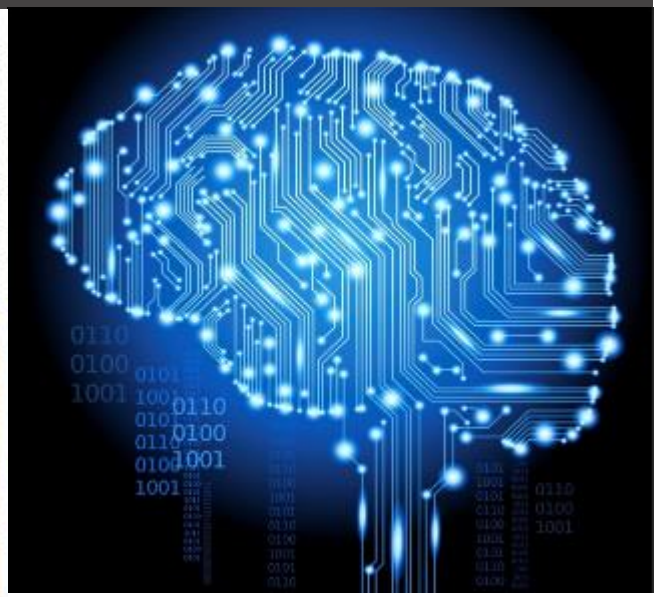
MONASH University

Information Technology

FIT1008/FIT2085 Lecture 2

MIPS Architecture

Prepared by: D. Albrecht, J. Garcia and M. Garcia de la Banda



Were are we at

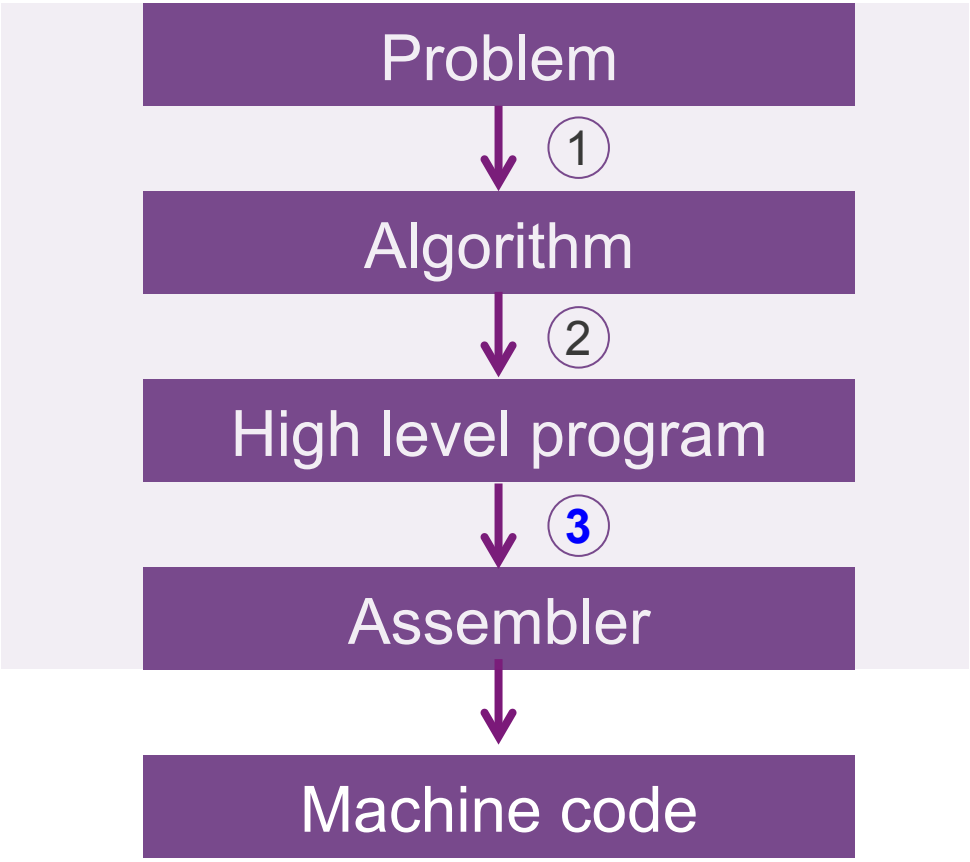
- **We know everything we need to know about FIT2085:**
  - Organisation, structure, assessment, hurdles, communication, etc
- **We know what the coding expectations for pracs are**
  - Layout, variable names, comments, logic, reuse, testing
- **We have started the transition to Python**
  - Fine for simple programs (no classes yet)
  - Unit testing using assertions

Objectives for this lecture

- **To get an idea of how your code can be run on a computer**
- **To understand the basics of MIPS R2000 architecture**
  - Memory organization
  - CPU registers
- **To be able to reason about some of the decisions made in this architecture**
- **To understand how programs are executed in this architecture**
  - The fetch-decode-execute cycle
  - Accessing main memory

We said this unit...

- Was 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
- The first 3 weeks of FIT2085 are devoted to the third step



In pictorial terms…

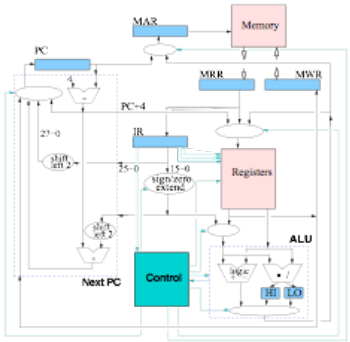
You

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

Human-readable code

????????

Machine code



CPU

Compiling high level code (simplified)

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

High -level code

Compiled

```
1   0 LOAD_CONST          0 (1)
    3 STORE_NAME          0 (f)

2   6 LOAD_NAME           1 (int)
    9 LOAD_NAME           1 (input)
                ...
```

bytecode

Machine independent

Compiled

Compiled

Machine code

Executed by Virtual Machine

Executed

Machine dependent

Compiled

Output

```
int f = 1, n = 0;
printf("Enter int: ");, scanf("%d", &n);
while (n > 0) {
    f = f*n;
    n = n-1;
}
printF("%d\n", f);
```

C code

Why do you need to learn about this?

- **All high level code ends up being translated into some kind of assembler code**
- **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
- **Hennessy recently retired as President of Stanford University**

Why MIPS?

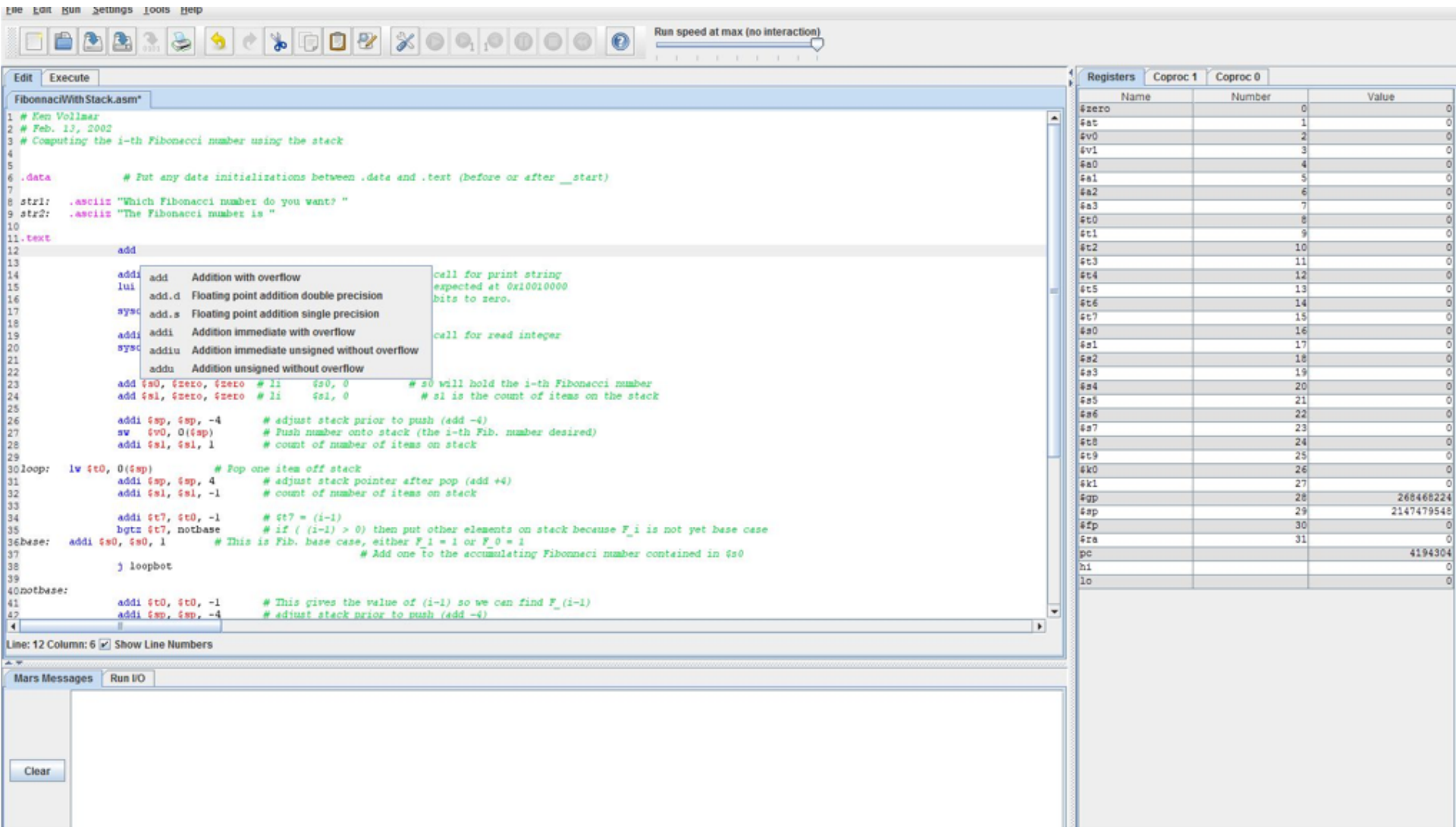
- **A real processor (not a toy one)**
  - MIPS32 & MIPS64 still in production
- **Ancestor (because of RISC) of many popular computers**
  - Apple/IBM/Motorola PowerPC (Macintosh), Digital Alpha (Alpha), ARM (3Com Palm and most embedded)
- **Knowledge of MIPS can be easily carried over to these other architectures**
- **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

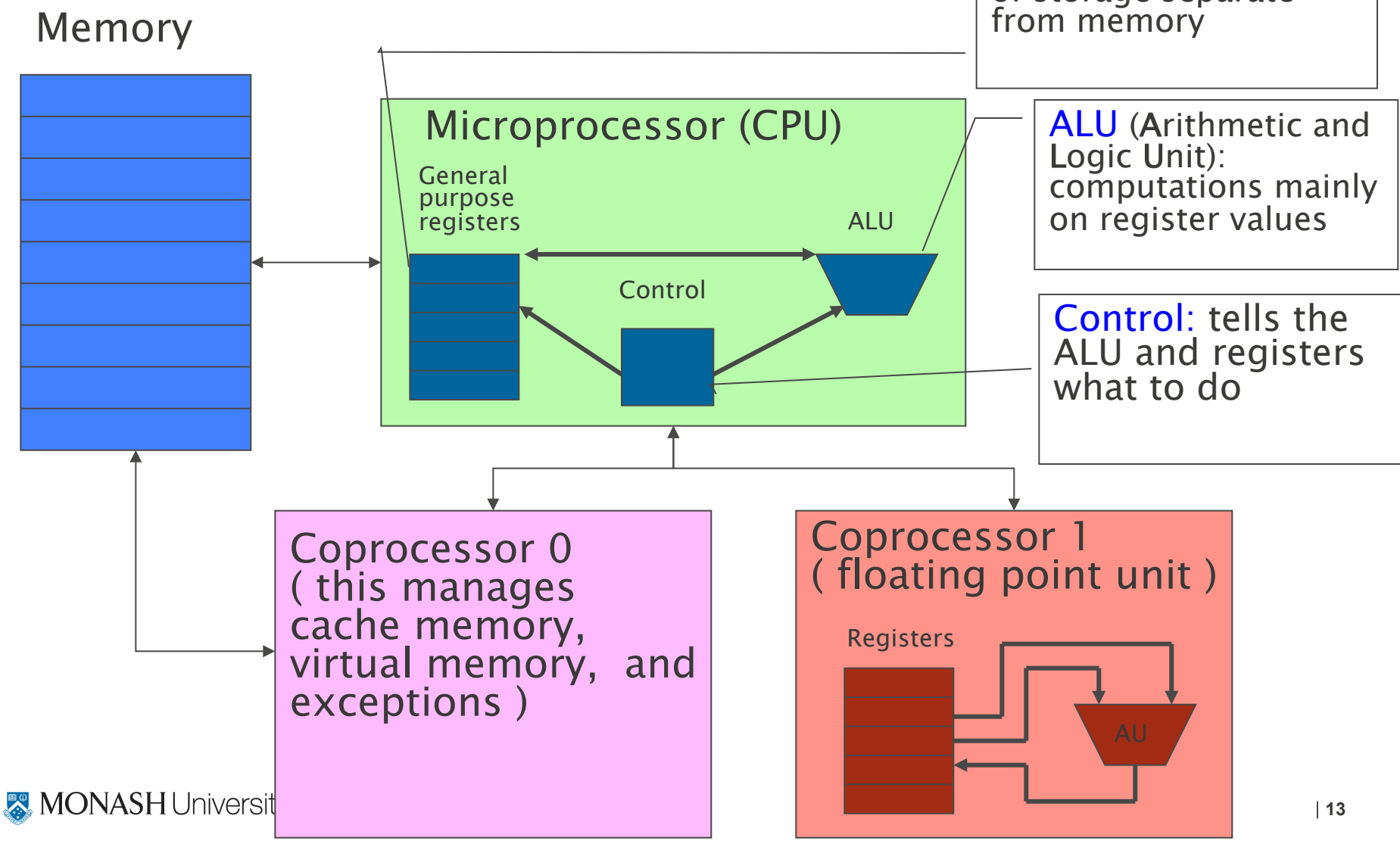
- **MIPS was first computer to use the term RISC**
  - **R**educed **I**nstruction **S**et **C**omputer
  - All instructions are
    - Same length (4 bytes, that is, 32 bits – a "word" for us)
    - Of similar complexity (simple)
    - (Mostly) able to run in same time (1 clock cycle)
    - Easily decoded and executed by computer hardware
  - Advantages: easier to build, cheaper, consumes less power
- **Intel x86 is considered CISC**
  - **C**omplex **I**nstruction **S**et **C**omputer
  - Instructions vary in length, complexity and execution time
  - Decoding and running instructions requires hardware-embedded program (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
    - 450,000 transistors in a MIPS R2000 (1986)
    - 780,000,000 transistors in an Intel i7 (2009)
- **MIPS is representative of modern computer architecture**
  - More useful to learn
- **MIPS has readily available simulators:**
  - http://courses.missouristate.edu/KenVollmar/MARS/
  - Makes it easier to write/test MIPS programs


File Edit Run Settings Tools Help
Run speed at max (no interaction)
Edit Execute
FibonnaciWithStack.asm*
1 # Ken Vollmar
2 # Feb. 13, 2002
3 # Computing the i-th Fibonacci number using the stack
4
5
6 .data # Put any data initializations between .data and .text (before or after __start)
7
8 str1: .asciiz "Which Fibonacci number do you want? "
9 str2: .asciiz "The Fibonacci number is "
10
11.text
12 add
13
14 addi
15 lui
16
17 sysc
18
19 addi
20 sysc
21
22
23 add $s0, $zero, $zero # li $s0, 0 # s0 will hold the i-th Fibonacci number
24 add $s1, $zero, $zero # li $s1, 0 # s1 is the count of items on the stack
25
26 addi $sp, $sp, -4 # adjust stack prior to push (add -4)
27 sw $v0, 0($sp) # Push number onto stack (the i-th Fib. number desired)
28 addi $s1, $s1, 1 # count of number of items on stack
29
30loop: lw $t0, 0($sp) # Pop one item off stack
31 addi $sp, $sp, 4 # adjust stack pointer after pop (add +4)
32 addi $s1, $s1, -1 # count of number of items on stack
33
34 addi $t7, $t0, -1 # $t7 = (i-1)
35 bgtz $t7, notbase # if ((i-1) > 0) then put other elements on stack because F_i is not yet base case
36base: addi $s0, $s0, 1 # This is Fib. base case, either F_1 = 1 or F_0 = 1
37 # Add one to the accumulating Fibonnaci number contained in $s0
38 j loopbot
39
40notbase:
41 addi $t0, $t0, -1 # This gives the value of (i-1) so we can find F_(i-1)
42 addi $sp, $sp, -4 # adjust stack prior to push (add -4)
call for print string
expected at 0x10010000
bits to zero.
call for read integer
add Addition with overflow
add.d Floating point addition double precision
add.s Floating point addition single precision
addi Addition immediate with overflow
addiu Addition immediate unsigned without overflow
addu Addition unsigned without overflow
Line: 12 Column: 6 Show Line Numbers
Mars Messages Run I/O
Clear
Registers Coproc 1 Coproc 0
Name Number Value
$zero 0 0
$at 1 0
$v0 2 0
$v1 3 0
$a0 4 0
$a1 5 0
$a2 6 0
$a3 7 0
$t0 8 0
$t1 9 0
$t2 10 0
$t3 11 0
$t4 12 0
$t5 13 0
$t6 14 0
$t7 15 0
$s0 16 0
$s1 17 0
$s2 18 0
$s3 19 0
$s4 20 0
$s5 21 0
$s6 22 0
$s7 23 0
$t8 24 0
$t9 25 0
$k0 26 0
$k1 27 0
$gp 28 268468224
$sp 29 2147479548
$fp 30 0
$ra 31 0
pc 4194304
hi 0
lo 0

MIPS Architecture (simplified)



General purpose registers: fast words of storage separate from memory

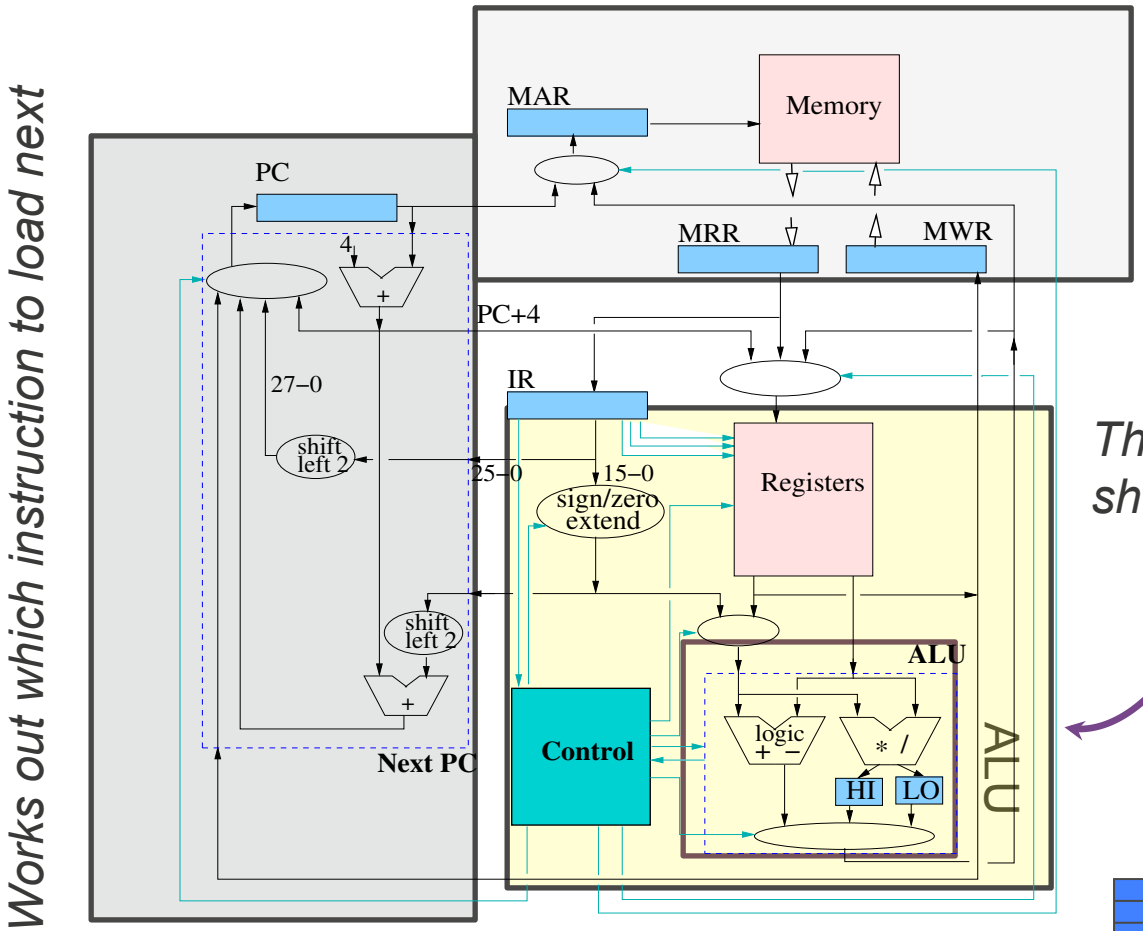
ALU (Arithmetic and Logic Unit): computations mainly on register values

Control: tells the ALU and registers what to do

Main components: basics

- **32 General-purpose registers**
  - Fast but expensive memory
  - Physically located on the CPU chip
  - Each 32 bits in size
- **Arithmetic Logic Unit (ALU)**
  - Performs computations on register values (not main memory):
    - Register-register (or load-store) 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 (again)

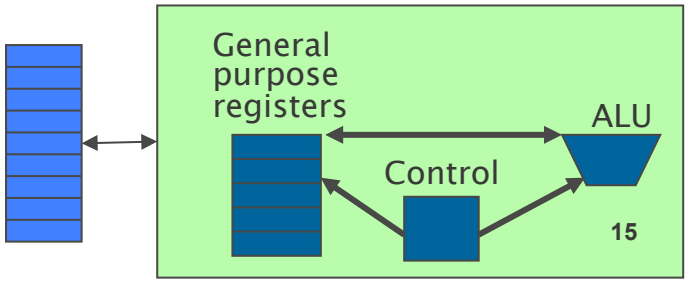


Works out which instruction to load next

Talks to main memory

The image in slide 12 showed these bits

Warning: Not oversimplified at all!



von Neumann architectures

- **This is an example of what's called a "von Neumann architecture"**
  - Separate modules for
    - Processing and
    - Main memory
  - Programs are stored in main memory
  - Each instruction must be loaded into the processor before execution

- **Named after John von Neumann, who described such an architecture**
- **Notation: a word is a chunk of bits (8, 16, 32 or 64) used as basic unit of data in a computer (to store, operate on, move, etc)**
  - Depends on the computer
  - We will use 32 bits, that is, 4 bytes

General Purpose Registers (GPRs)

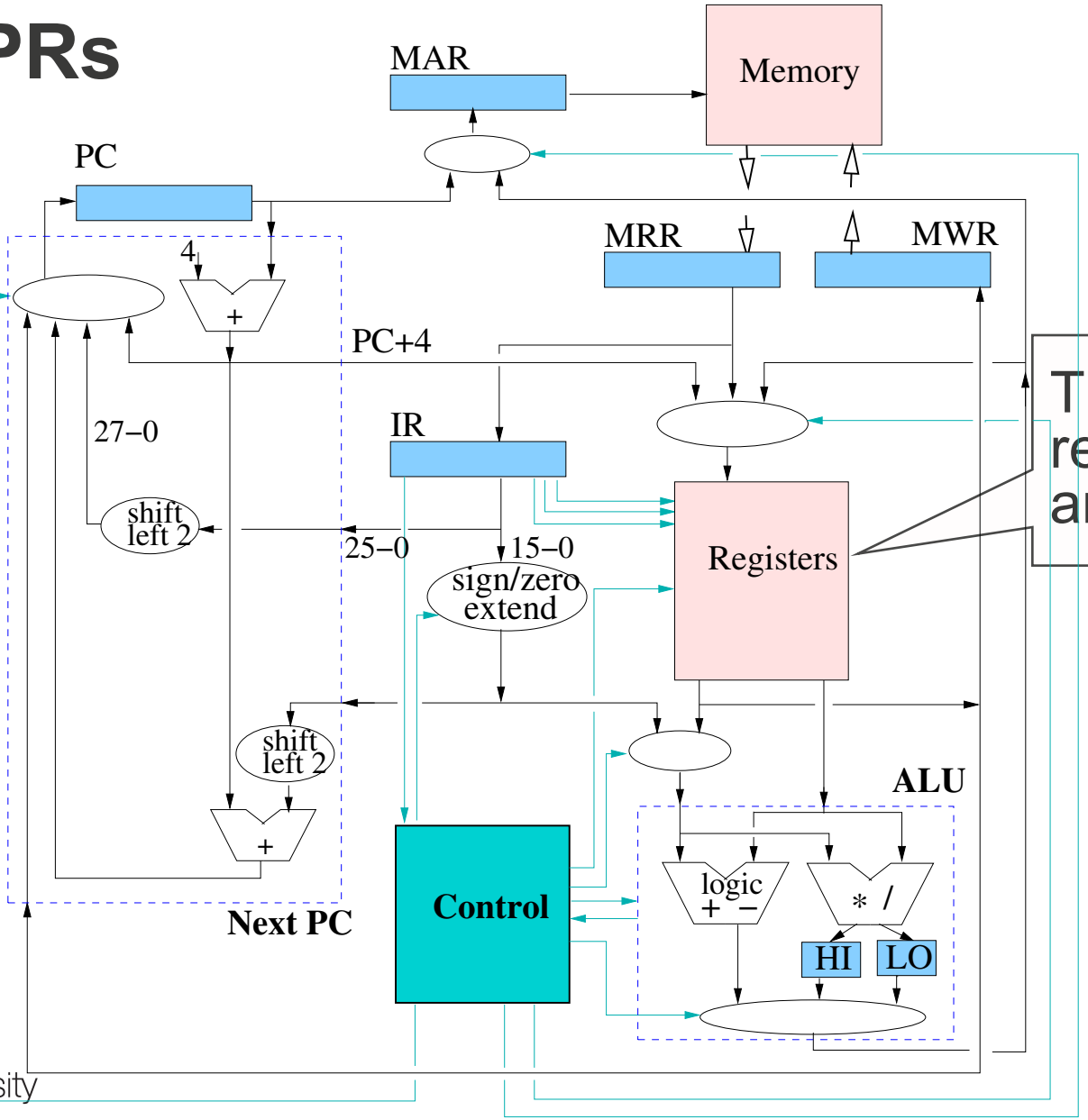
- **Prefixed with `$` in assembly language**
  - Numbered `$0` to `$31`
  - Also given names based on usage conventions, e.g.:
    - `$0` ⇔ `$zero` (special case read-only register, always set to 0)
    - `$4` ⇔ `$a0`
    - `$29` ⇔ `$sp`
- **Unlike variables, you can't name them yourself**
  - They're 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 in FIT2085	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

MIPS GPRs


MAR
Memory
PC
4
+
MRR
MWR
PC+4
27–0
IR
shift left 2
25–0
15–0
sign/zero extend
Registers
The 32 registers are here
shift left 2
+
Next PC
Control
ALU
logic + −
* /
HI
LO

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
```

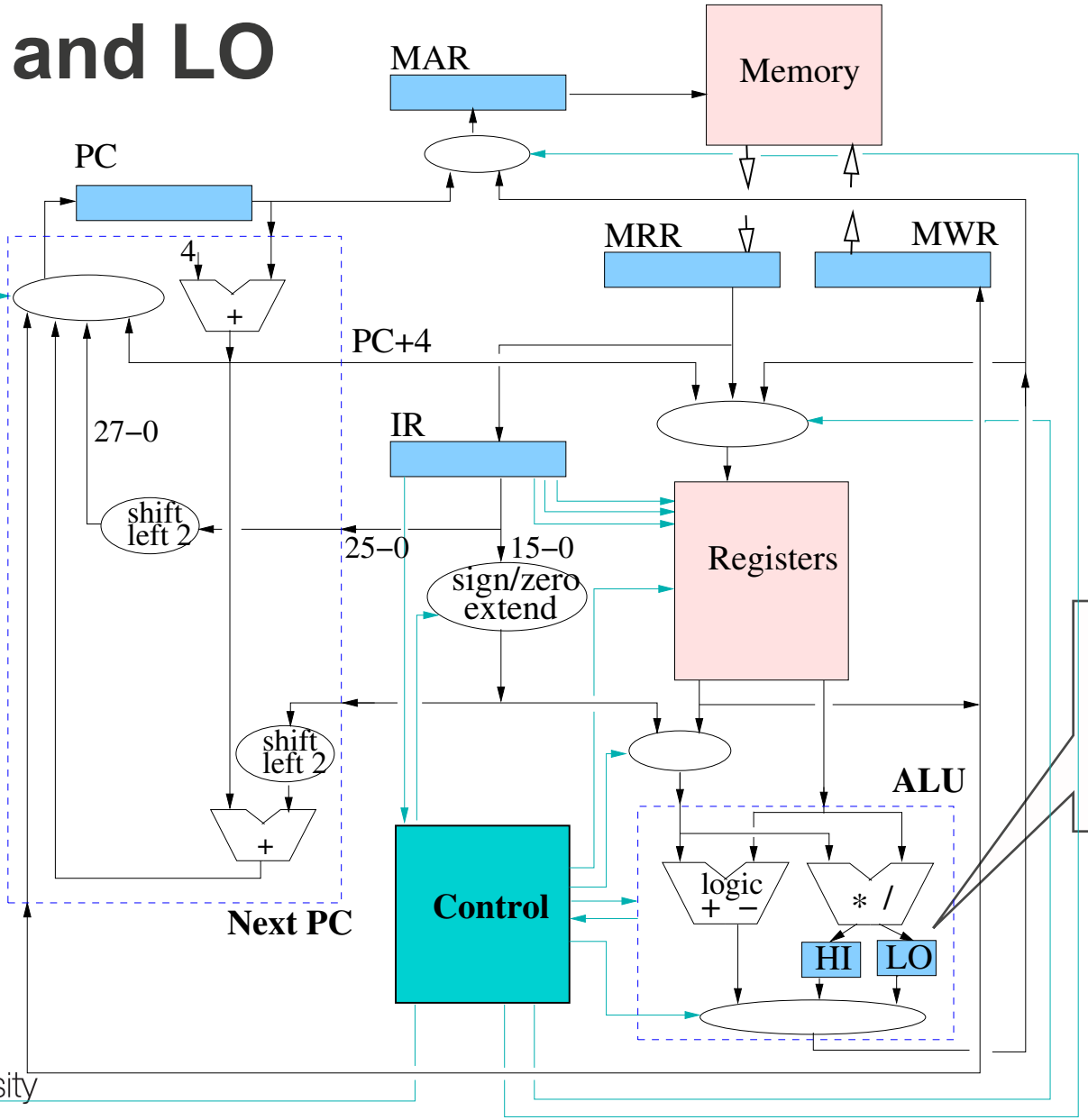
- We'll see more examples next lecture
- Floating-point operations are beyond the scope of this unit

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 might be used to get the (integer) result or to get the remainder**
- **After an integer division:**
  - LO contains the (integer) result
  - HI contains the remainder
- **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



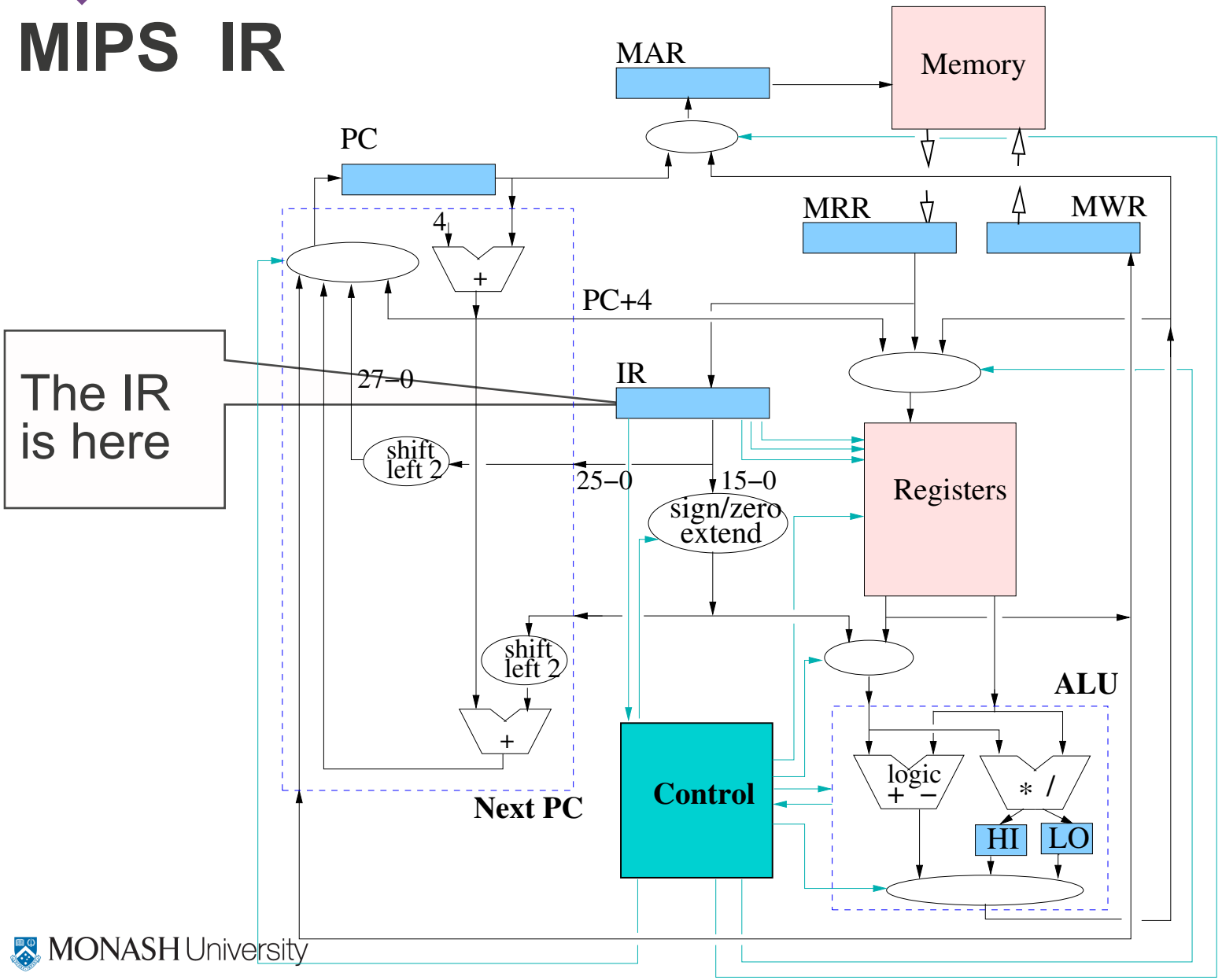
Special Purpose Registers (IR)

- **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`)
- **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
    - For example: Are they GPRs? Are they immediate values?
- **The opcode tells the control circuitry which set of microinstructions needs to be followed**
- **The IR is not visible to the programmer**

MIPS IR

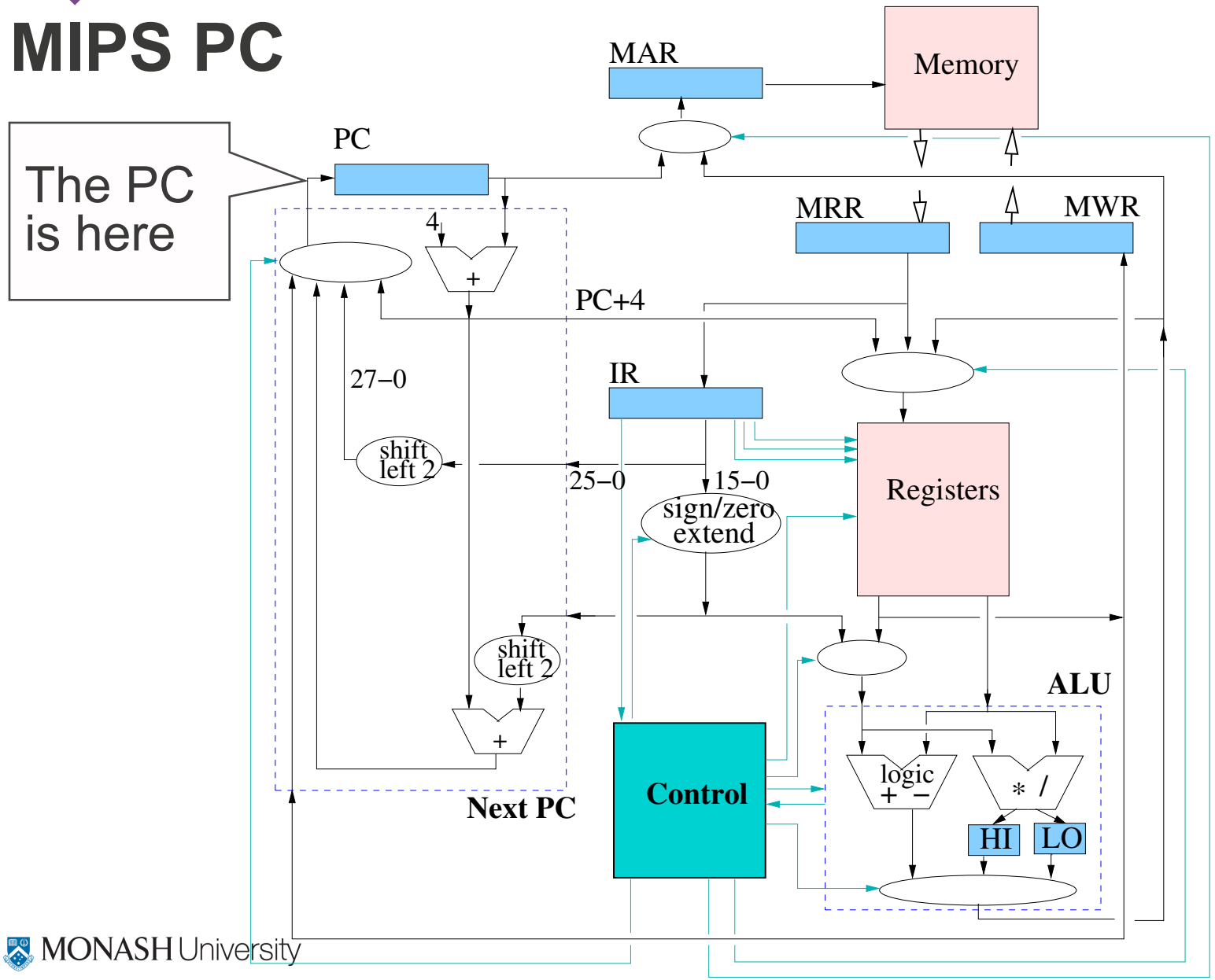

MAR
Memory
PC
4
+
MRR
MWR
PC+4
The IR is here
27–0
IR
shift left 2
25–0
15–0
sign/zero extend
Registers
shift left 2
+
Next PC
Control
ALU
logic + –
* /
HI
LO

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 (PC=PC+4) by most machine instructions to point to next instruction**
  - Remember: 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

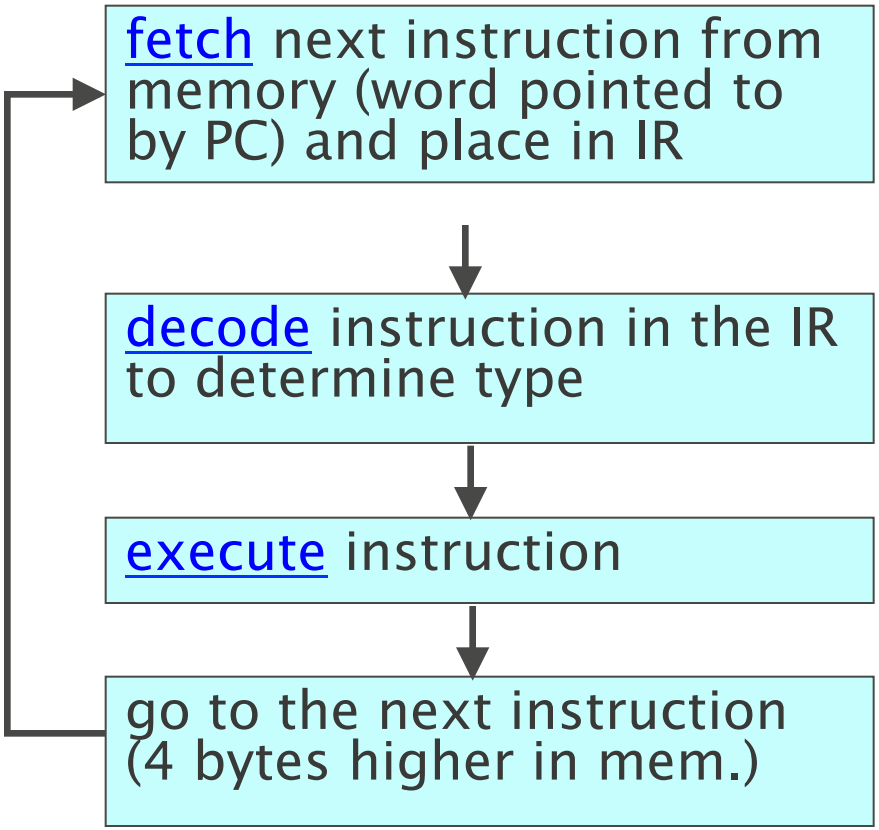
MIPS PC

The PC is here

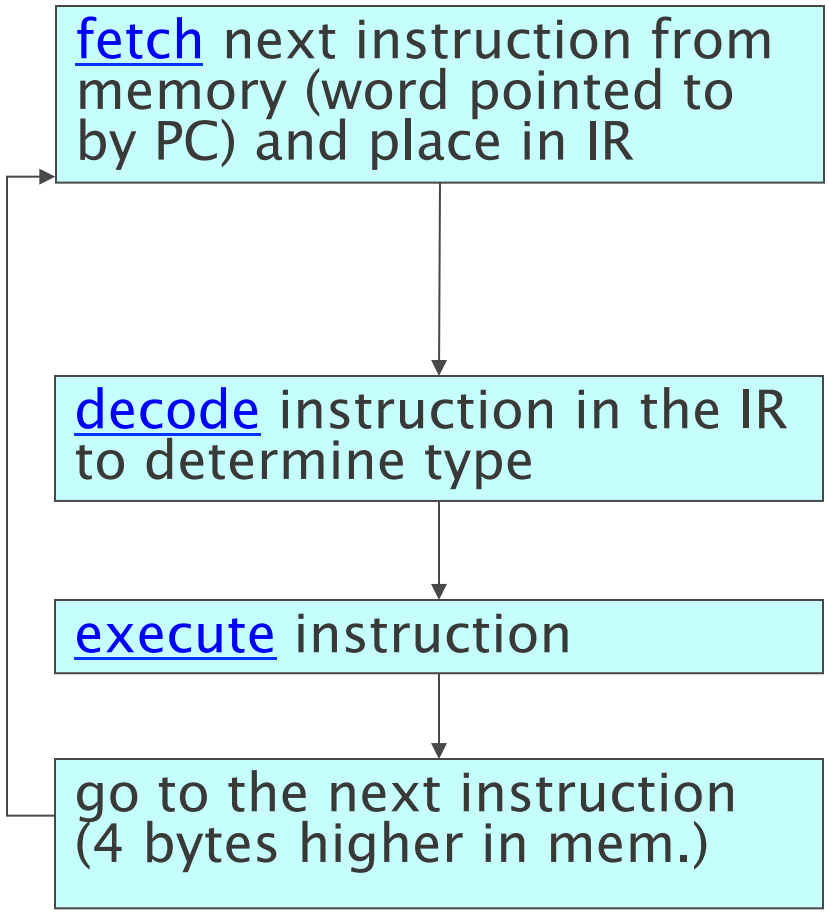


MIPS Instruction Execution

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



MIPS Instruction Execution: Example



hexadecimal

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

0010 0000 1010 1001 1111 1111 1111 1101

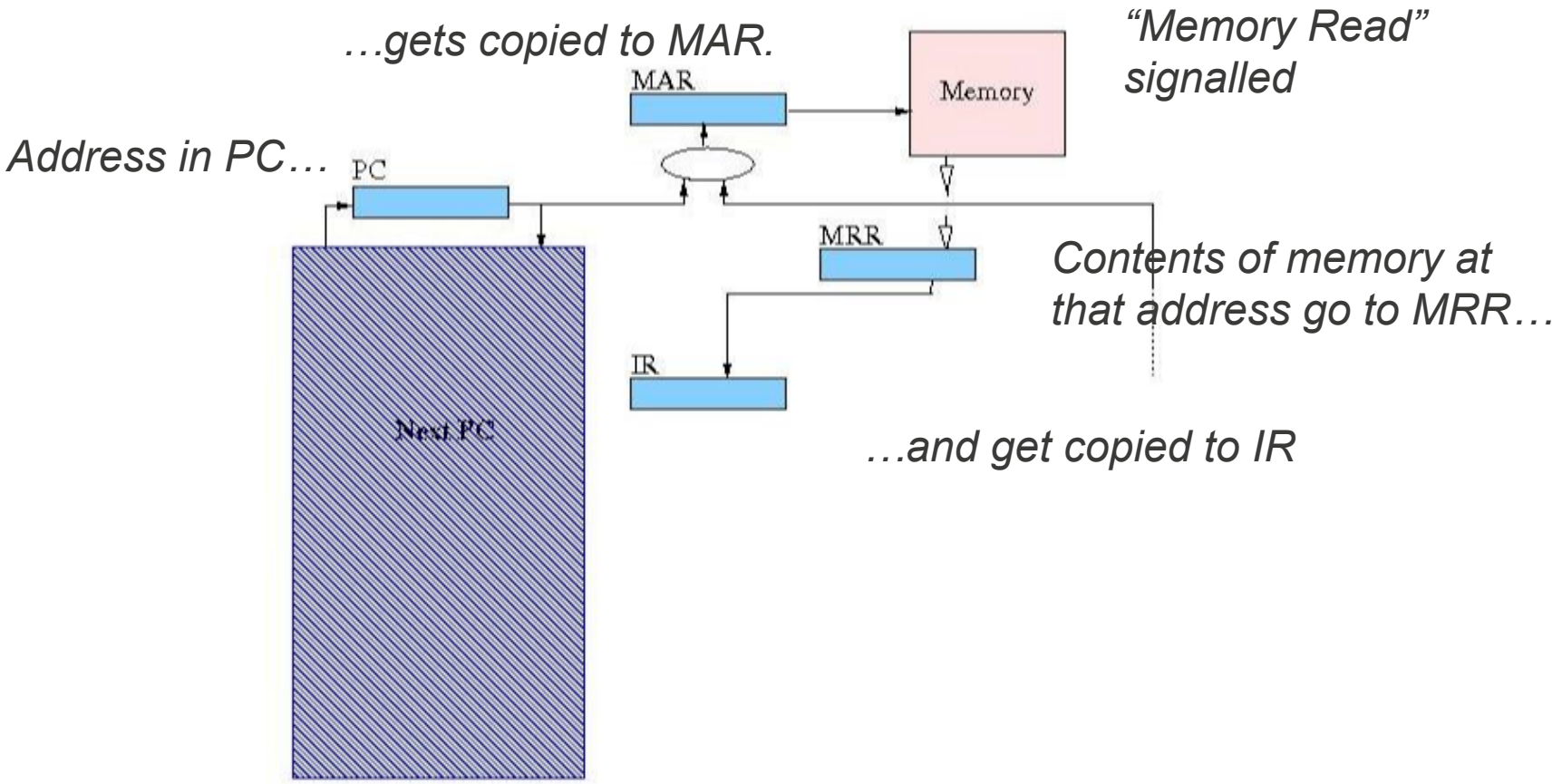
001000 00101 01001 1111111111111101

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

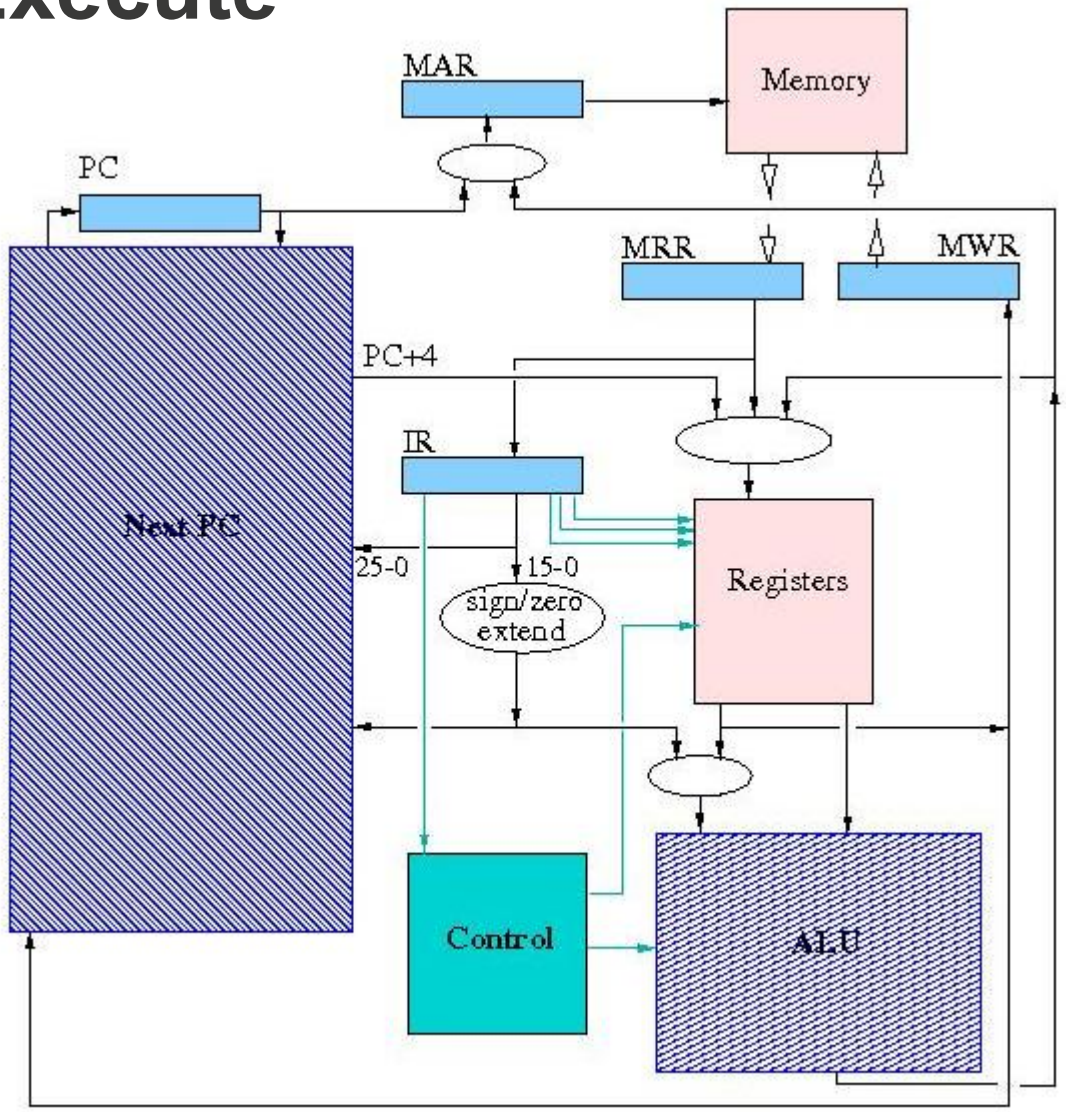
Send reg $5 and -3 to ALU, add them, result to reg $9

PC = PC + 4

The fetch Hardware



Decode-Execute

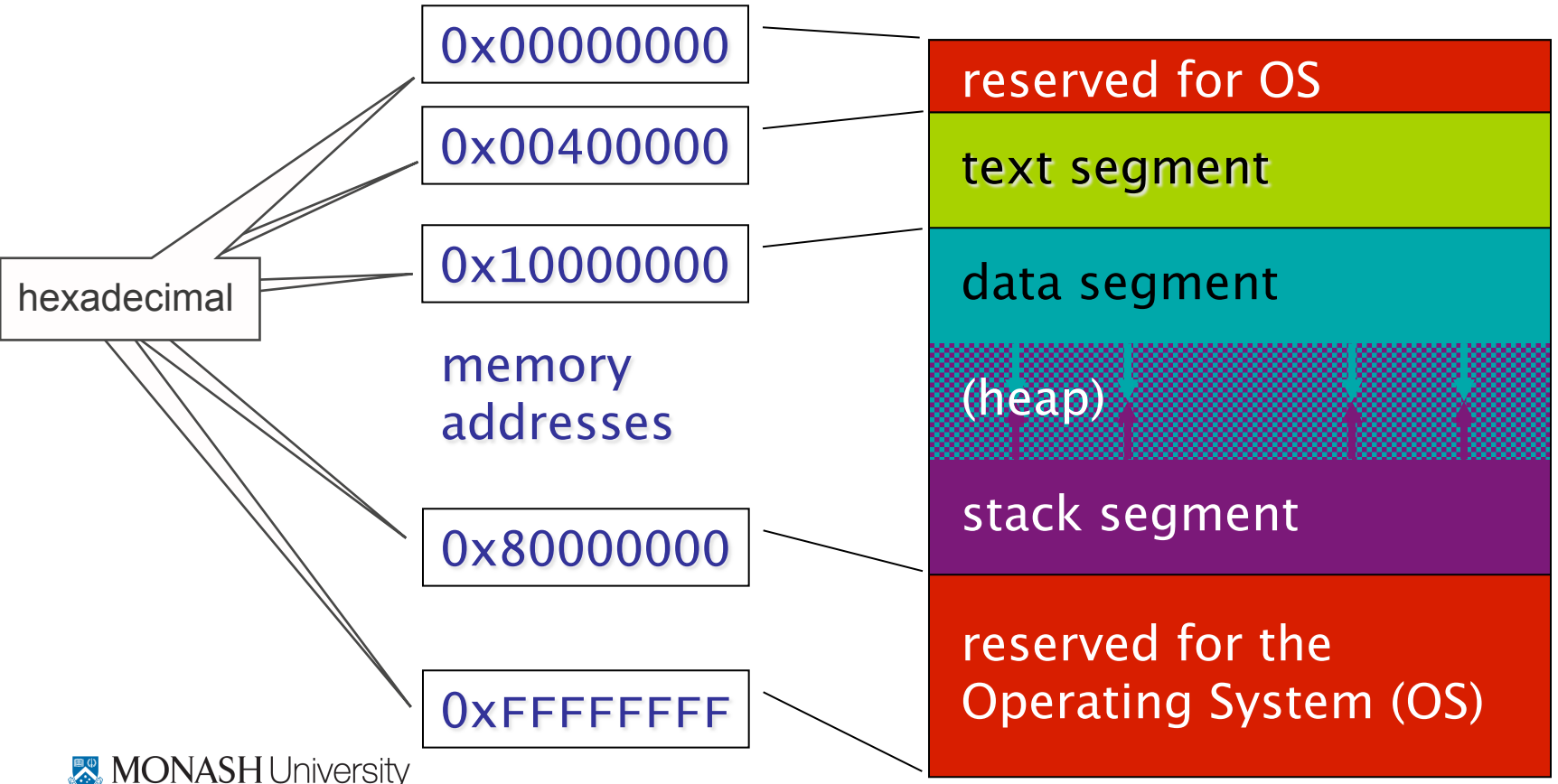


Accessing Main memory

- **Recall: MIPS is a load-store architecture**
  - That is, computations take place 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 it 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 memory A to memory B)
- 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?**
  - Smallest possible is 0x00000000
  - Largest possible is 0xFFFFFFFF, which is $2^{32} - 1$
- **Each address refers to one byte of memory (so 8 bits)**
  - Total potential address space: 4 Gb

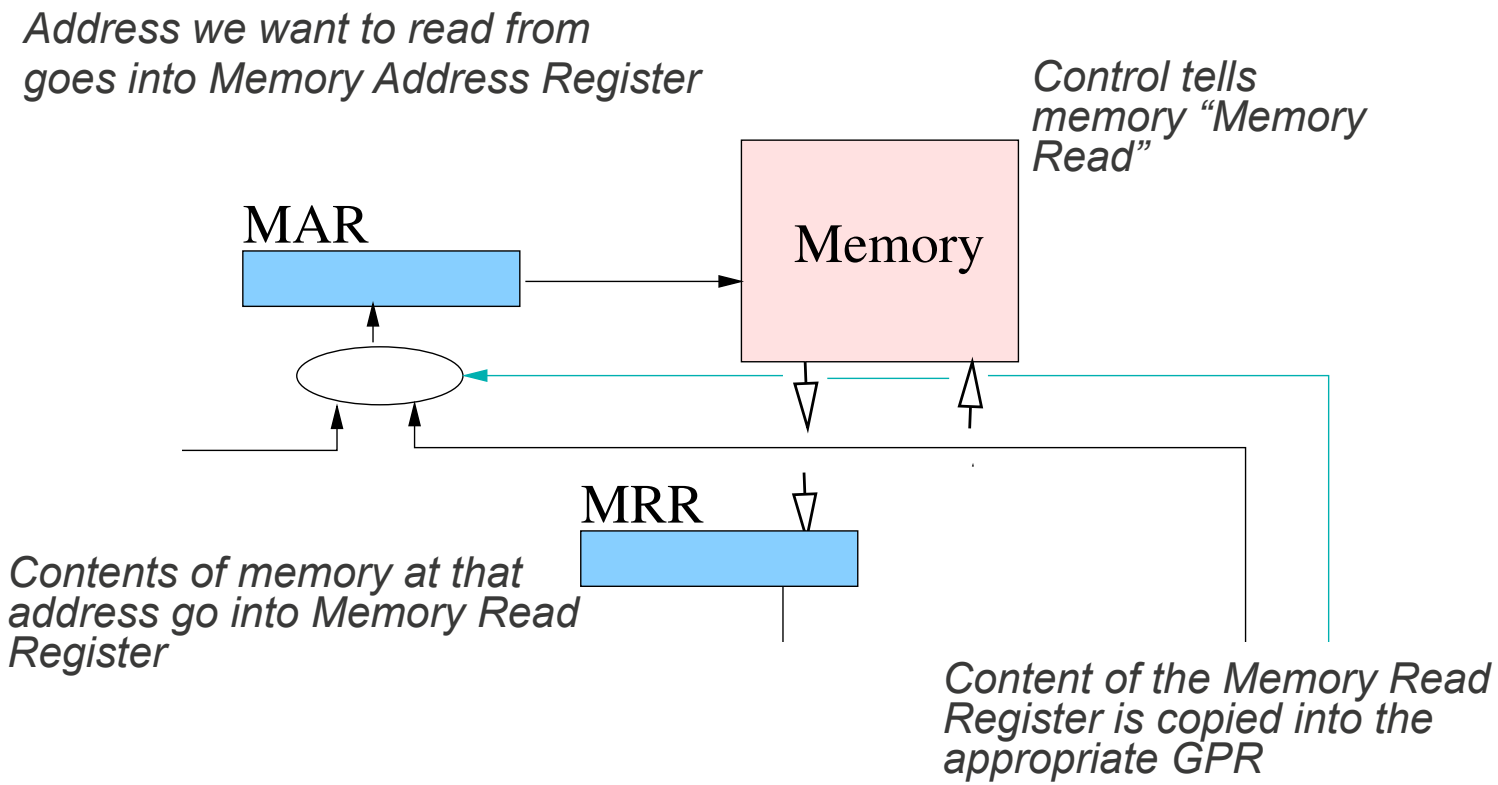
BTW: Looking for meaning in memory

- **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
  - 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 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 this unit 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

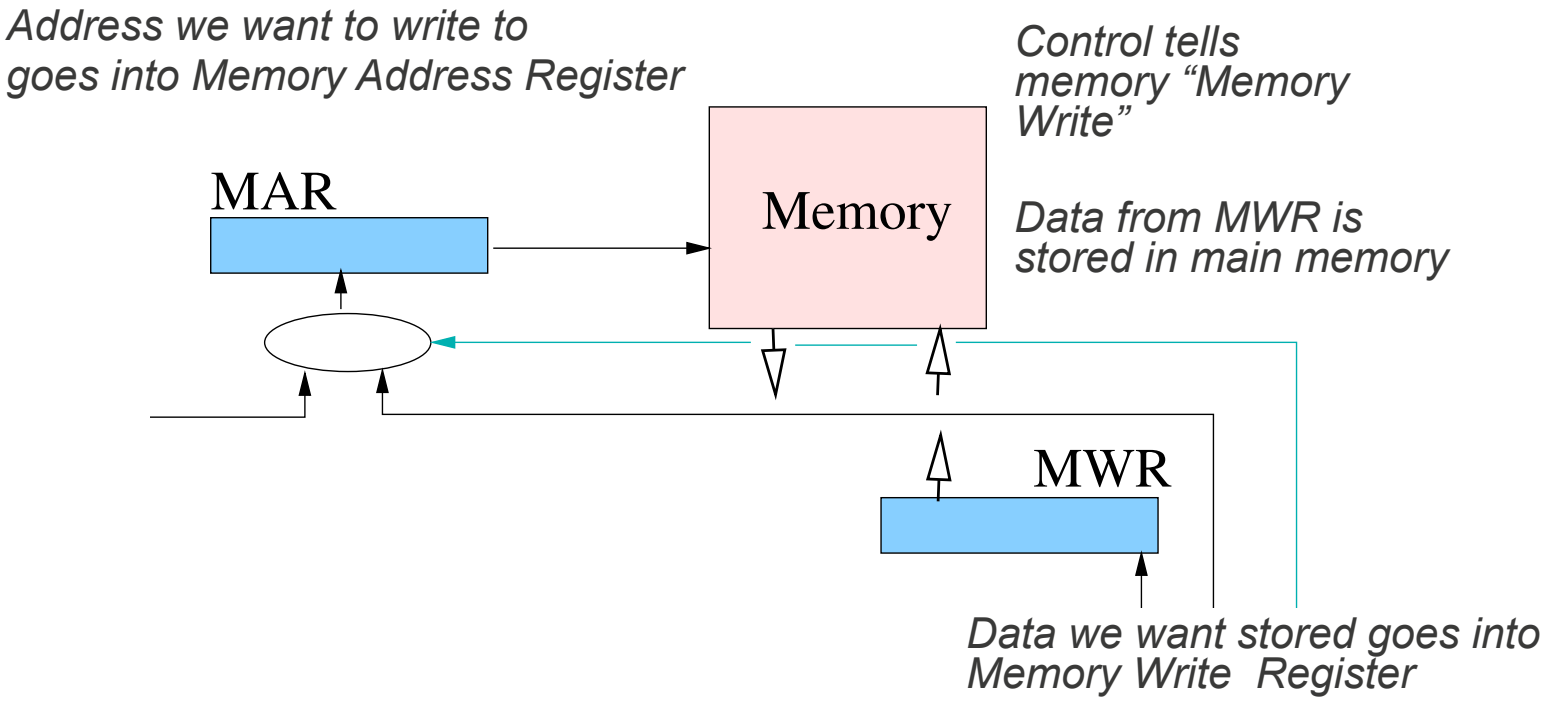
Getting data out of main memory (cont'd)



Putting data into main memory

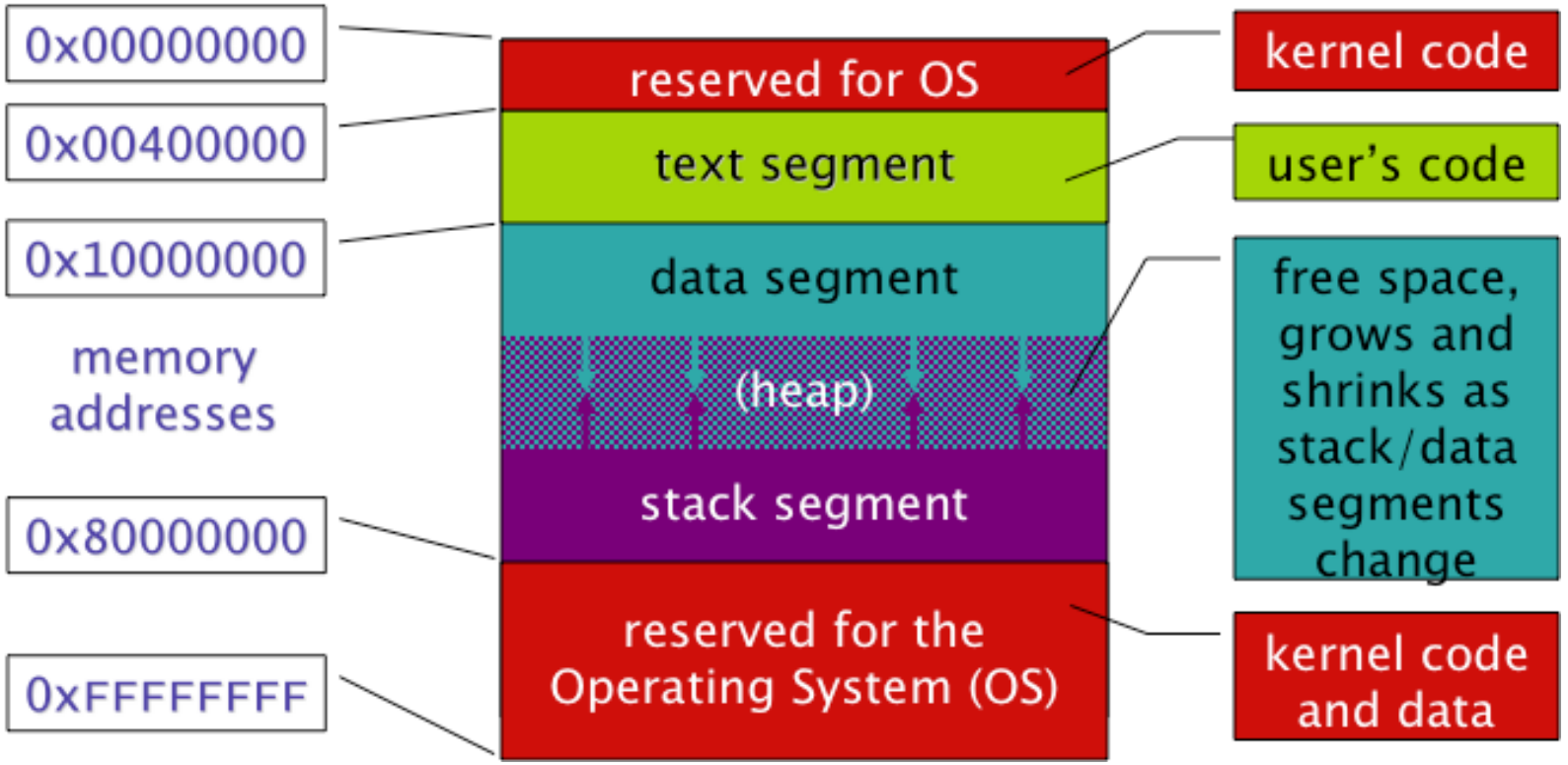
- **We use a store instruction for this**
  - We can store 1, 2, or 4 bytes at a time
  - Again, in this unit, usually a word (4 bytes): “store word” or **`sw`**
- **To execute a store instruction:**
  - The contents of the GPR specified in the instruction are copied to the MWR
  - The address to be 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)

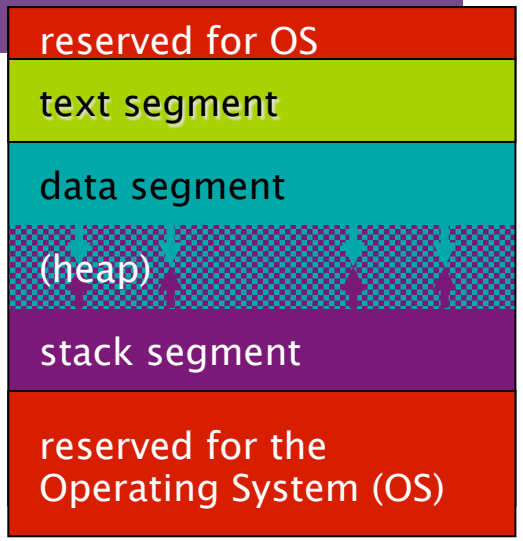


MIPS memory architecture

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

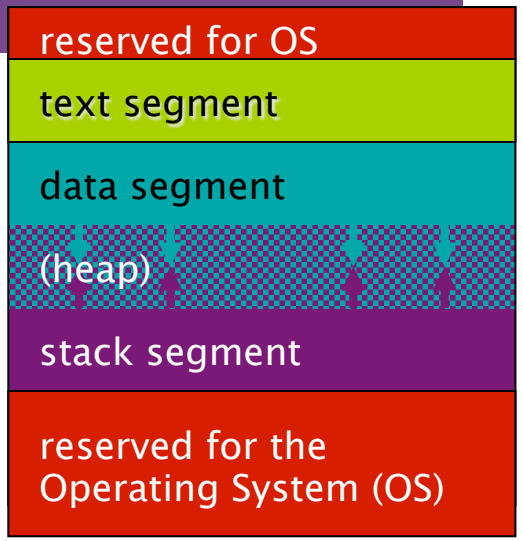


The Text Segment



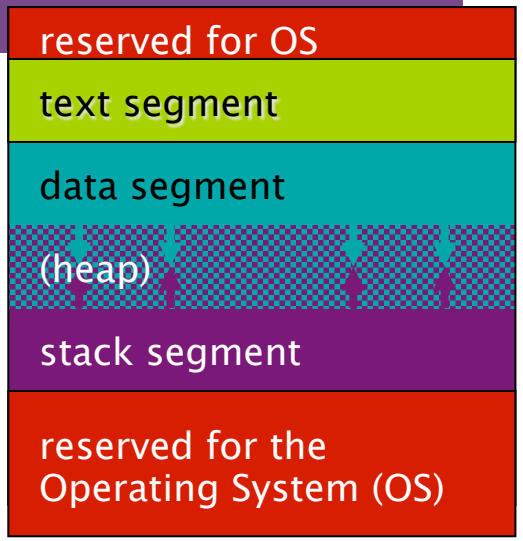
- **Starts at 0x00400000 (that's 4194304 in decimal)**
- **Executable code goes here**
  - In machine-readable format (remember?)
- **PC register value is effectively a CPU "reference" into the text segment**
- **How does code get here? For compiled programs, the OS puts it there when you tell it to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

The Data Segment



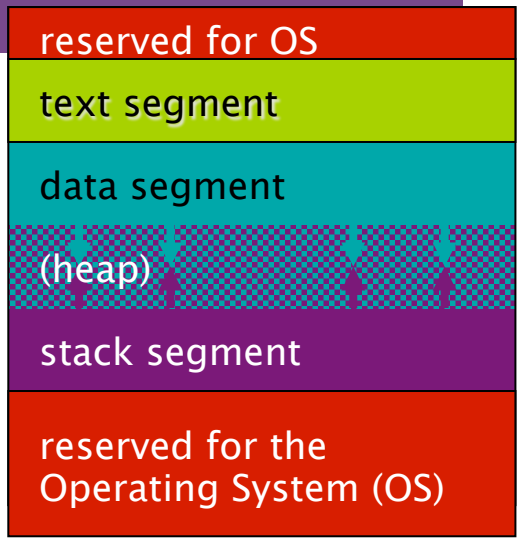
- **Starts at 0x10000000 (268435456 in decimal)**
- **Conceptually divided into two parts**
- **First: stores "static data". Static in the sense of "its size is known before execution":**
  - When the OS loads the program – values may change but no new memory is needed
- **This includes:**
  - Class 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

The Heap Segment



- **Technically part of data segment**
  - Located at the end, after all static data
- **All dynamically allocated memory comes from here**
  - All Python objects and their instance variables go here
- **Grows "downwards" as more memory is allocated**
  - 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**

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:**
  - Method information:
    - Local variables
    - Value of parameter
    - Return address
  - Saved register values
- **Note: this really is a stack like the ones we will see later in the unit**

Summary

- **MIPS R2000 architecture**
  - CPU registers
    - GPRs, PC, HI, LO, IR, MAR
  - Accessing memory locations
    - computed load/store addresses
  - Memory segments
- **Running programs**
  - Fetch-decode-execute cycle