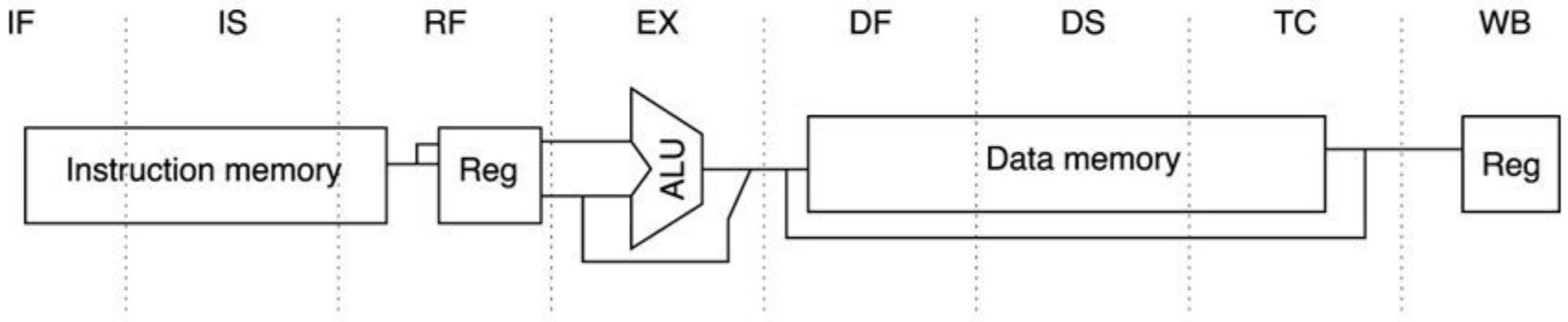
Comparison:
http://www.alexvoica.com/wp-content/uploads/2015/09/great-mips-chips-past_f.png

Typical MIPS Pipeline

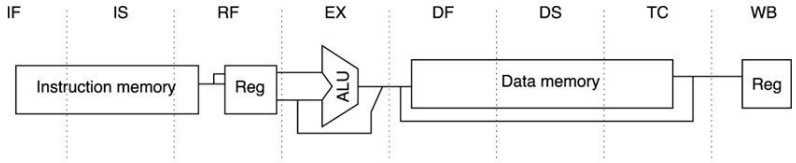


MIPS R4000 Pipeline

- 8 stage pipeline (super-pipelined)
- Extend IF and MEM stages to account for cache overheads
  - Split into “First” and “Second” stages followed by a “Tag Check” for misses
  - The IF tag check is done in the RF stage



MIPS R4000 Pipeline Stages

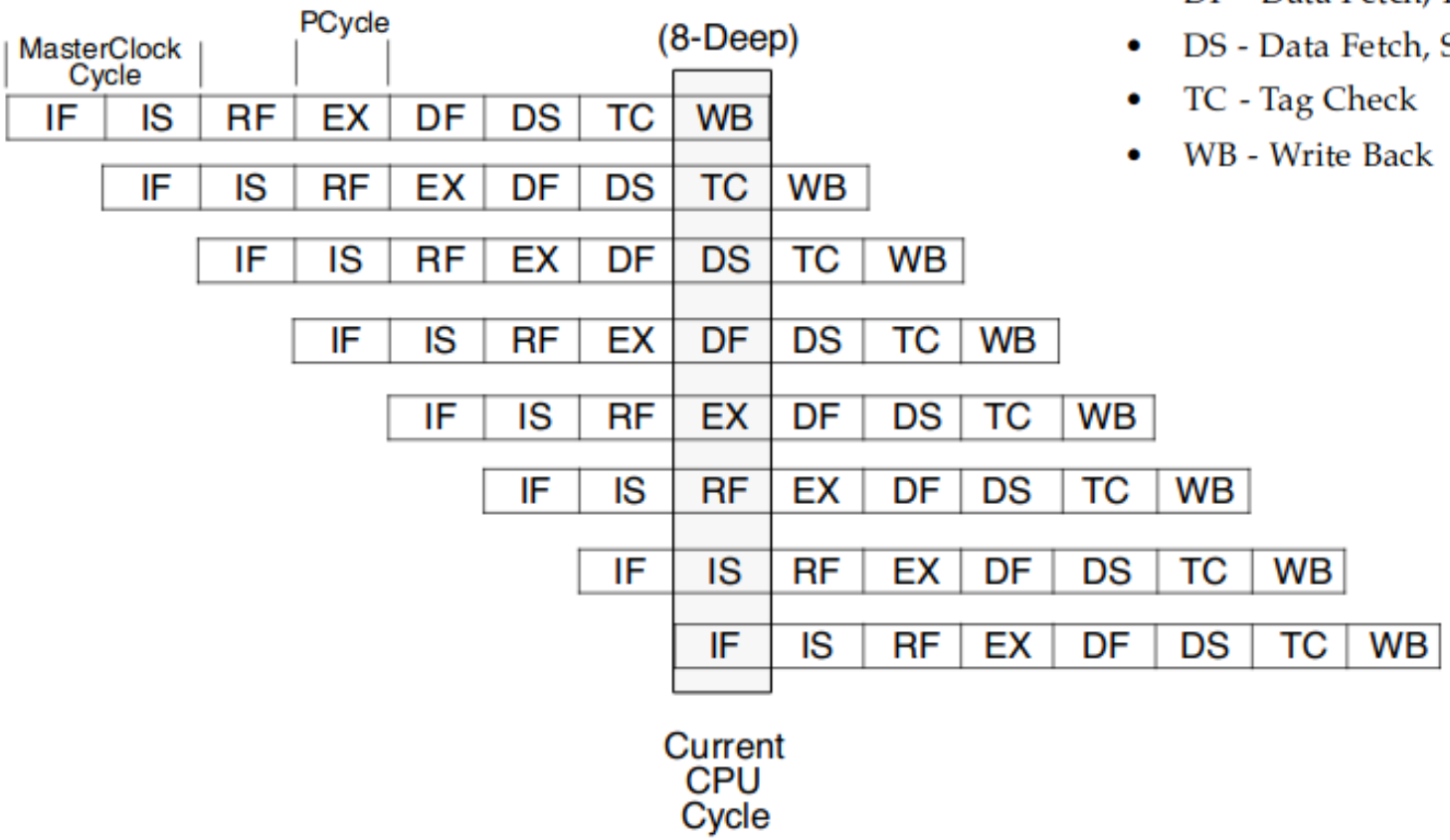


1) **IF** : First half of instruction fetch; PC selection actually happens here, together with initiation of instruction cache access
2) **IS** : Second half of instruction fetch, complete instruction cache access.
3) **RF** : Instruction decode and register fetch, hazard checking, and also instruction cache hit detection
4) **EX** : Execution, which includes effective address calculation, ALU operation, and branch target completion of data cache access
5) **DF** : Data fetch, first half of data cache access
6) **DS** : Second half of data fetch, completion of data cache access
7) **TC** : Tag check, determine whether the data cache access hit
8) **WB** : Write back for loads and register-register operations

Source: https://www.cs.umd.edu/~meesh/cmsc411/website/proj01/MIPS4000/

MIPS R4000 Pipeline Stages

- IF - Instruction Fetch, First Half
- IS - Instruction Fetch, Second Half
- RF - Register Fetch
- EX - Execution
- DF - Data Fetch, First Half
- DS - Data Fetch, Second Half
- TC - Tag Check
- WB - Write Back



MIPS R4000 – Key Features

- Program counter, instruction register, control unit, FPU.
- 32 x 32-bit/64-bit registers, denoted $0-$31 (or r0-r31)
- CPU registers can be either 32 bits or 64 bits wide, depending on the R4000 processor mode of operation.
  - $0 (r0) always stores 000...000
  - $31 (r31) is the link register used by Jump and Link instructions. It should not be used by other instructions.
- Arithmetic and Logic unit
  - Takes input from up to two registers
  - Sends output to registers only
  - **One exception:** also used to calculate memory addresses

Note: In the remainder of lecture, we will talk about 32-bit registers only and denote them using $ notation.

Assembly Language

- The **binary object** (strings of 0's & 1's) file is the only form a computer can be given software.
- Computers can be programmed to translate software expressed in other forms into binary object code.
  - This process includes assembling.
- Assembly language is an alternative form of machine language that uses letters and normal numbers so people can understand it

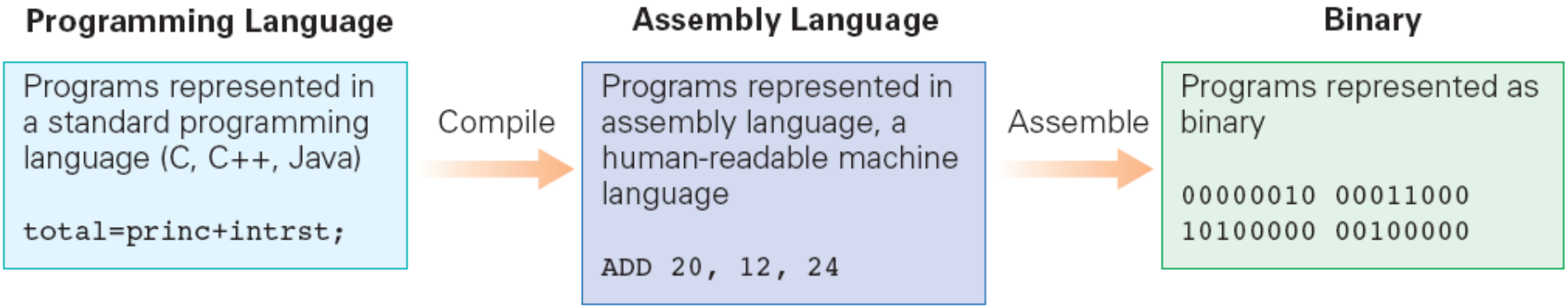
  1) Computer scans assembly code

  2) As it encounters words it looks them up in a table to convert to binary

  3) Converts numbers to binary, then assembles (puts together) the binary pieces into an instruction

Programming Languages

- Most modern software is written in a **high-level programming language**
- High-level languages are compiled (translated) into assembly language, which is then assembled into binary
- These languages are preferred because they have special statement forms that help programmers describe the complicated tasks they want done.



Instruction Sets

- Computers do not understand Java by themselves
  - Several layers of software between your code and the machine itself
- **Java**: machine independent high-level language
  - Runs on anything with a JVM
  - Language has many sophisticated features
  - Can do complex things with little code
- **Machine Level**: programs composed of complex combinations of simple “instructions”
- Each machine has a unique “*instruction set*” also known as ISA (Instruction Set Architecture)

Instruction Sets

- The instruction set is what the programmer “sees” and “uses”
- Set of operations that the CPU can execute
- Each MIPS instruction has 32-bit representation
  - Binary representation – **machine code**
  - Human readable **assembly language**
- High-level program, describe “what” and “how”
- Low-level program, also specify “where”

MIPS Assembly Language program for n!

```
 1  factorial:
 2  bgtz  $a0, doit          # Argument > 0
 3  li    $v0, 1             # Base case, 0! = 1
 4  jr    $ra                # Return
 5  doit:
 6  subi  $sp,8              # Allocate stack frame
 7  sw    $s0,($sp)          # Position for argument n
 8  sw    $ra,4($sp)         # Remember return address
 9  move  $s0, $a0           # Push argument
10  sub   $a0, 1             # Pass n-1
11  jal   factorial          # Figure v0 = (n-1)!
12  mul   $v0,$s0,$v0        # Now multiply by n, v0 = n*(n-1)!
13  lw    $s0,($sp)          # Restore registers from stack
14  lw    $ra,4($sp)         # Get return address
15  addi  $sp,8              # Pop
16  jr    $ra                # Return
```

MIPS Instructions

Load/Store	Load word	lw $1 , &a	Load contents of address &a into register r1
	Store word	sw $1 , &a	Store contents of register r1 into address &a
	Load immediate	li $1 , 100	Load value 100 into $1
Arithmetic	Add	add $1,$2,$3	r1 = r2 + r3
	Subtract	sub $1,$2,$3	r1 = r2 - r3
	Add immediate	addi $1,$2,100	r1 = r2 + 100
	Add unsigned	addu $1,$2,$3	r1 = r2 + r3
	Subtract unsigned	subu $1,$2,$3	r1 = r2 - r3
	Add immediate unsigned	addiu $1,$2,100	r1 = r2 + 100
Multiply/Divide	Multiply	mult $1,$2,$3	r1 = r2 x r3
	Multiply Unsigned	multu $1,$2,$3	r1 = r2 x r3
	Multiply Immediate	multi $1,$2,4	r1 = r2 x 4
	Divide	div $1,$2,$3	r1 = r2 / r3
	Divide unsigned	divu $1,$2,$3	r1 = r2 / r3
	Divide Immediate	divi $1,$2,5	r1 = r2 / 5
Logical	AND	and $1,$2,$3	r1 = r2 & r3
	OR	or $1,$2,$3	r1 = r2 | r3
	NOR	nor $1,$2,$3	r1 = !(r2 | r3)
	AND immediate	andi $1,$2,100	r1 = r2 & 100
	OR immediate	ori $1,$2,100	r1 = r2 | 100
	XOR immediate	xori $1,$2,100	r1 = r2 Î00
	Shift left logical	sll $1,$2,10	r1 = r2 « 10
	Shift right logical	srl $1,$2,10	r1 = r2 » 10

https://course.ece.cmu.edu/~ece447/s13/lib/exe/fetch.php?media=mips_r4000_users_manual.pdf

Very Simple Example

- How to add two numbers?
- In Java: a = b + c;
- In MIPS assembly language, need to know:
  - Where is b?
  - Where is c?
  - Where to put the result (where is a)?
  - What instruction(s) should be used?

Very Simple Example

- a = b + c
- We need to
  - Put **b** and **c** into the registers
    - **Load** them from the memory
  - Add them together
    - The result will go into a register
  - Put the result into the memory location of **a**
    - **Store** it

Very Simple Example

a = b + c

Load b:	lw $8, &b
Load c:	lw $9, &c
Add:	add $10, $8, $9
Store as a:	sw $10, &a

Another Example

a = b + c – d

Load b:	lw $8, &b
Load c:	lw $9, &c
Load d:	lw $10, &d
Add:	add $11, $8, $9
Sub:	sub $11, $11, $10
Store as a:	sw $11, &a

A Bit More Complex Example

```
if (a == 1) b = 0;
else b = 1;
```

```
Load a:                lw $8, &a
Value 1:               li $9, 1
If a != 1:             bne $8, $9, L1
b = 0:                 li $10, 0
jump:                  j L2
b = 1:          L1:    li $10, 1
End:            L2:    sw $10, &b
```

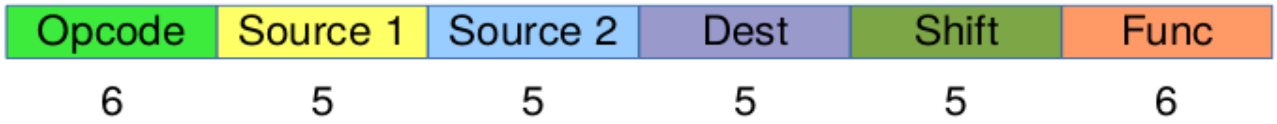
Branch Not Equal (BNE)
If $8 != $9, jump to instruction from label L1

Some instructions use one register, some use two, some use three, some take values, some take addresses. How?

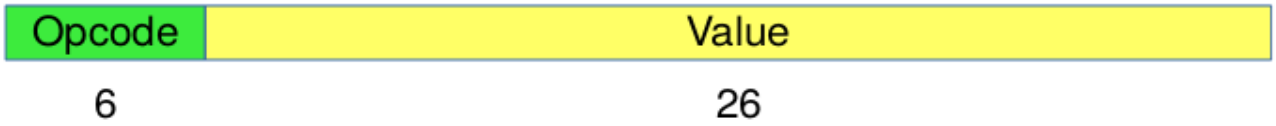
MIPS Instruction Types

- I-Type — Immediate
  - Require one or two registers and a value

- R-Type — Register
  - Require three registers



- J-Type — Jump
  - Require a value



A For Loop

```
y=0;
for (i=1; i<=x; i++) {
    y = y + i;
}
```

How can we write this in MIPS Assembly ?

Hint: bgt

Summary

- We have seen the von Neumann Architecture and compared it with Harvard Architecture.
- We have been introduced to pipelined execution of instructions.
- We've also seen some detail on MIPS R4000 and programmed it in its own language
- Examples of constructing "useful" programs at the machine level, as you learn to program in Java,