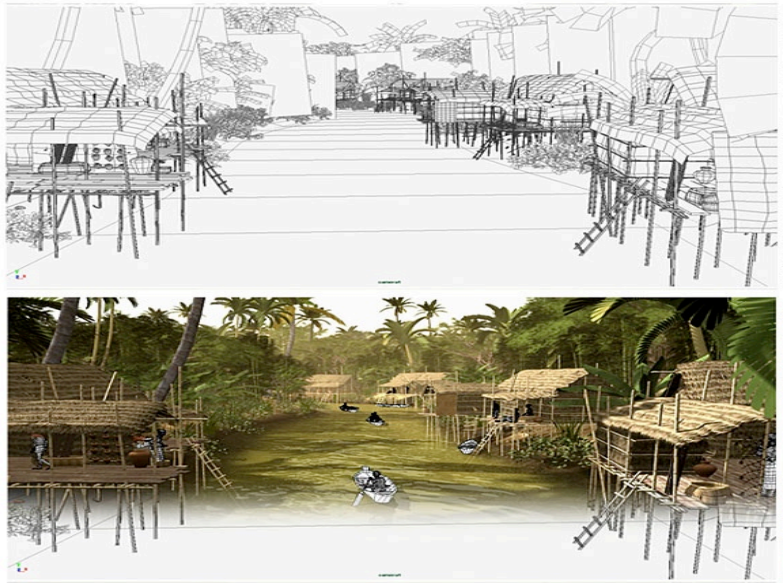
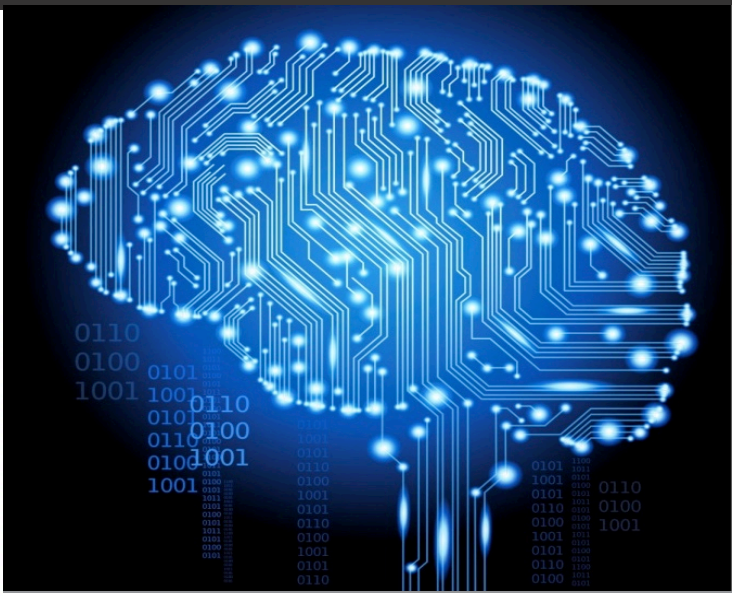
MONASH University

Information Technology

FIT1008/2085
MIPS – Isn't learning this a waste of time?

Prepared by:
Maria Garcia de la Banda
Revised by A. Aleti, D. Albrecht, G. Farr, J. Garcia and P. Abramson





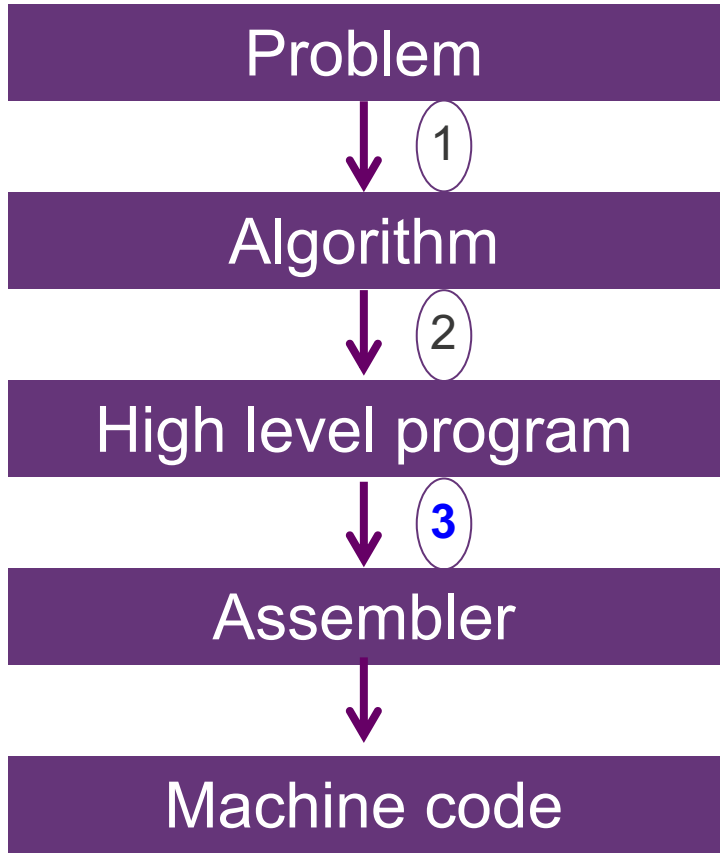
Learning objectives for this lesson block

- **To understand why we are going to learn about MIPS**
- **In the process, to learn about:**
  - von Neuman architecture
  - Machine code and Machine language
  - The concept of “word" in computer architecture
  - RISC vs CISC

Make sure you are comfortable with integer representation. We will assume you are!

We said that FIT1008/2085 ...

- **Were about the first three steps**
- **You have already done a bit on the first two**
- **But the third step might still be a mystery for you**

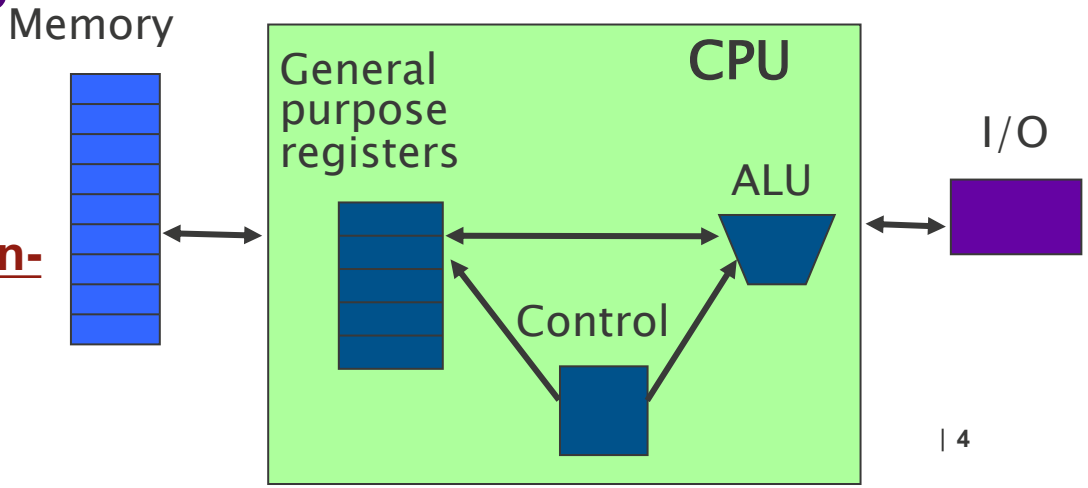


But what is this?

von Neumann architecture (1945)

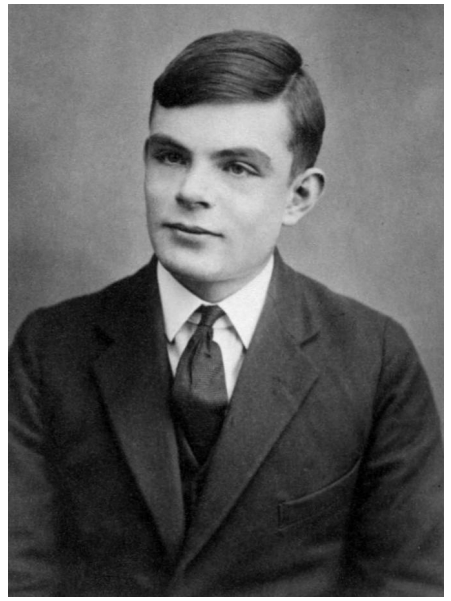
- **Consists of three main components:**
  - **Memory**: stores both data and instructions
  - Control Processing Unit (**CPU**) divided into:
    - Arithmetic/Logic Unit (**ALU**): does the calculations
    - **Registers**: store temporary results
    - **Control:** decides what to do next
  - Input/Output (**I/O**)
- **Instructions & data must be loaded into the CPU before being processed**
- **If interested, look into**

https://www.microcontrollertips.com/difference-between-von-neumann-and-harvard-architectures/

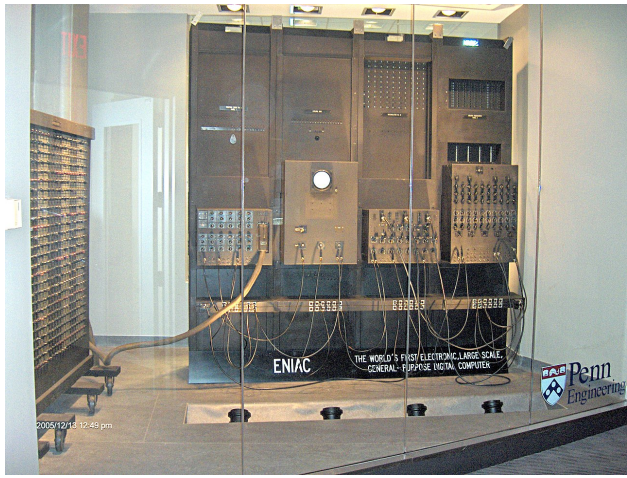


Others who deserve credit

By Unknown author - http://www.turingarchive.org/viewer/?id=521&title=4, Public Domain, https://commons.wikimedia.org/w/index.php?curid=22828488



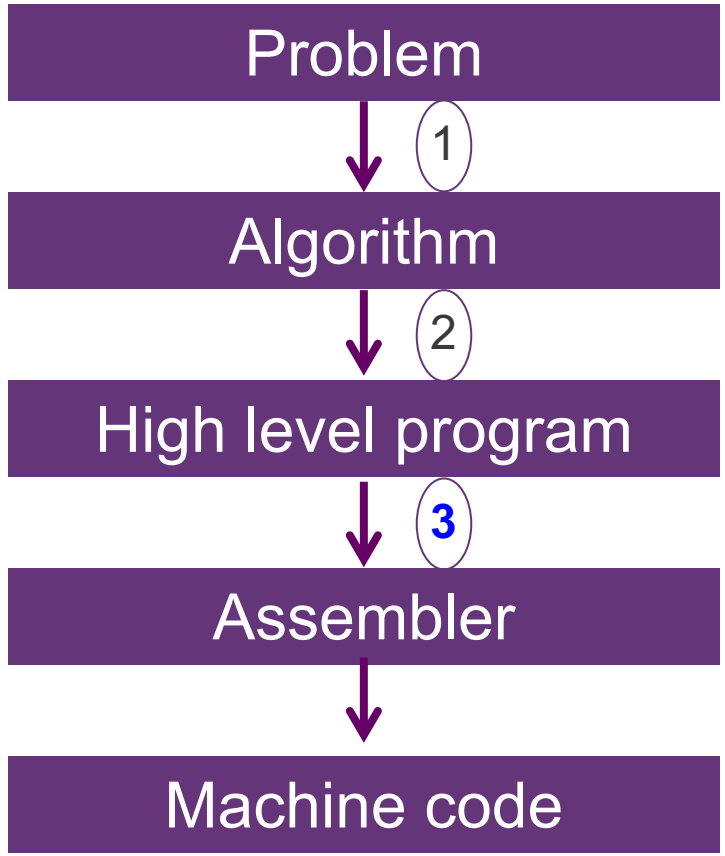
- **Alan Turing: in 1936 published the Universal Computing Machine**
  - Also has a memory containing both instructions and data
  - von Neumann interacted with Turing in 1935-1937
- **Konrad Zuse who in 1936 had two patents on the same**
- **Presper Eckert & John Mauchly: developed ENIAC, when designing EDVAC in 1943 wrote about similar designs**
  - von Neumann joined the project and wrote the first draft report
  - A colleague circulated it with only von Neumann's name



By The original uploader was TexasDex at English Wikipedia. - Transferred from en.wikipedia to Commons by Andrei Stroe using CommonsHelper., CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=6557095

We said that FIT1008/2085 ...

- **Were about the first three steps**
- **You have already done a bit on the first two**
- **But the third step might still be a mystery for you**



But what is this?

Machine Code and Machine Language

- **The CPU runs machine code written in machine language**
  - Not in Java, C, Python, JavaScript, or assembler
- **Each CPU type usually requires a different machine language**
- **Machine code is stored in memory as bit patterns**
- **Example: the machine code to compute the factorial of a number for a "MIPS" R2000 CPU consists of the following bit patterns:**

```
00110100000000010000000000000001
00101000100000010000000000000010
00010100001000000000000000000100
01110000010001000001000000000010
00100000100001001111111111111111
00001000000100000000000000001010
00000011111000000000000000001000
```

Each line of 32 bits is a different machine code instruction

Loading and storing… what?

- **Words!**
  - But not the kind of words you think…
- **In computer architecture, a word is a chunk of bits (8, 16, 32 or 64) used as basic unit of data in a computer:**
  - That is, the basic block the computer uses to store, operate on, move, etc
  - Depends on the computer
  - For MIPS we will use 32 bits, that is, 4 bytes
  - That is why MIPS machine code instructions are 32 bits

Why is Assembly Language needed?

- **Problem with machine language:**
  - Very difficult to write and read for humans
- **Need a compromise, a language that:**
  - Supports comments, variable names, line labels, etc
  - Has human-readable versions of machine instructions
  - But is easily converted to machine language
    - Usually a 1-to-1 relationship between each assembler language instruction and its equivalent machine language instruction
- **Languages of this kind called assembly languages**

The first three weeks are all about ????????

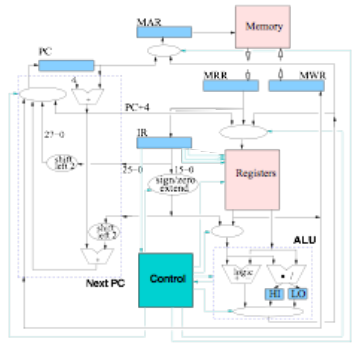
You

```
f = 1
n = int(input("Enter int:"))
while n > 0:
    f = f*n
    n = n-1
print(f)
```

Human-readable code

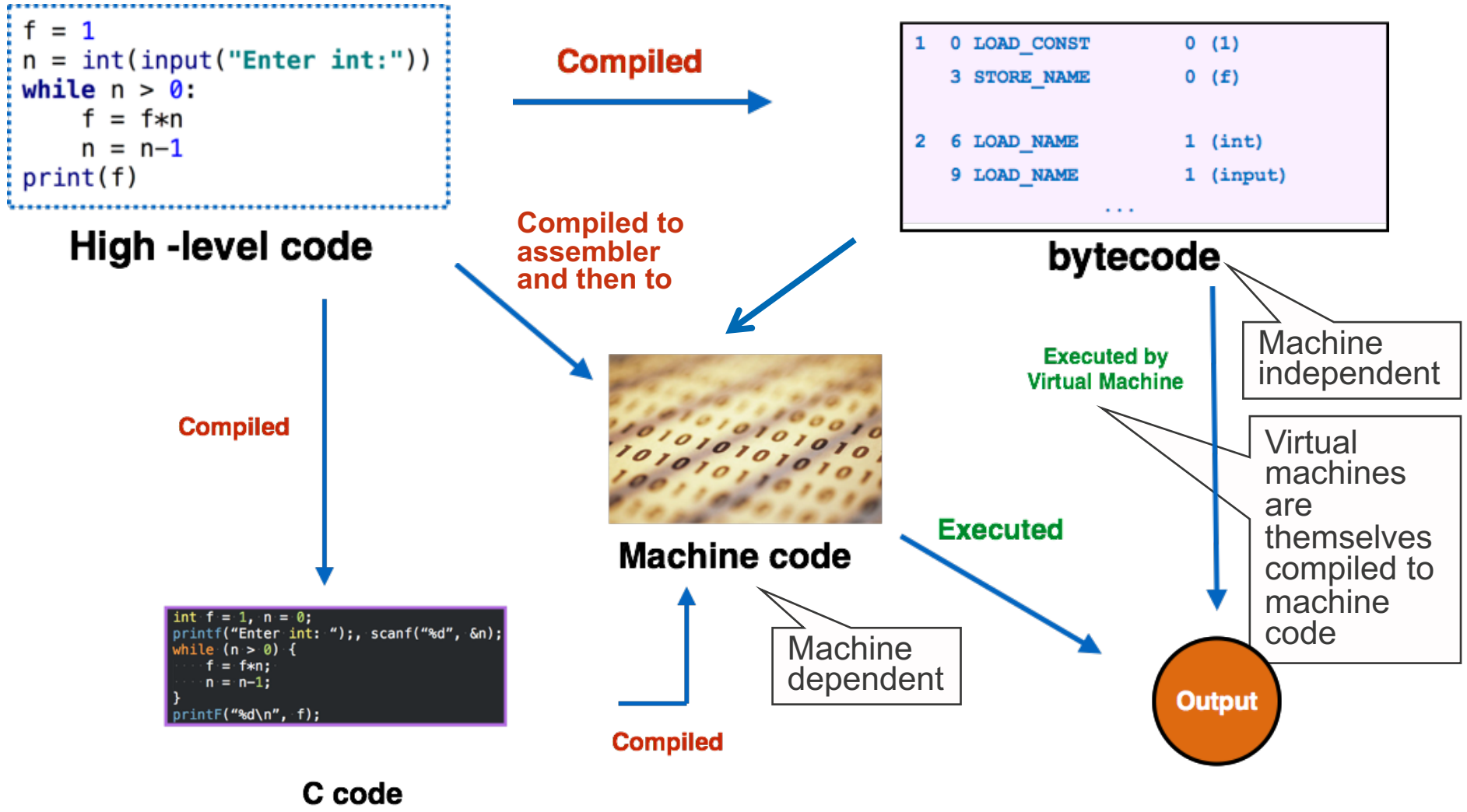
????????

Machine code



CPU

Nowadays, there are lots of possibilities



Why do you need to learn about Assembly Language?

- **All high level code ends up being translated into some kind of assembler code (bytecode is a kind of assembler)**
- **Understanding the translation will make you really "get" how high level code works, from variables to loops, if-then-elses, and function calls**
  - It will make you a MUCH valuable programmer
- **You might need to write in it when timing is critical or when memory size is limited**
  - E.g., device drivers or embedded computers
- **You might need to read it**
  - E.g., to inspect the optimisations made by the compiler

- **In this unit we will study the MIPS assembly language**

MIPS Architecture

- **1981: John L. Hennessy starts a research group at Stanford, focusing on RISC (reduced instruction set computer) architectures**
- **1984: takes a year off to commercialize his research**
  - Founds MIPS Computer Systems
  - Now MIPS Technologies (www.mips.com)
- **MIPS name:**
  - "Microcomputer without Interlocking Pipeline Stages"
  - Also a pun on "Millions of Instructions Per Second"
- **R2000 model (1985)**
  - First and simplest of MIPS processors
  - Later MIPS models extend basic architecture

By Eric Chan - originally posted to Flickr as Provost John Etchemendy (However this picture shows President John L. Hennessy instead of Provost John Etchemendy), CC BY 2.0, https://commons.wikimedia.org/w/index.php?curid=6504543

Why MIPS?

- **A real processor (not a toy one)**
  - MIPS32 & MIPS64 still in production
- **Ancestor (because of RISC) of many popular computers**
  - Apple/IBM/Motorola PowerPC (Macintosh), Digital Alpha (Alpha), ARM (3Com Palm and most embedded)
- **Knowledge of MIPS can be easily carried over to these other architectures**
  - Again, we are learning the fundamentals (and, thus, transferrable knowledge)
- **Also used in many embedded systems**
  - Sony Aibo, Sony Playstation 1 & 2, Sony PSP, Nintendo 64, HP Laser Printers, Minolta digital camera, lots of routers and network appliances

RISC vs CISC

- **MIPS was first computer to use the term RISC**
  - **R**educed **I**nstruction **S**et **C**omputer
  - All instructions are
    - Same length (4 bytes, that is, 32 bits – a word)
    - Of similar complexity (simple)
    - (Mostly) able to run in same time (1 clock cycle)
    - Easily decoded and executed by computer hardware
  - Advantages: easier to build, cheaper, consumes less power
  - RISC is again all the rage thanks to RISC-V open source instr. set
- **Intel x86 is considered CISC**
  - **C**omplex **I**nstruction **S**et **C**omputer
  - Instructions vary in length, complexity and execution time
  - Decoding & running instructions requires hardware-embedded code (microcode)
  - Advantage: potential for optimisation of complex instructions

Why not Intel 80x86?

- **MIPS is a simple, clean architecture**
  - Easier to learn
  - x86 architecture is cumbersome with many confusing addressing modes and exceptions
- **MIPS is representative of modern computer architecture**
  - Easier to transfer knowledge
- **MIPS has readily available simulators:**
  - http://courses.missouristate.edu/KenVollmar/MARS/
- **RISC architectures are gaining in popularity!**
  - So knowledge or RISC might be good for getting a job…

There are many RISC architectures, including

- SuperH (with open J-2 implementation).
- ARM (fully proprietary)
- MIPS (open, royalty-free)
- Power (open, royalty-free)
- AVR (proprietary)
- SPARC (open, royalty-free)
- OpenRISC (open, royalty-free)
- RISC-V (open, royalty-free)
  - This is the one that has really caused the **rise in popularity**!

https://hackaday.com/2019/11/12/risc-v-why-the-isa-battles-arent-over-yet/

We use MIPS mainly for simplicity, clarity, and availability of simulators

Going deeper than you have with MARIE

- **Some students have already learned about MARIE**
  - A simplified CPU and ISA architecture
- **Everything you learned for that is useful here too**
- **With MIPS you will go deeper by:**
  - Using more realistic instructions for if-then-else and loop conditions
  - Learning about more realistic function call/return
  - Dealing with complex memory (arrays)
  - Using system calls for input/output/memory

Summary

- **To understand why we are going to learn about MIPS**
- **In the process, to learn about:**
  - von Neuman architecture
  - Machine code and Machine language
  - The concept of “word" in computer architecture
  - RISC vs CISC

MONASH University

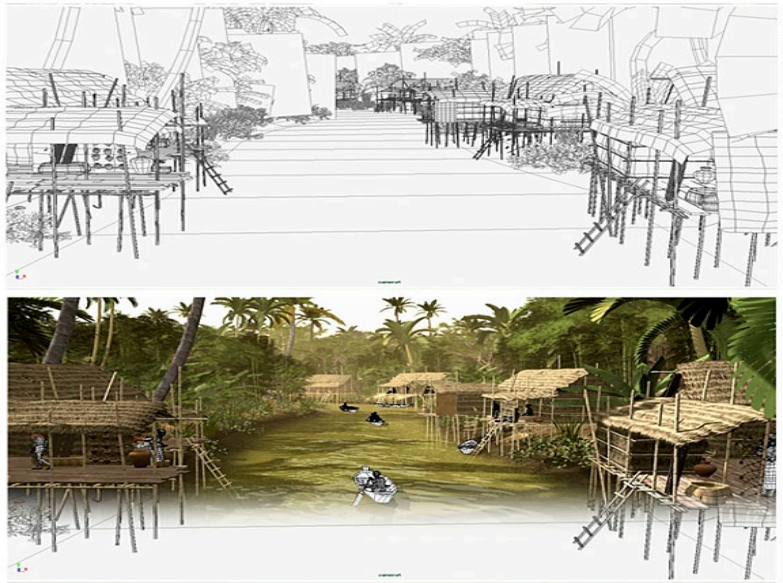
Information Technology

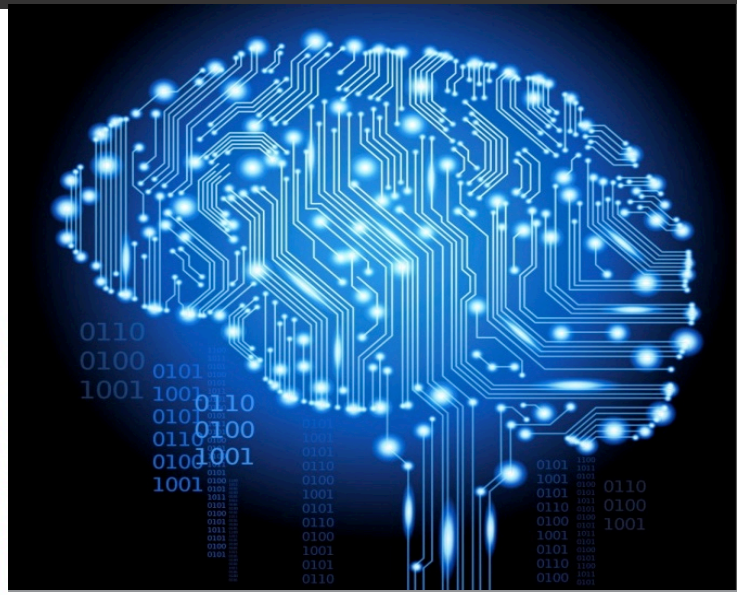
FIT1008/2085
MIPS – Architecture

Prepared by:

Maria Garcia de la Banda

Revised by A. Aleti, D. Albrecht, G. Farr, J. Garcia and P. Abramson





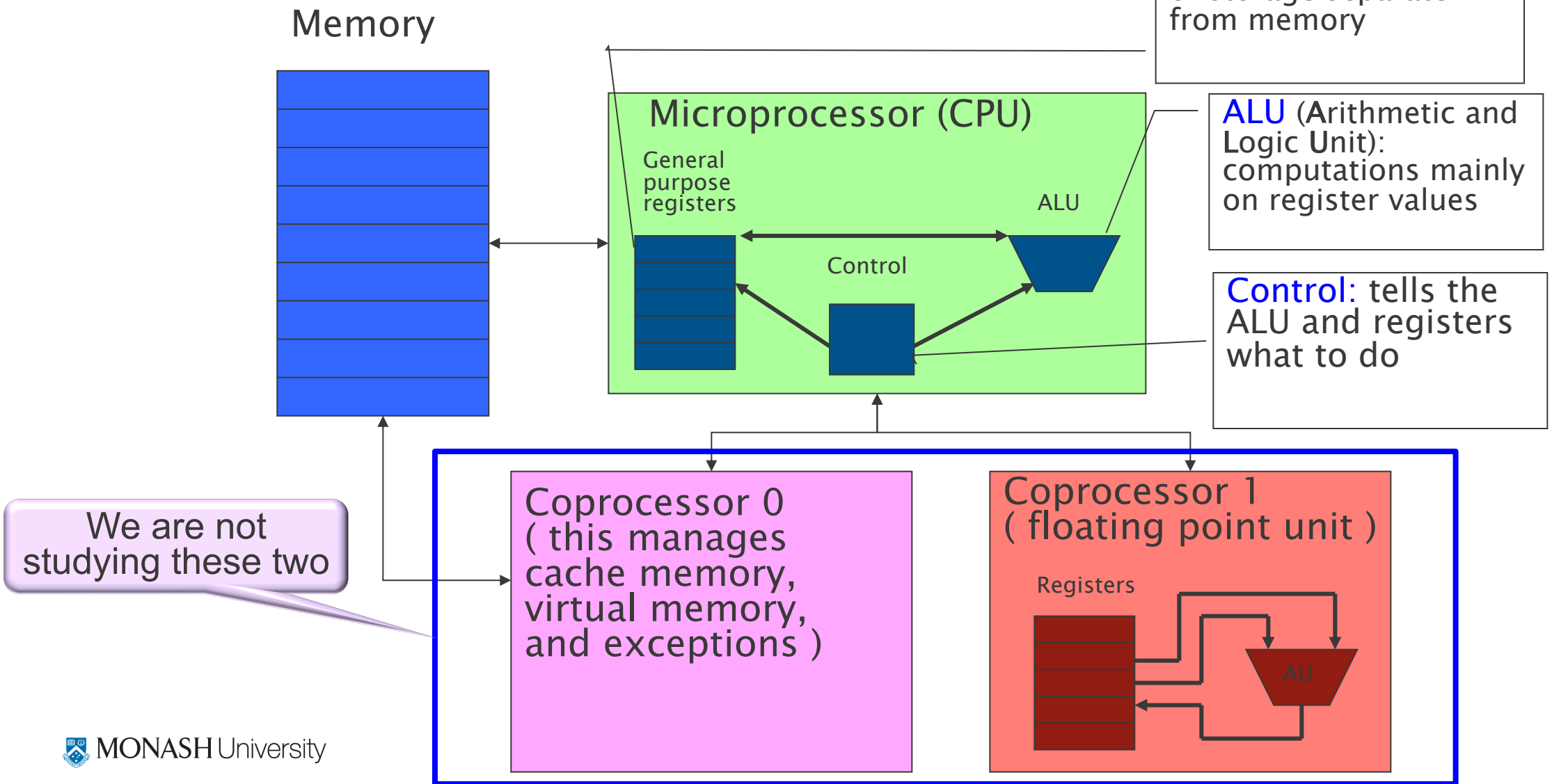
Where are we at

- **We have seen how learning MIPS is useful**
- **In the process, learned about:**
  - von Neuman architecture
  - Machine code and Machine language
  - The concept of “word" in computer architecture
  - RISC vs CISC

Learning objectives for this lesson block

- **To understand the basics of MIPS R2000 architecture**
- **To understand the CPU's (general and special) purpose registers**
  - Their aim, use and location in the CPU
- **To understand how programs are executed in this architecture**
  - The fetch-decode-execute cycle

MIPS Architecture (simplified)



General purpose registers: fast words of storage separate from memory

ALU (Arithmetic and Logic Unit): computations mainly on register values

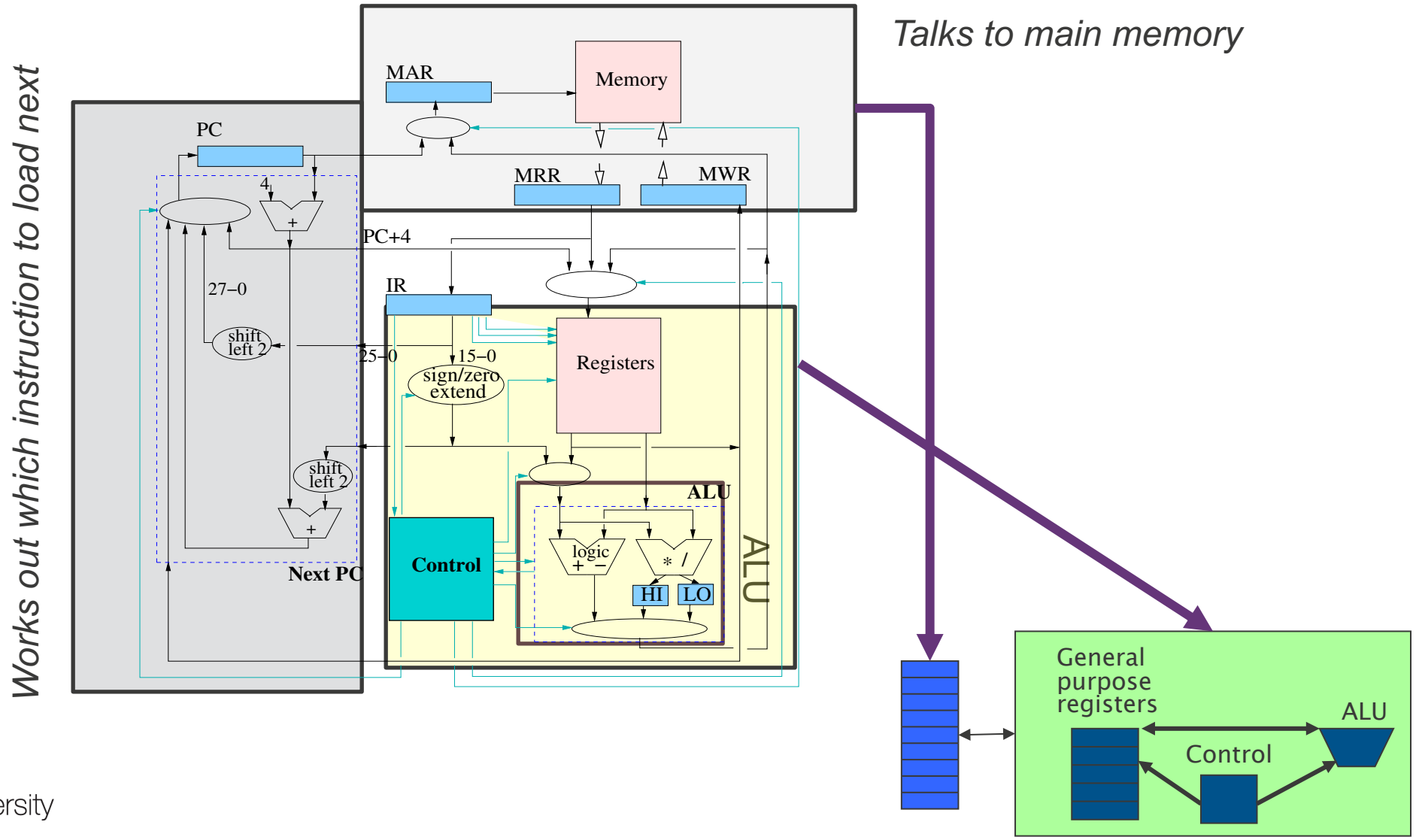
Control: tells the ALU and registers what to do

We are not studying these two

Main components: basics

- **32 General-purpose registers**
  - Fast but expensive memory
  - Physically located on the CPU chip
  - Each 32 bits in size (a word)
- **Arithmetic Logic Unit (ALU)**
  - Performs computations on register and integer values (not main memory):
    - As it is a von Newman architecture
    - Integer and bitwise arithmetic (including comparisons)
- **Several special-purpose registers**
  - PC (Program Counter)
  - HI, LO (multiplication/division results)
  - IR (Instruction Register)
  - MAR/MRR/MWR (Memory Address/Read/Write Register),

Inside the MIPS microprocessor (not simplified at all!)



General Purpose Registers (GPRs)

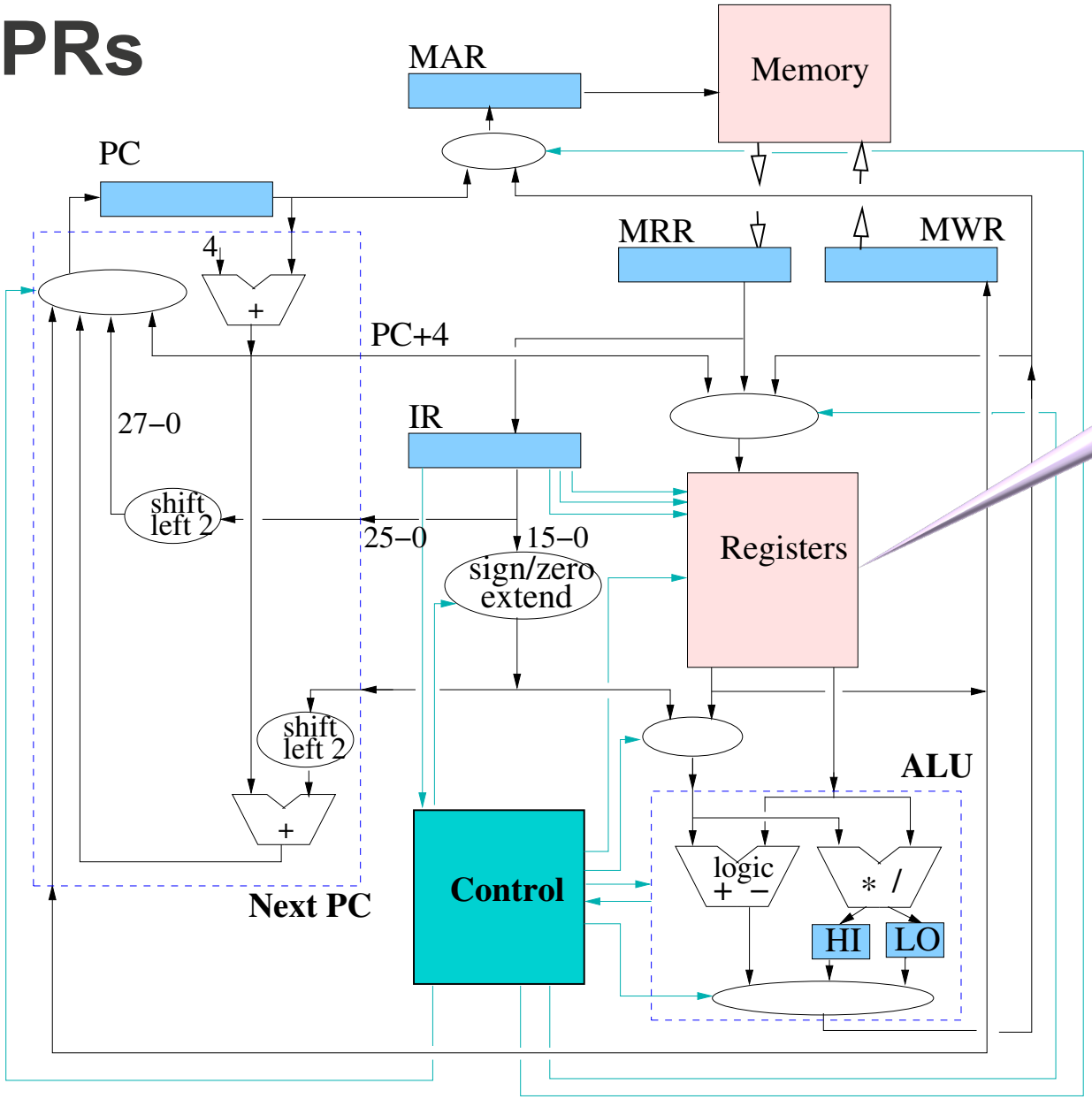
- **Prefixed with $ in assembly language**
  - Numbered `$0` to `$31`
  - Also given names based on usage conventions, e.g.:
    - `$0` ⇔ `$zero` (special case read-only register, always set to 0)
    - `$4` ⇔ `$a0`
    - `$29` ⇔ `$sp`
- **Unlike variables, you can't name them yourself**
  - They're hard-coded
- **Names increase readability, so we will use them**
- **Can theoretically be used in any way**
- **Conventions assign certain uses to certain GPRs**
  - Conventions help your program cooperate with others

General Purpose Registers (cont'd)

used by us	Register name	Register number	Typical use
✓	$zero	$0	constant zero, cannot change, read only
✗	$at	$1	assembler uses for pseudoinstructions
✓	$v0, $v1	$2, $3	function return values; system call number
✓	$a0 - $a3	$4 - $7	function and system call arguments
✓	$t0 - $t7, $t8, $t9	$8 - $15, $24, $25	temporary storage (caller-saved)
✗	$s0 - $s7	$16 - $23	temporary storage (callee-saved)
✗	$k0, $k1	$26, $27	reserved for kernel trap handler
✗	$gp	$28	pointer to global area
✓	$sp	$29	top-of-stack pointer
✓	$fp	$30	stack frame pointer
✓	$ra	$31	function return address

We will use the names, not the numbers (except for $0)

MIPS GPRs



The 32 General Purpose Registers are here

A quick look at arithmetic instructions

- What do arithmetic instructions look like? Here are a few examples:

```
sub $t0, $t3, $t1
addi $sp, $sp, -1
xor $a0, $zero, $t5
div $t1, $t2
```

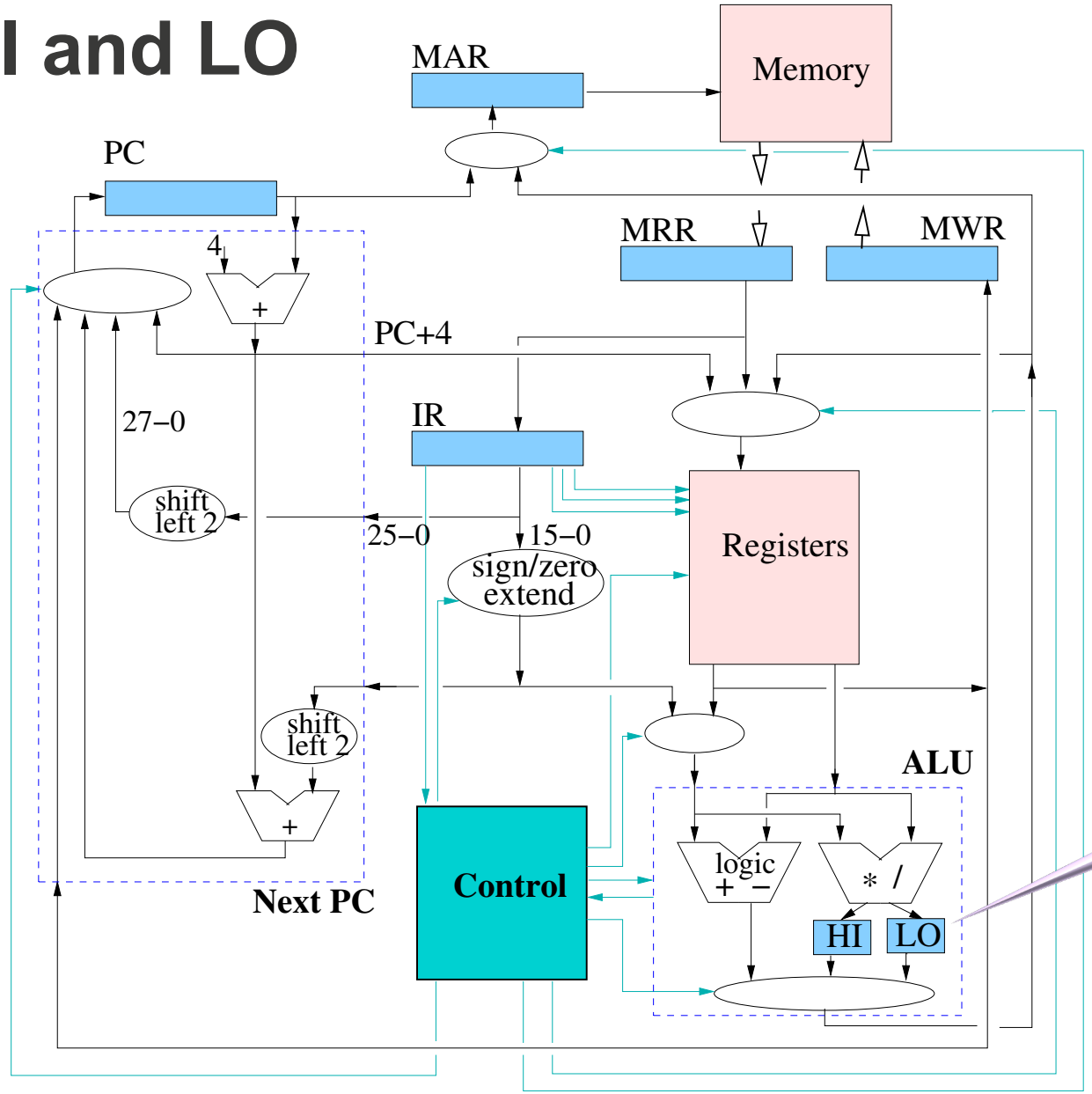
- We'll see more examples next lesson
- Floating-point operations are beyond the scope of this unit

Special Purpose Registers (HI and LO)

- **Multiplying two 32bits numbers might require 64 bits to fit**
- **After an integer multiplication:**
  - HI contains the "high" 32
  - LO contains the "low" 32
- **Integer division (say 5/3) might be used to get the integer result (1) or to get the remainder (2)**
- **After an integer division:**
  - LO contains the integer result (1 above)
  - HI contains the remainder (2 above)
- **There are instructions to move the contents of LO or HI back to a GPR**

Remember, only general purpose registers start with $

MIPS HI and LO



The 32 HI and LO are here

Special Purpose Registers (IR)

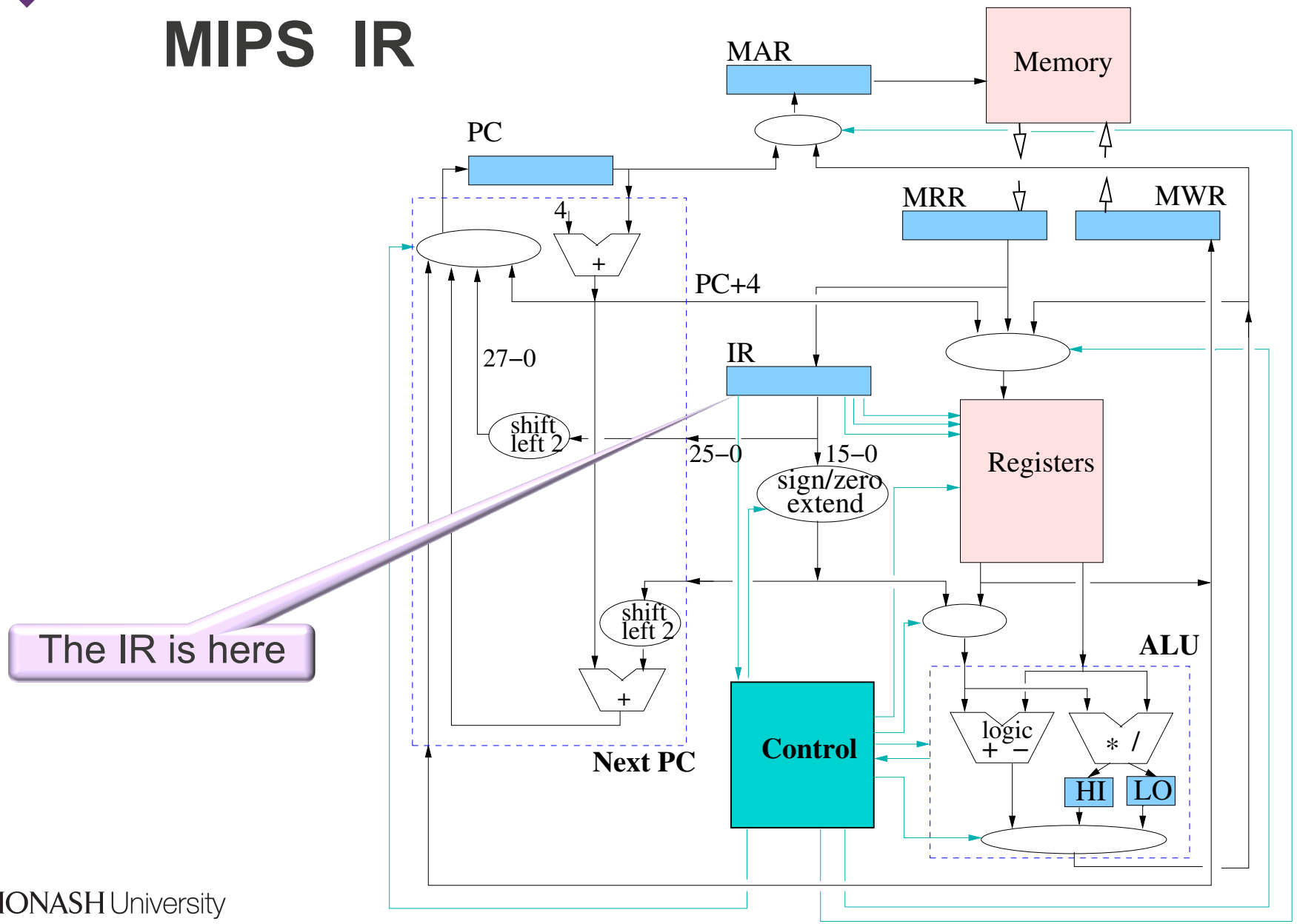
- **The Instruction Register stores the instruction currently being executed (32 bits; 4 bytes; a word)**

```
addi $9, $5,  -3
```

```
001000 00101 01001 1111111111111101
```

- **Bits b31 to b26 inclusive (leftmost 6 bits) are the operation code**
  - opcode encodes the kind of instruction (e.g., `sub`, `addi`, `xor`)
  - Tells the control circuitry which set of microinstructions needs to be followed
- **The rest of the info (operands) depends on the opcode's value:**
  - How many bit fields the remaining IR bits split into
  - How each bit field is used when instruction is run (GPRs? immediate values?)
- **The IR is not visible to the programmer**

MIPS IR

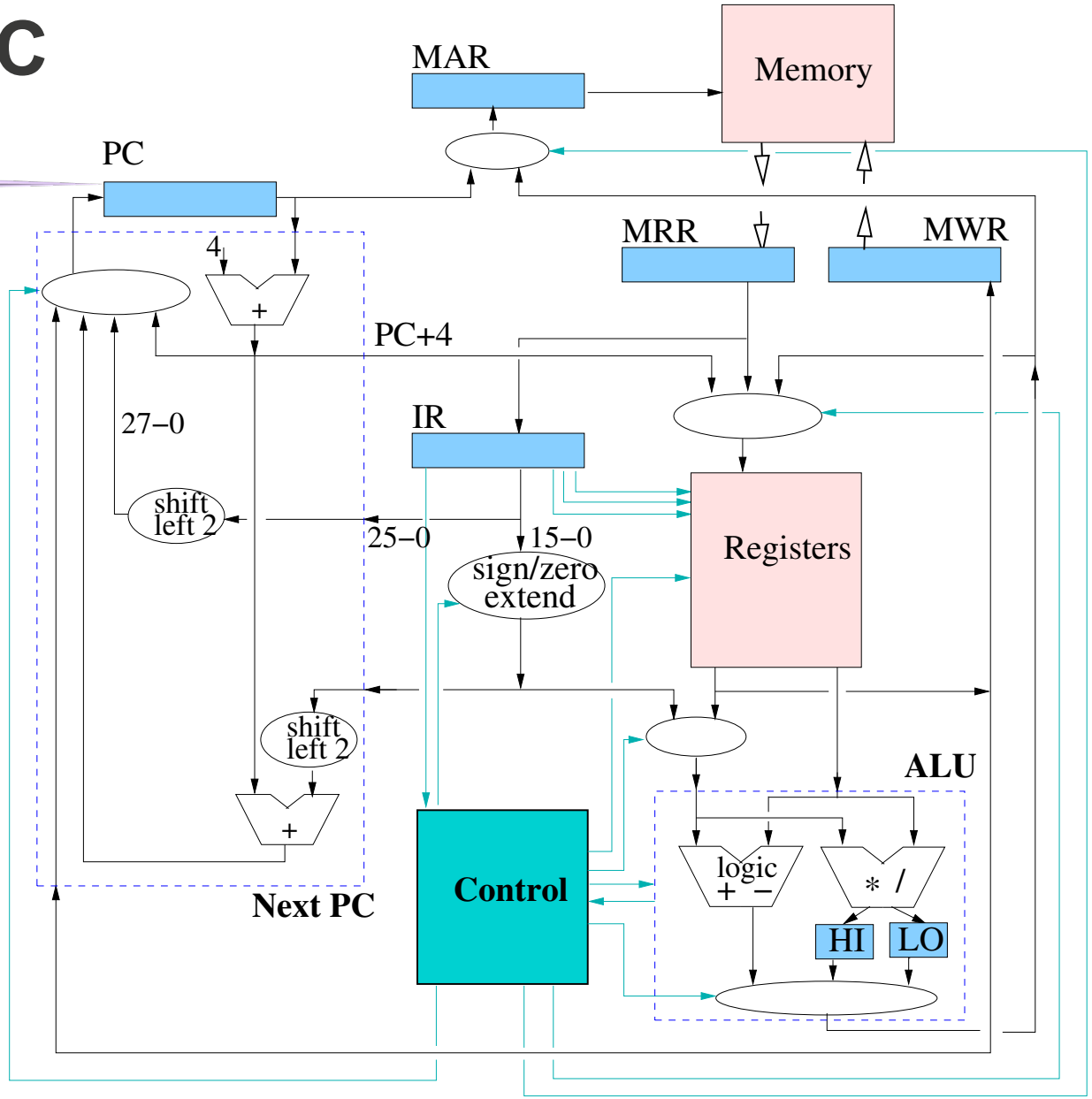

MAR
Memory
PC
MRR
MWR
4
+
PC+4
27–0
IR
shift left 2
25–0
15–0
sign/zero extend
Registers
shift left 2
+
ALU
Control
logic + –
* /
HI
LO
Next PC
The IR is here

Special Purpose Registers (PC)

- **The Program Counter acts as a bookmark:**
  - Tells the computer where is it up to
- **How? It holds the memory address of the machine instruction currently being executed**
- **Advanced by most machine instructions to point to next instruction (PC=PC+4)**
  - Why +4? Because MIPS instructions are 4 bytes (32 bits; a word)
- **Jump instructions write a new value to PC to move execution to a new place in program**
  - Useful for encoding loops (go back to beginning), function calls, etc
- **Branch instructions jump only if a given condition is met**
  - Useful for if-then statements, failed while/for conditions, etc

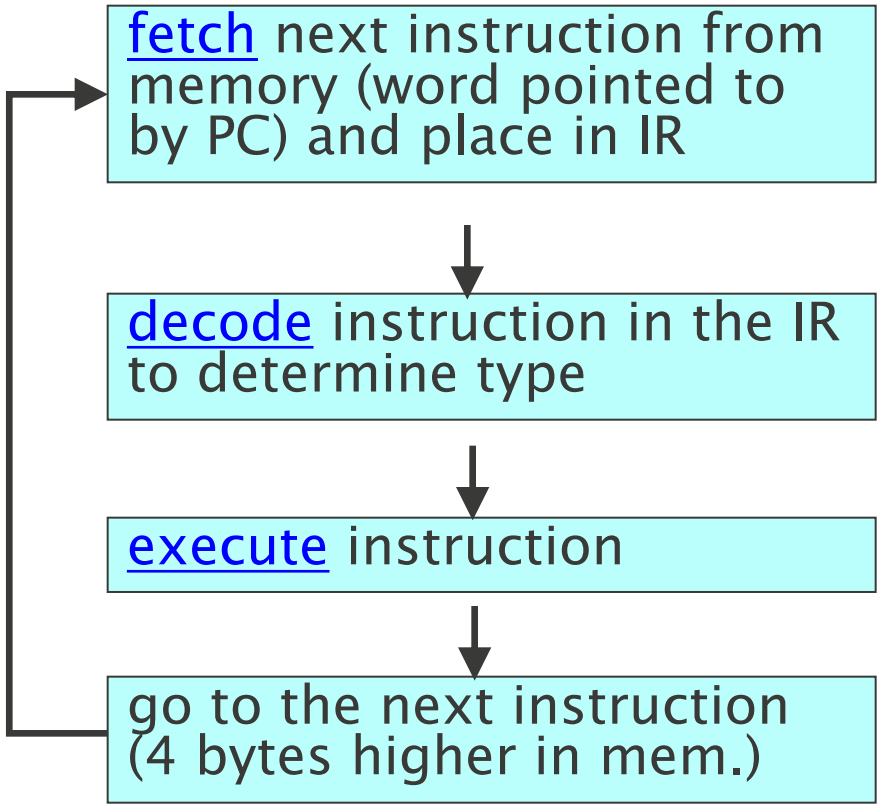
MIPS PC

The PC is here



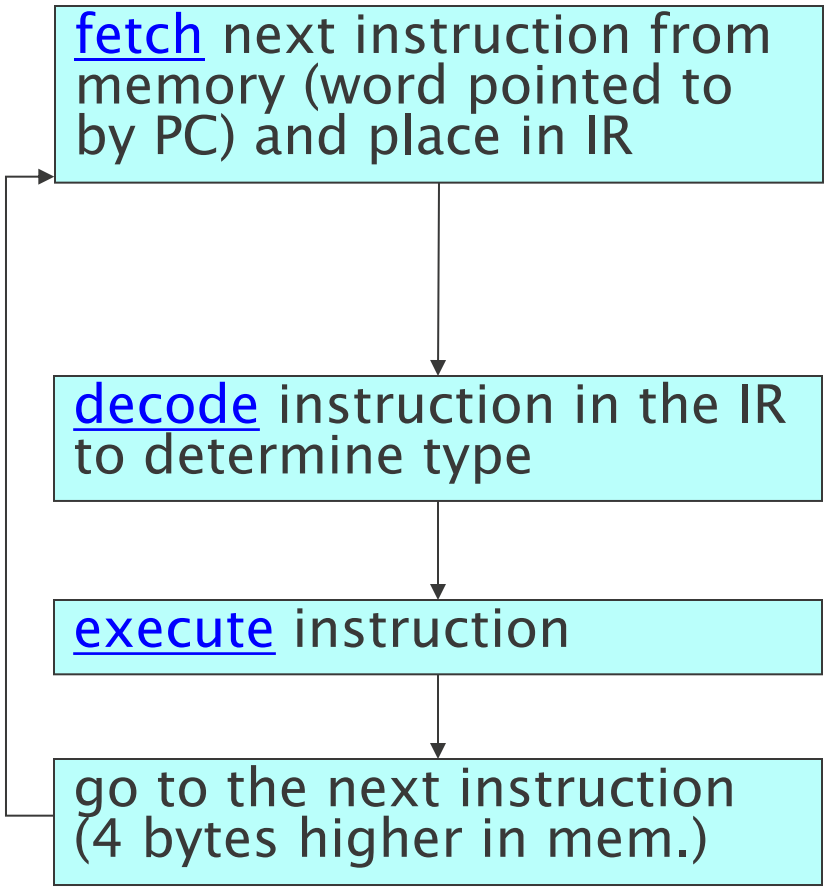
MIPS Instruction Execution

- **Programs are run by the MIPS hardware performing fetch-decode-execute cycles**



MIPS Instruction Execution: Example

I write it in hexadecimal, but it is really stored as 32 bits



Example: instruction word at mem[PC] is 0x20A9FFFD

0010 0000 1010 1001 1111 1111 1111 1101

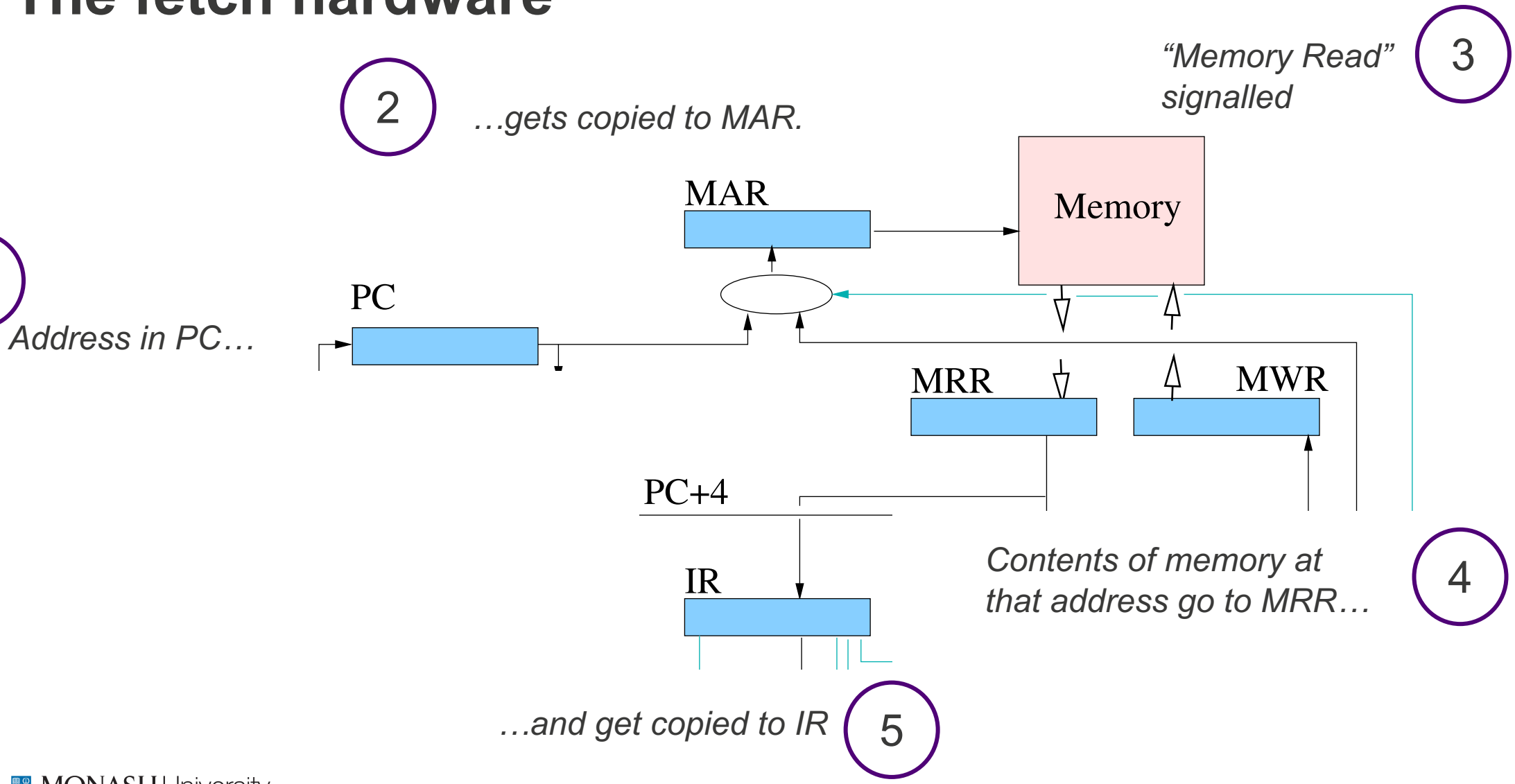
001000 00101 01001 1111111111111101

Opcode 8 is “add immediate”, source reg is $5, “target” reg is reg $9, add amount is -3

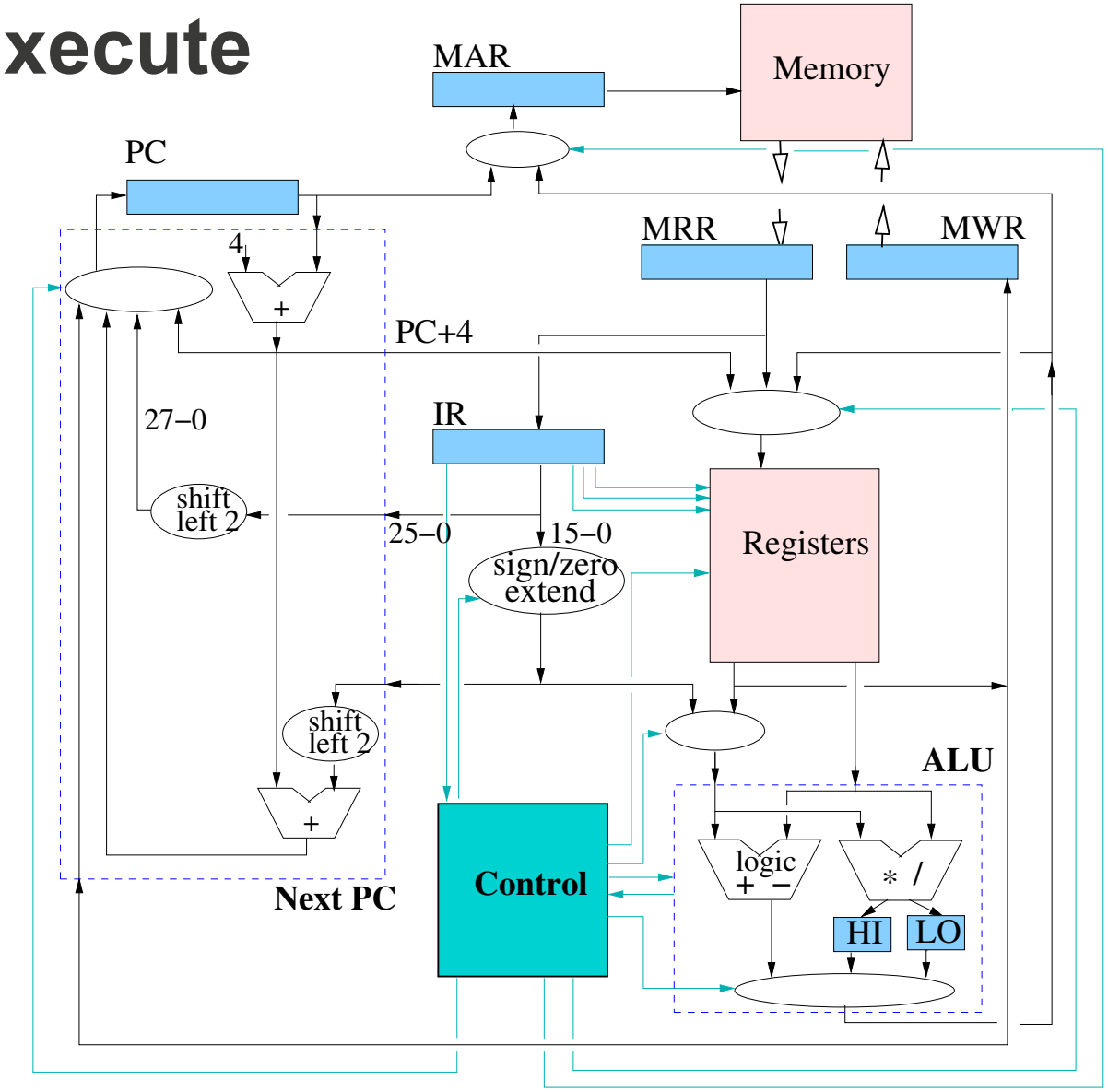
Send reg $5 and -3 to ALU, add them, result to reg $9

PC = PC + 4

The fetch hardware



Decode-Execute



Summary

- **We have learned the basics of MIPS R2000 architecture**
- **We now understand the CPU's (general and special) purpose registers**
  - Their aim, use and location in the CPU
- **We now understand how programs are executed in this architecture**
  - The fetch-decode-execute cycle

MONASH University

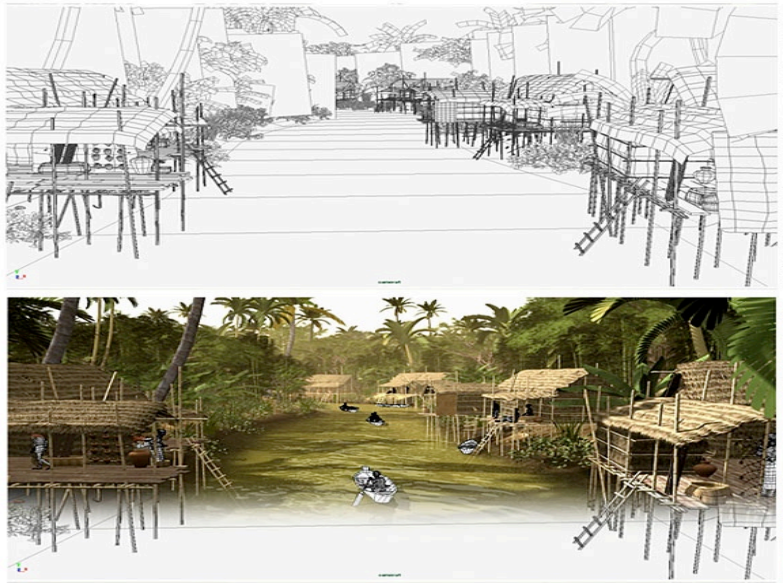
Information Technology

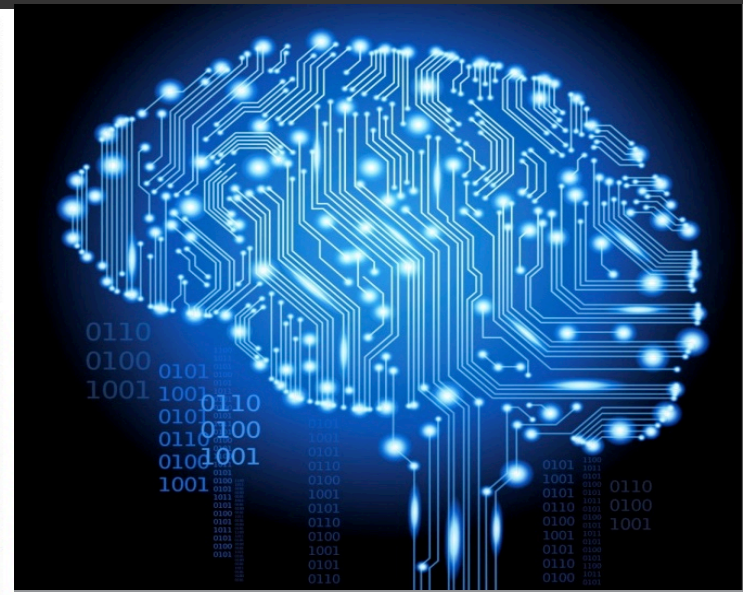
FIT1008/2085
MIPS – Memory

Prepared by:

Maria Garcia de la Banda

Revised by A. Aleti, D. Albrecht, G. Farr, J. Garcia and P. Abramson





Where are we at

- **We have seen how learning MIPS is useful and learned about:**
  - von Neuman architecture
  - Machine code and Machine language
  - The concept of “word" in computer architecture
  - RISC vs CISC
- **We have seen the basics of MIPS R2000 architecture**
- **We are familiar with the**
  - CPU’s components and
  - General and special purpose registers
- **We know how programs are executed in this architecture**
  - The fetch-decode-execute cycle

Learning objectives for this lesson block

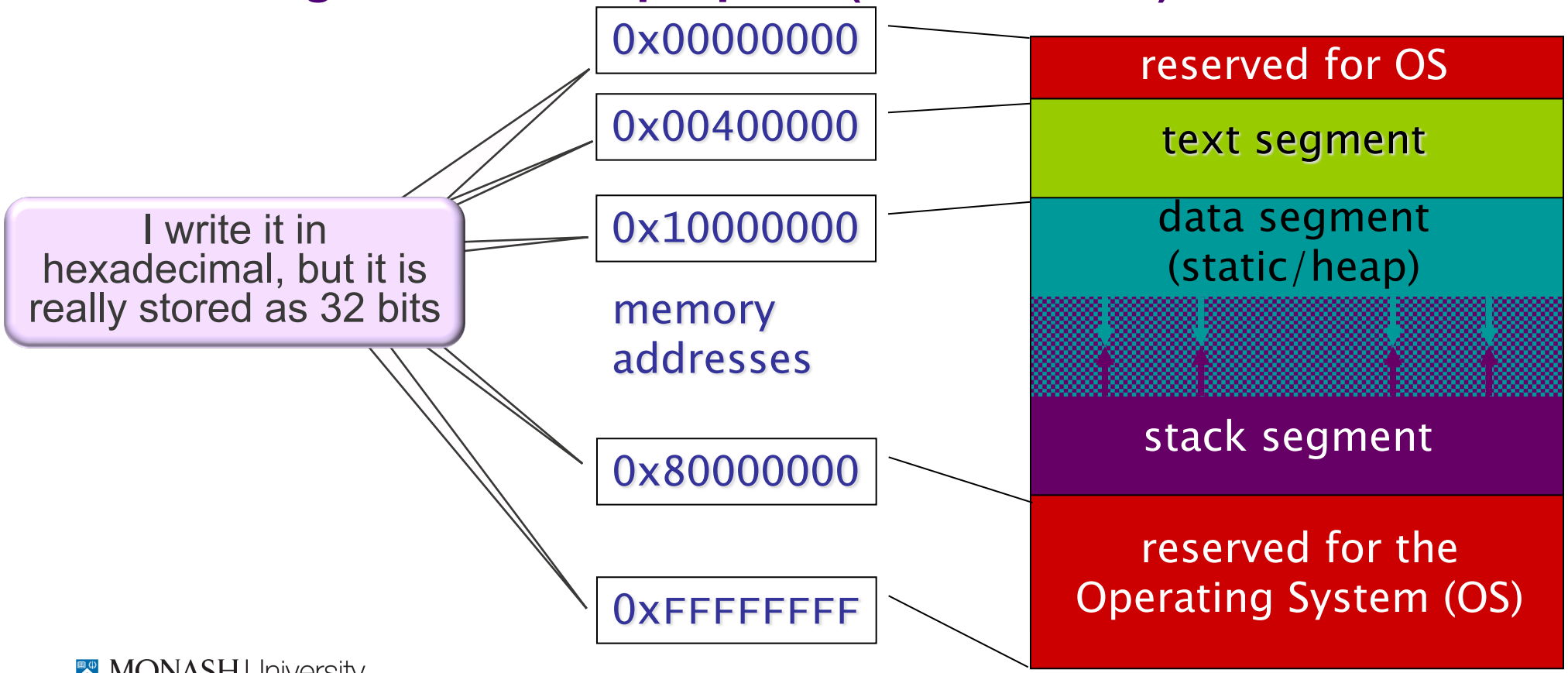
- To understand how MIPS memory is structured and accessed

Accessing Main memory

- **Recall: MIPS is a load-store architecture:**
  - Computations can only consider registers (and integers) as arguments
  - Their result is stored in registers
- **But programs and their data live in main memory (not on the CPU chip)**
  - So we will have to load data into a register to work on it
  - And then store the result (which is in a register) back into memory when we're done
- **To do this loading/storing, we need to know:**
  - Which register we're loading to/storing from, and
  - Where in memory to find/put the data

MIPS memory architecture

- **Memory is divided into segments (from address A to address B excluded)**
- **Each segment has its purpose (will see later)**



Memory addresses

- **Memory addresses in MIPS are 32 bits long and unsigned**
- **What is the range then? (i.e., smallest-largest values)**
  - Smallest possible is 0x00000000
  - Largest possible is 0xFFFFFFFF, which is $2^{32} - 1$ (so the range is $2^{32} - 1$)
- **Each address refers to one byte of memory (so 8 bits)**
  - Total potential address space: 4 Gb

BTW: Looking for meaning in memory

- **What does "01011010" mean?**
  - Number 90?
  - ASCII character 'Z'?
  - Instruction DECB for Motorola 6800 CPU?
  - Just garbage bits?
- **Depends on context**
  - Bits have no intrinsic meaning; their value depends on how it is being used
  - If interpreting as number, has value 90
  - If interpreting as character, has value 'Z'
- **This is why memory is divided in segments**
  - Each 32 bits in, for example, the text segment is known to be a MIPS instruction
- **And is why program variables (or values) have types**
  - So that the bits in the data segment can be understood

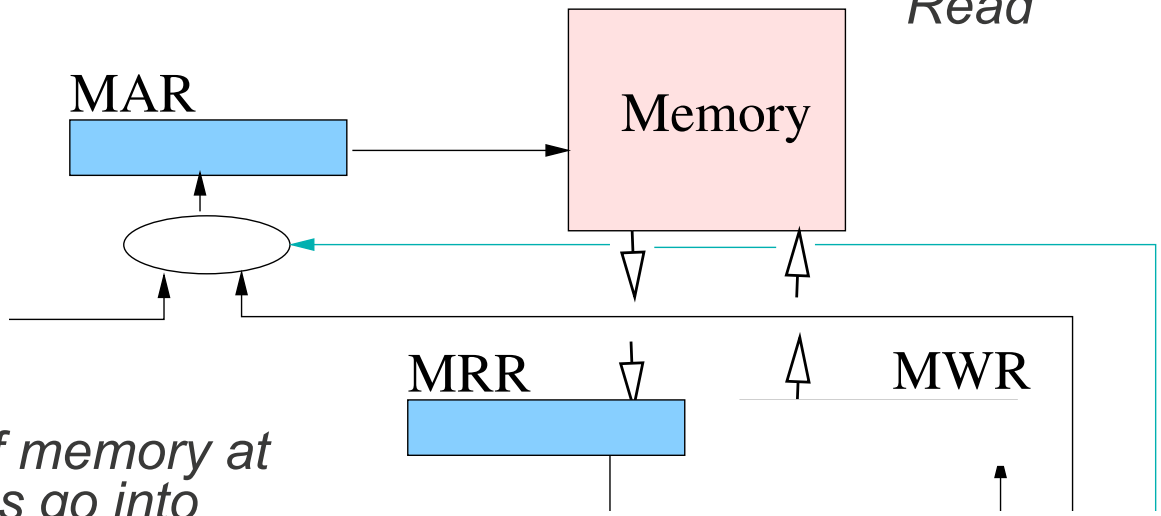
Getting data out of main memory

- **We use a load instruction for this**
  - We can load 1, 2, or 4 bytes at a time
  - In FIT1008/FIT2085 it is usually a word (4 bytes):
    - "load word" or `lw`
- **To execute a load instruction:**
  - The address to be loaded from goes into the MAR
  - The memory controller gets told to do a read
  - The data at that address goes into the MRR
  - The MRR is copied to the destination register (GPR) specified in the instruction

Getting data out of main memory (cont'd)

1 *Address we want to read from goes into Memory Address Register*

Control tells memory "Memory Read" 2



3 *Contents of memory at that address go into Memory Read Register*

Content of the Memory Read Register is copied into the appropriate GPR 4

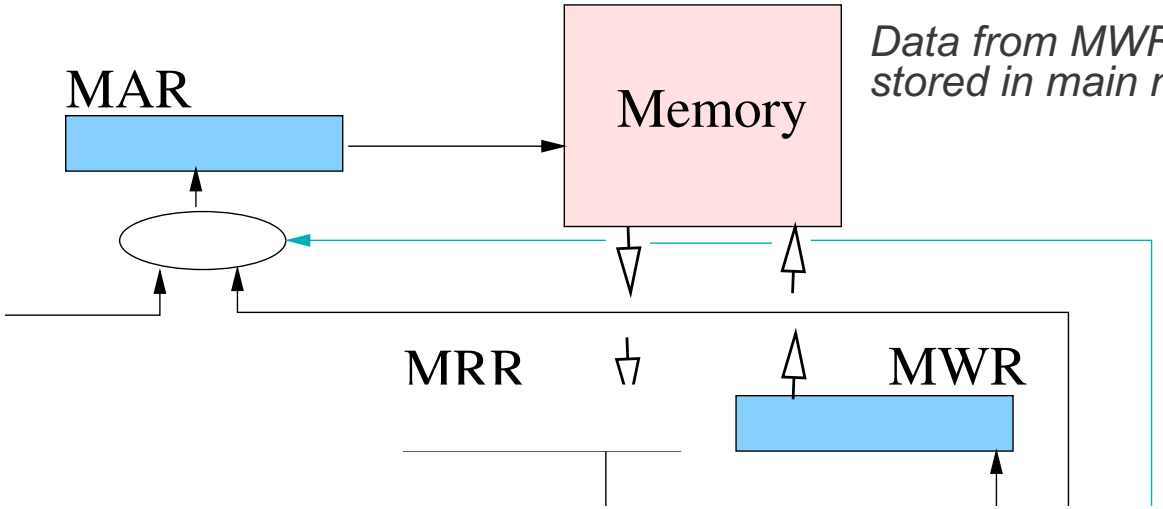
Putting data into main memory

- **We use a store instruction for this**
  - We can store 1, 2, or 4 bytes at a time
  - Again, in this unit, usually a word (4 bytes):
    - "store word" or `sw`
- **To execute a store instruction:**
  - The contents of the GPR specified in the instruction are copied to the MWR
  - The address to be stored to goes into the MAR
  - The memory controller gets told to do a write
  - The data in the MWR gets written to memory

Putting data into main memory (cont'd)

1

Address we want to write to goes into Memory Address Register



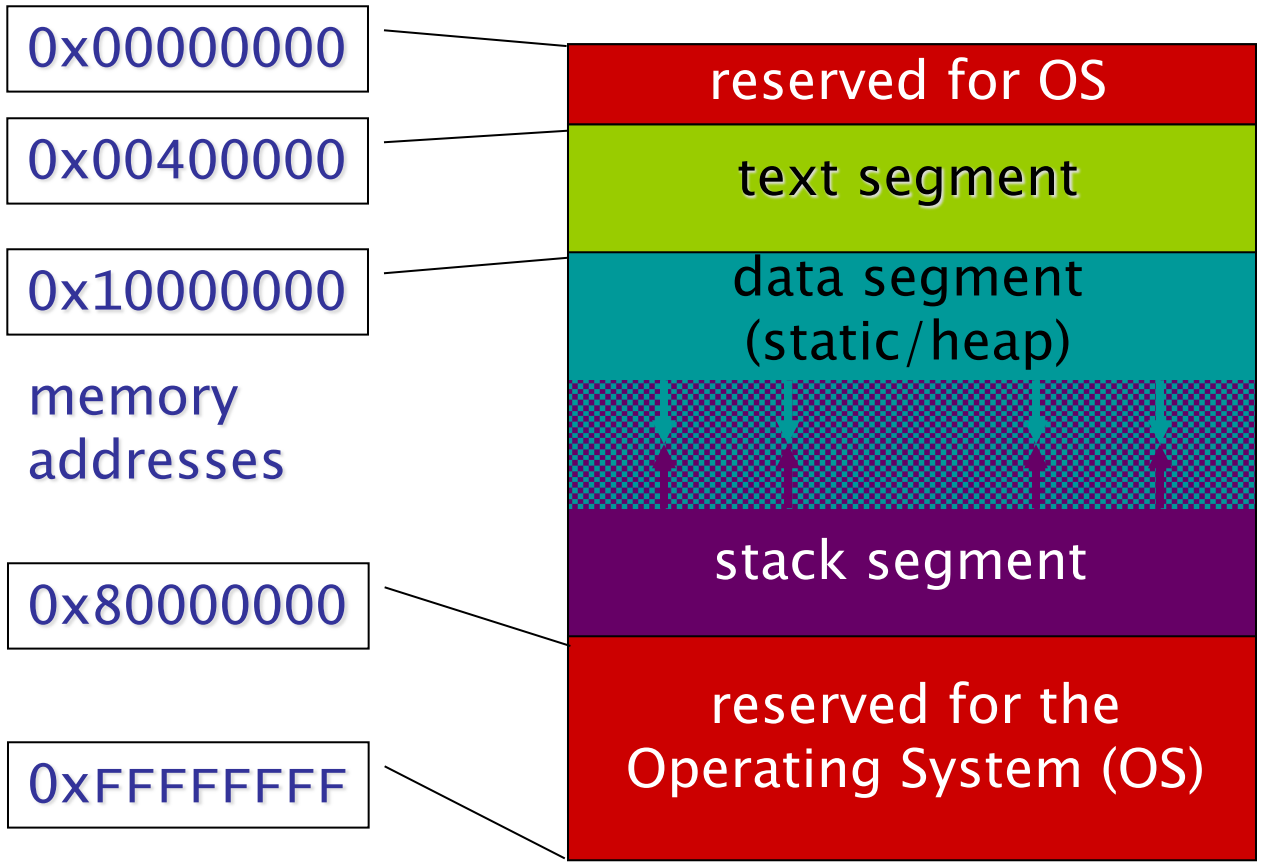
Data from MWR is stored in main memory 4

Control tells memory "Memory Write" 3

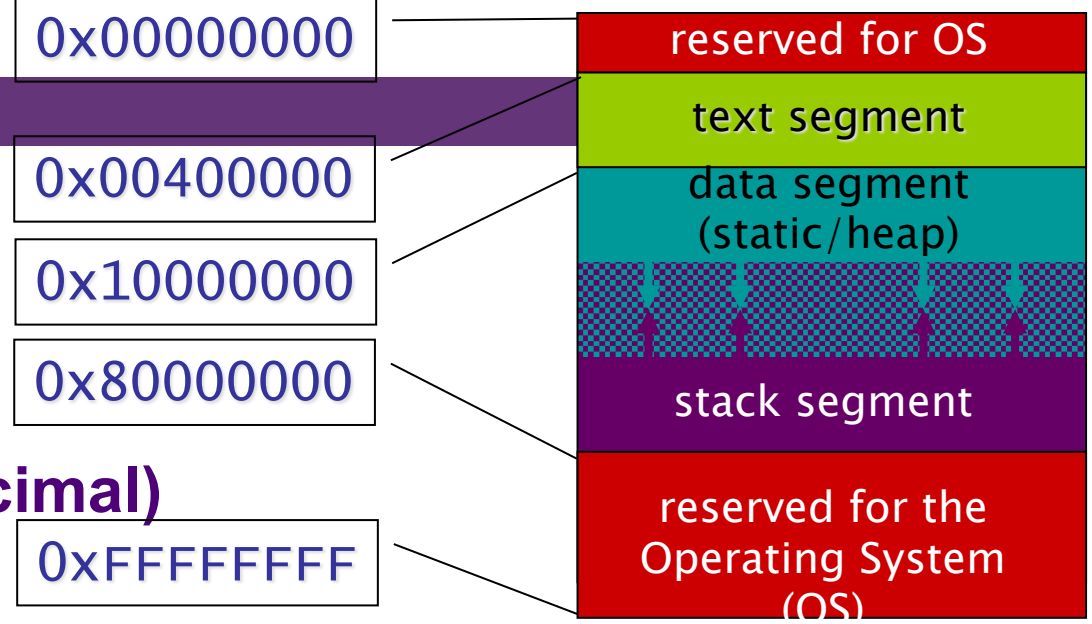
Data we want stored goes into Memory Write Register 2

MIPS memory architecture

- **Lets see what each segment is reserved for**

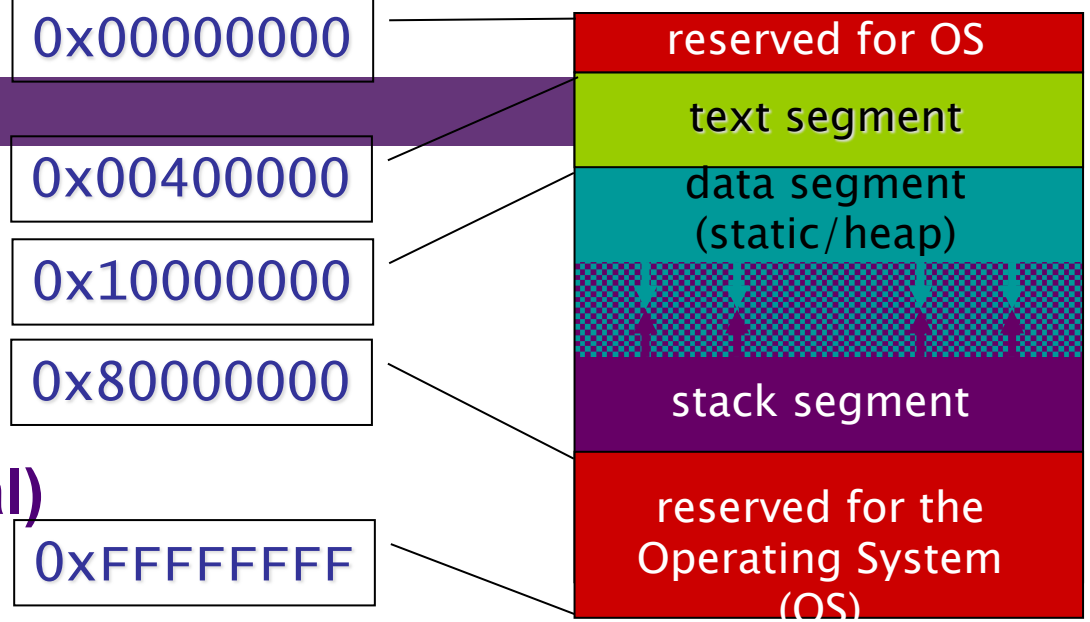


The Text Segment



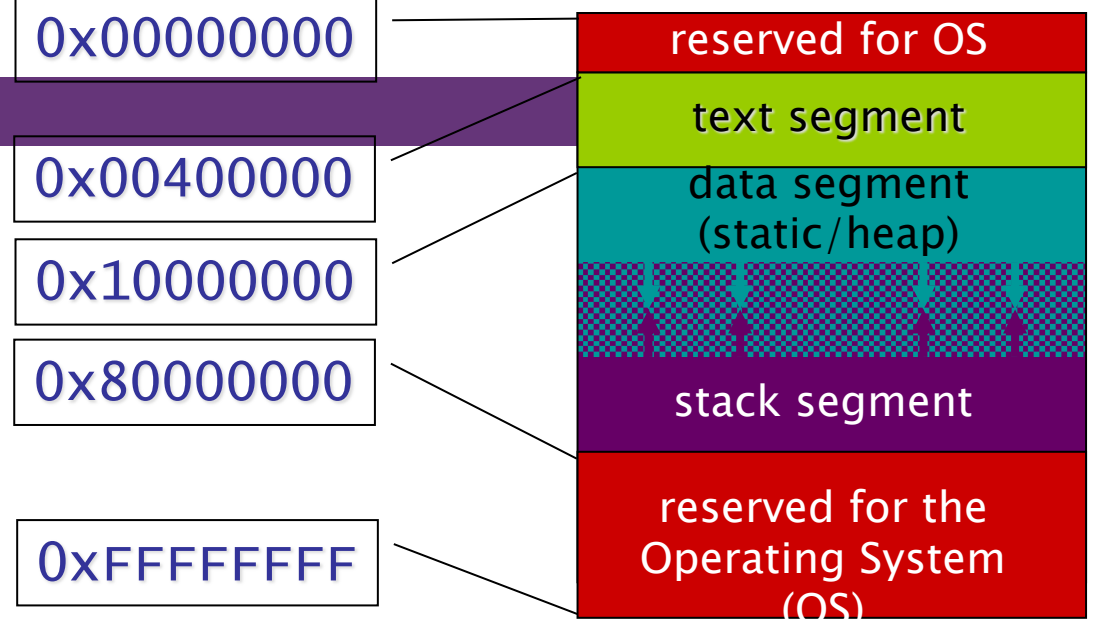
- **Starts at 0x00400000 (that's 4194304 in decimal)**
- **Executable code goes here**
  - In machine-code format (bit-patterns) remember?
- **PC register value is effectively a CPU "pointer" to the text segment**
- **How does code get here? For compiled programs, the OS puts it there when you tell it to run a program**
  - This process is called "loading"
- **Memory addresses smaller than 0x00400000 are reserved for the OS**

The Data Segment



- **Starts at 0x10000000 (268435456 in decimal)**
- **Conceptually divided into two parts**
- **First: for static (size known at compile-time) and globally available data:**
  - Values may change but size does not, and lifetime is the entire execution
- **This includes:**
  - Class data and static variables in Java
  - Literal strings in the source code, such as “Hello, World”
  - Global variables in languages that have them
- **Second: dynamic data (usually referred to as the Heap)**
  - The Data Segment often means the static part, so be careful

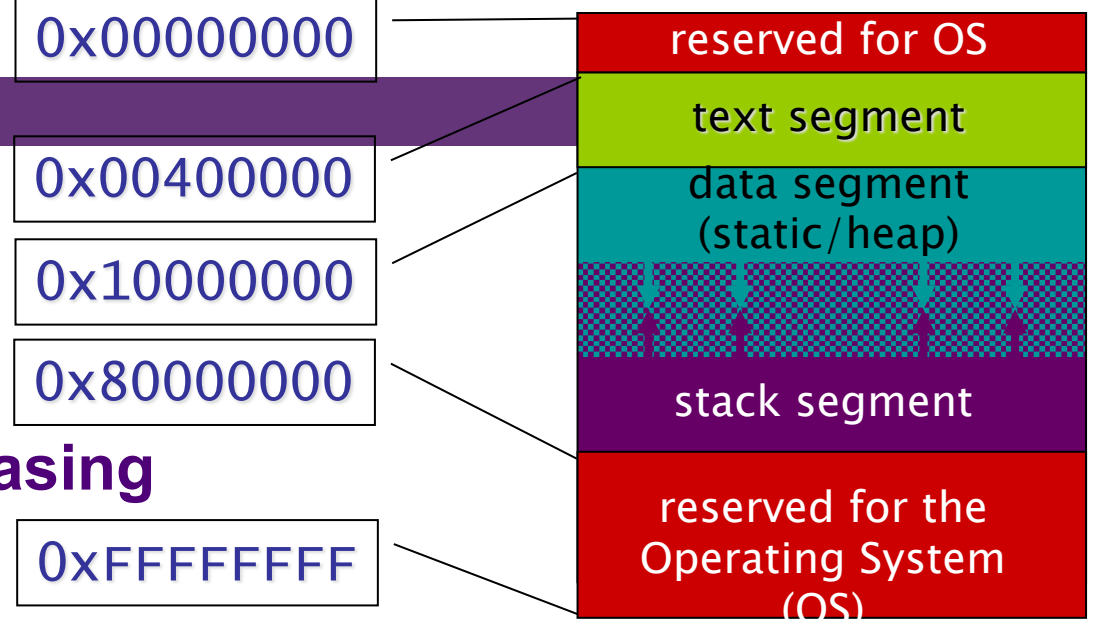
The Heap Segment



- **Technically part of data segment**
  - Located at the end, after all static data
- **All dynamically allocated memory (i.e., allocated during execution) is here**
  - All Python objects and their instance variables go here
- **Grows "downwards" as more memory is allocated (from where static ends)**
  - For example, after creating an object
- **Shrinks when memory is deallocated**
  - Either by the garbage collector or the programmer
- **Empty at the start of program execution**
- **Note: "heap" means "pile", not as in max/min heap data type**

The Stack Segment

- **Starts at memory address 0x7FFFFFFF**
- **Grows "upwards" in the direction of decreasing memory addresses**
  - That is, towards the heap
- **Contains the system stack**
- **It is used for the temporary storage of:**
  - Function information:
    - Local variables
    - Value of parameter
    - Return address
  - Saved register values (not studied in this unit)
- **Note: this really is a stack like the data types we will see later**



Summary (of the entire lesson)

- **We have seen how learning MIPS is useful and learned about:**
  - von Neuman architecture
  - Machine code and Machine language
  - The concept of “word" in computer architecture
  - RISC vs CISC
- **We have seen the basics of MIPS R2000 architecture**
- **We are familiar with the CPU’s registers**
  - GPRs, PC, HI, LO, IR, MAR, etc
- **We know how programs are executed in this architecture**
  - The fetch-decode-execute cycle
- **We know how MIPS memory is structured and accessed**
  - Memory segments and load/store instructions