Prepared by Julian García, based on material by
David Albrecht and María García de la Banda

Lecture 2
MIPS Architecture

FIT 1008
Introduction to Computer Science

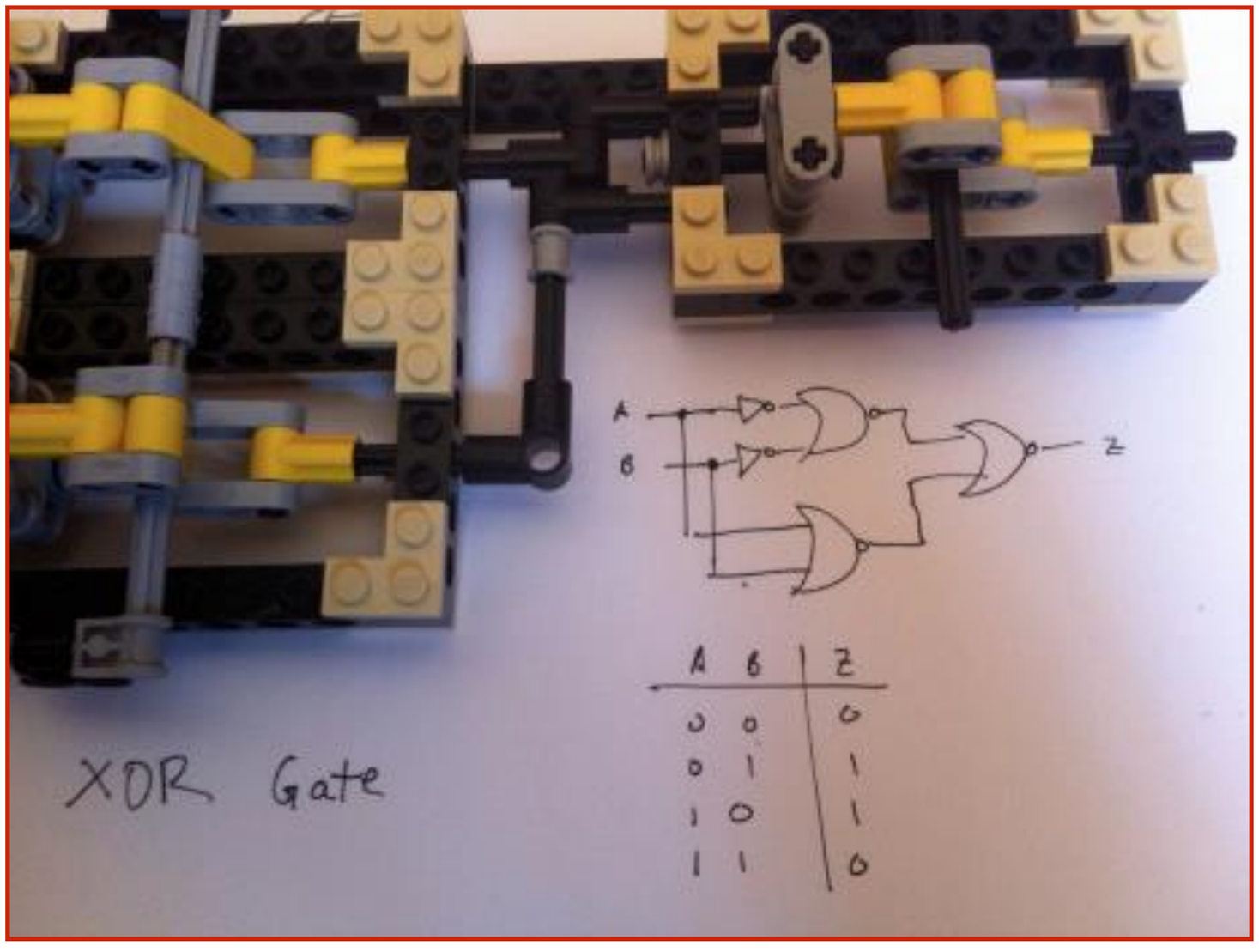
COMMONWEALTH OF AUSTRALIA
Copyright Regulations 1969
WARNING
This material has been reproduced and communicated to you by or on behalf of Monash University pursuant to Part VB of the Copyright Act 1968 (the Act).The material in this communication may be subject to copyright under the Act.Any further reproduction or communication of this material by you may be the subject of copyright protection under the Act.
Do not remove this notice.

```
1               .data
2   str:        .asciiz "Hello World!”
3
4               .text
5               la      $a0, str            # print str
6               addi    $v0, $0, 4
7               syscall
8               addi    $v0, $0, 10         # exit
9               syscall
```

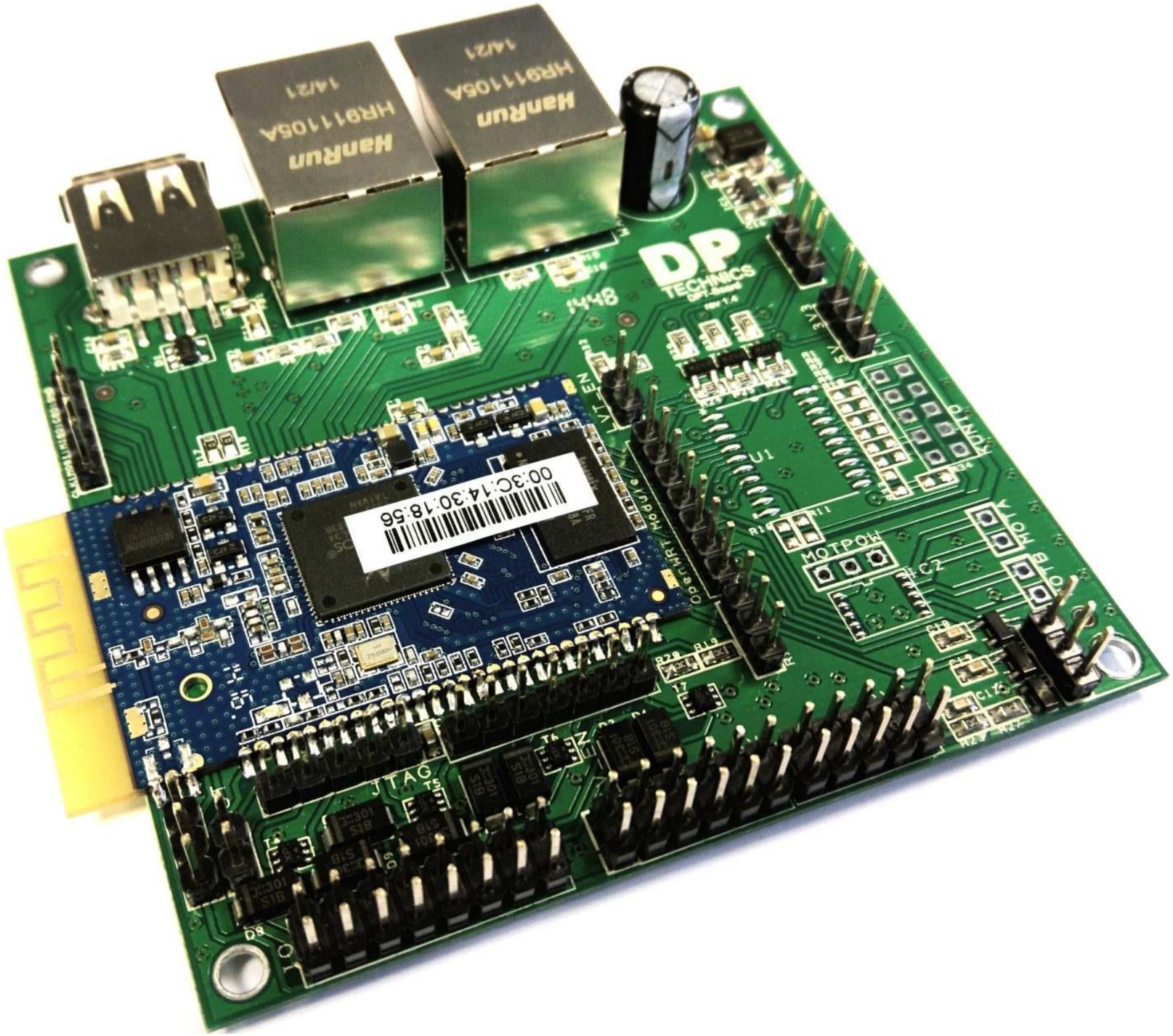
Objectives

- To understand how a Python program can run on a computer.
- To understand the MIPS R2000 architecture
  - ➞ Memory organisation
  - ➞ CPU registers
- To understand how programs are executed in this architecture
  - ➞ The fetch-decode-execute cycle
  - ➞ Accessing main memory

lone by large combinations of **on-and-off** switches, wired together in m



https://keshavsaharia.files.wordpress.com/2011/05/img_0270.jpg?w=460&h=343


HanRun
HR911105A
14/21
DP
TECHNICS
00:3C:14:30:18:56
JTAG
MOTPOW

0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111

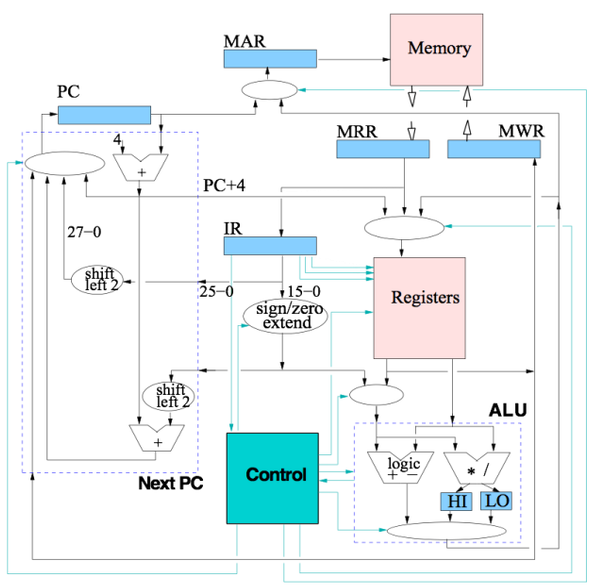
You

???

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

Human-readable code

Machine code



CPU

High level programming language

```
def find_duplicates(a_list):
    n = len(a_list)
    k = 0
    while k < n:
        j = k + 1
        while j < n:
            if a_list[k] == a_list[j]:
                print(a_list[k])
            j += 1
        k += 1
```

Assembly Language Program

```
main:   # 1 * 4 = 4 bytes local.

        addi $fp, $sp, 0

        addi $sp, $sp, -4

        sw   $0, -4($fp) # n = 0

        addi $v0, $0, 5

        syscall

        sw $v0, -4($fp)  # n
```

Machine code

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

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

Human-readable 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

Compiled

Compiled

Machine code

Executed by Virtual Machine

Executed

Output

Compiled

```
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

Machine Code

- To run programs on a computer we need machine code.
- Each machine code instruction is executed by the CPU.
- Machine code is hard to read.
- Corresponding to each machine code instruction is an assembly instruction.
- Assembly is easier to read.
- We will study MIPS Assembly Language

Objective:

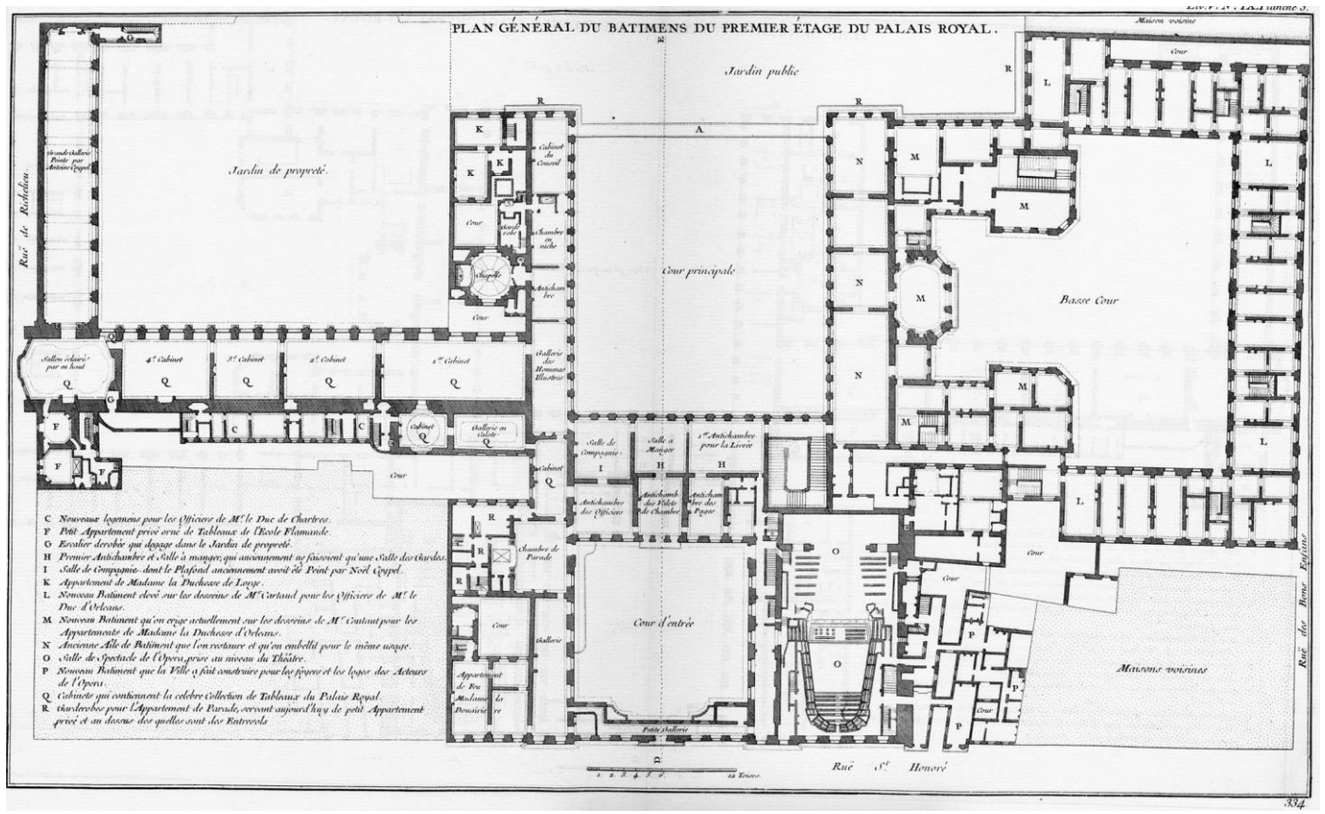
Understand MIPS architecture and MIPS Assembly language. Understand its instructions to be able to compile high level code to assembler (Variables, loops, if-then-elses, function calls, etc)

```
1               .data
2   str:        .asciiz "Hello World!”
3
4               .text
5               la      $a0, str            # print str
6               addi    $v0, $0, 4
7               syscall
8               addi    $v0, $0, 10         # exit
9               syscall
```

Why?

- Really understand how high level code works. Make you a MUCH better 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 optimisations made by the compiler

We need to know something about the machine….



[blueprint]

MIPS Architecture

- 1981: John L. Hennessy starts a research group at Stanford, focusing on RISC 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 is soon retiring, as President of Stanford University.

Why MIPS?

- A real processor (not a toy one). MIPS32 & MIPS64 still in production.

- Ancestor 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.

  Reduced Instruction Set Computer

- All instructions are
  - Same length (4 bytes)
  - 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.

  Complex Instruction Set Computer
  - 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 than x86 architecture is cumbersome with many confusing addressing modes and exceptions.

- MIPS is representative of modern computer architecture.

- MIPS has readily available simulators: http://courses.missouristate.edu/KenVollmar/MARS/

D:\2010F CSC 320\MIPS assembly code\FibonnaciWithStack.asm* - MARS 4.0.1

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                                          call for print string
15                 lui                                           expected at 0x10010000
16                                                               bits to zero.
17                 sysc
18
19                 addi                                          call for read integer
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
30 loop:  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
36 base:  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
40 notbase:
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)
```

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

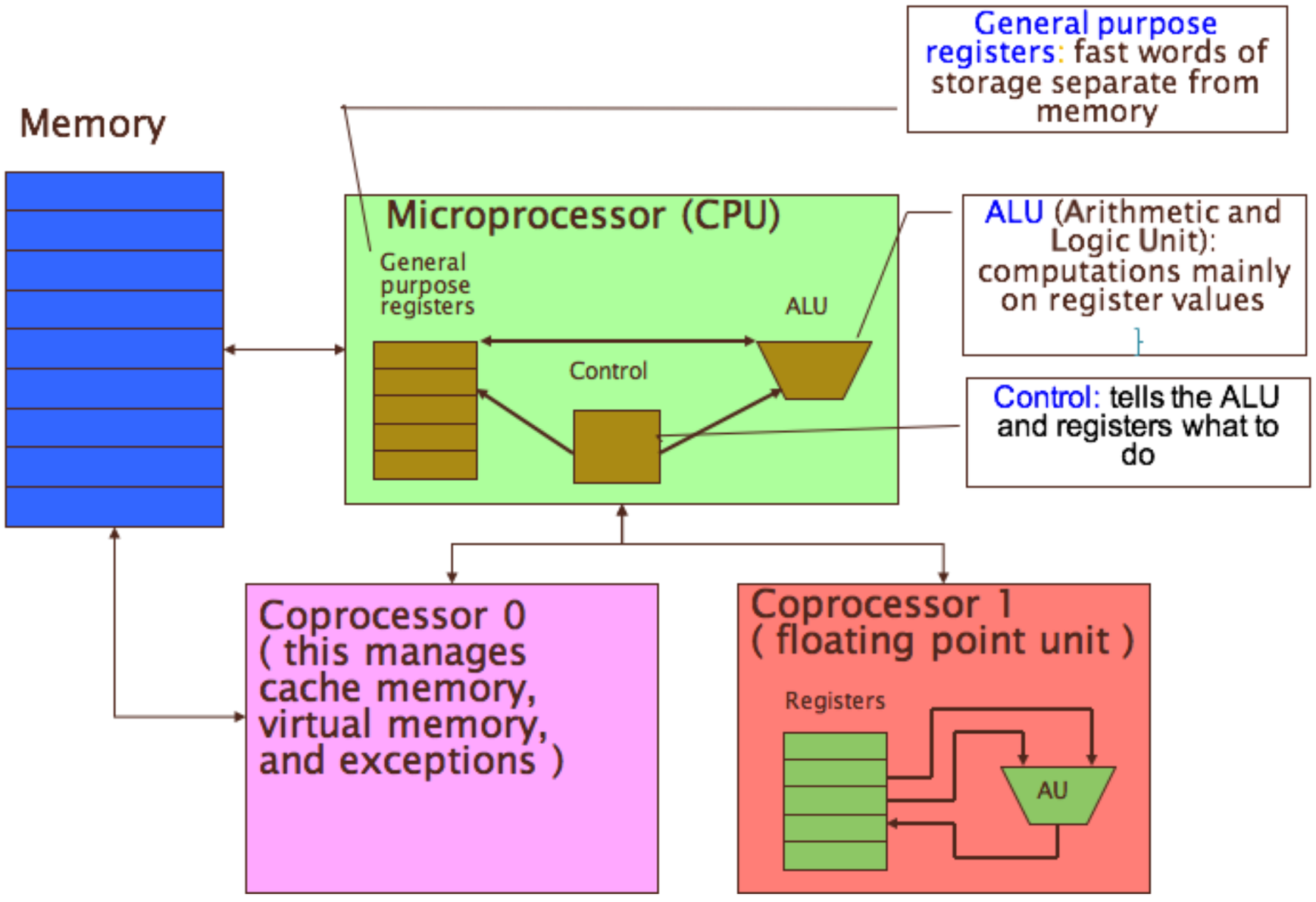
Clear

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

MARS

Simplified MIPS Architecture



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),

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: hard-coded
- Names increase readability
- Can theoretically be used in any way.
- Conventions assign certain uses to certain GPRs. Conventions help your program cooperate with others.

General Purpose **Registers** (GPRs)

used in FIT1008	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

General Purpose Registers

Register name	Register number	Typical use
$zero	$0	constant zero, cannot change, read only
$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)
$sp	$29	top-of-stack pointer
$fp	$30	stack frame pointer
$ra	$31	function return address

◀ Week 0: Revision

Week 1: Simple programs and Assembly

Learning Objective:

The first week is dedicated to:

- Revise simple Python Programs
- MIPS Architecture
- MIPS Simple programs

Please make sure you have read EVERYTHING included in Week 0. You will need it.

Documents for Tute 1

Tute 1 77.8KB PDF document

Documents for Prac 1

FIT1008 PracGuide

Workshop Week 1

Prac Submission

Workshop Week 1

Lecture Notes Week 1

Lecture 1: Introduction 3.4MB PDF document

Lecture 1: Introduction (with animations) 5.1MB PDF document

Lecture 2: MIPS Architecture 12.9MB PDF document

Lecture 3: Assembly Programming 603.4KB PDF document

Resources

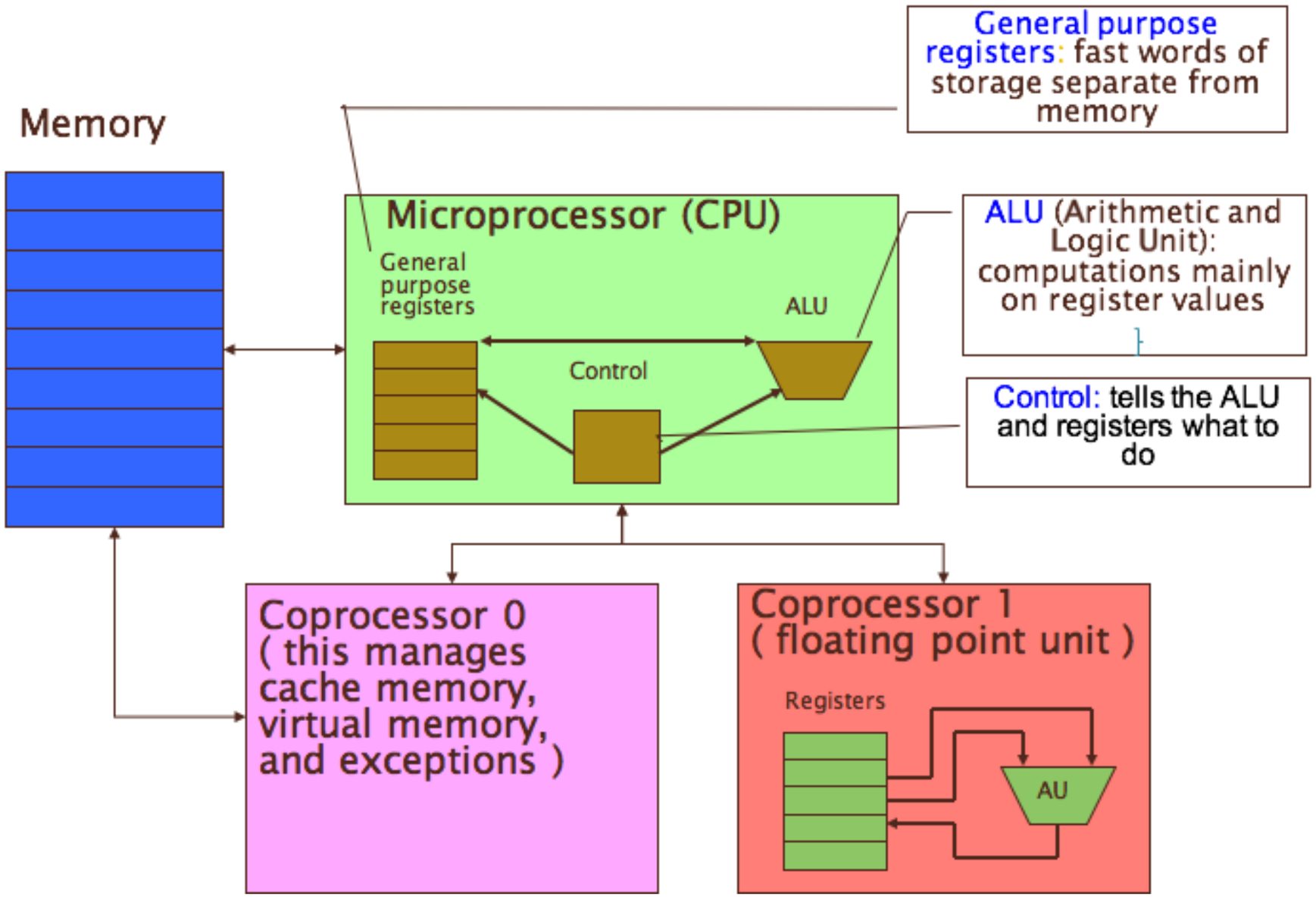
MIPS Pamphlet (epub available)

MIPS Reference Sheet 100.2KB P F document

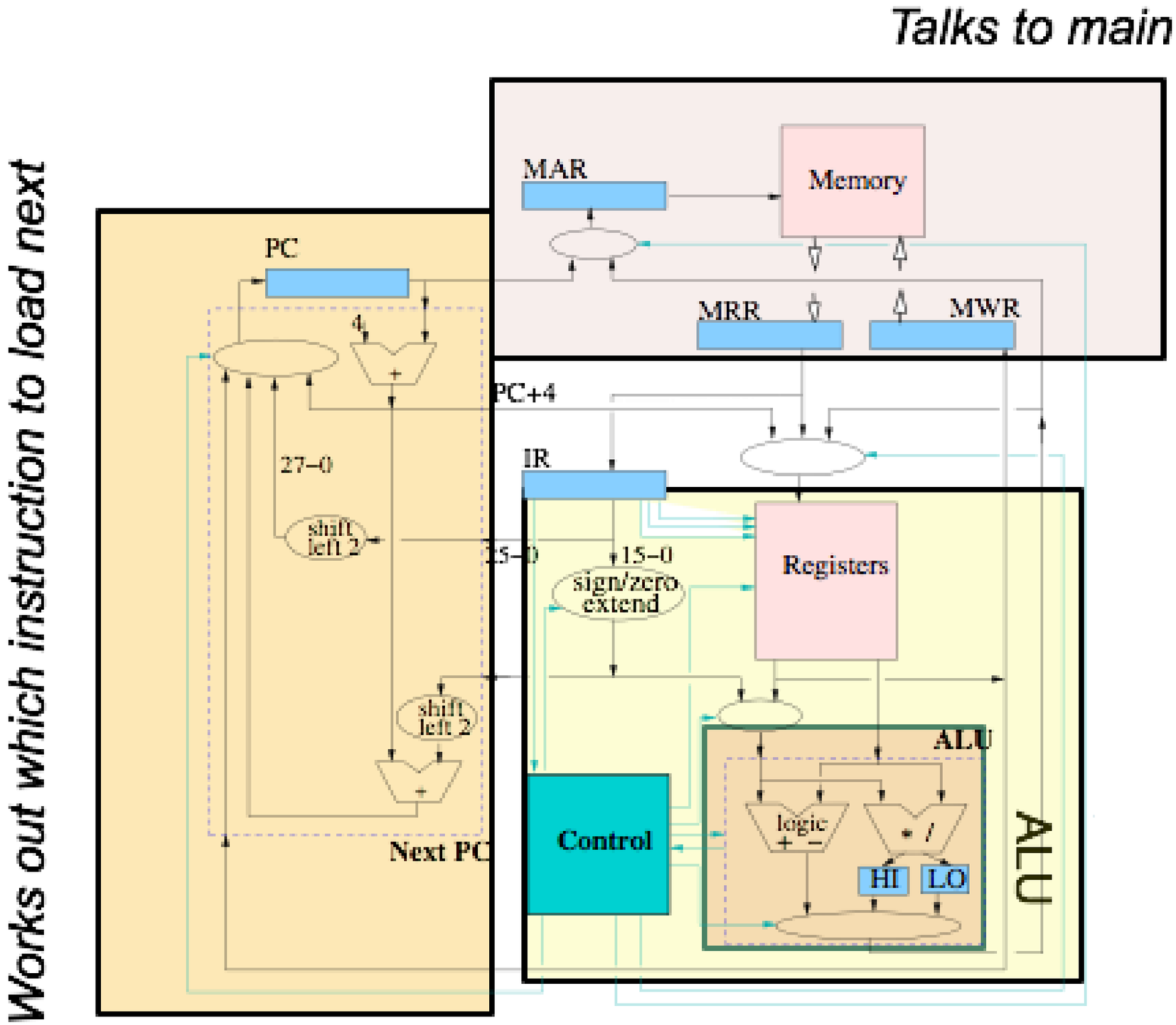
MIPS_AppendixA

Lecture MIPS code

Simplified MIPS Architecture



Inside the MIPS microprocessor (again)



special registers

HI

LO

PC

IR

MAR

MRR

MWR

HI and LO Registers

- Multiplying two 32 bits numbers might require 64 bits to fit
- After an integer multiplication:
  - HI contains the “high” 32 bits
  - LO contains the “low” 32 bits
- Integer division might be used to get the result or to get the reminder
- After an integer division:
  - LO contains the result
  - HI contains the reminder
- There are instructions to move the contents of LO or HI back to a GPR

IR Register

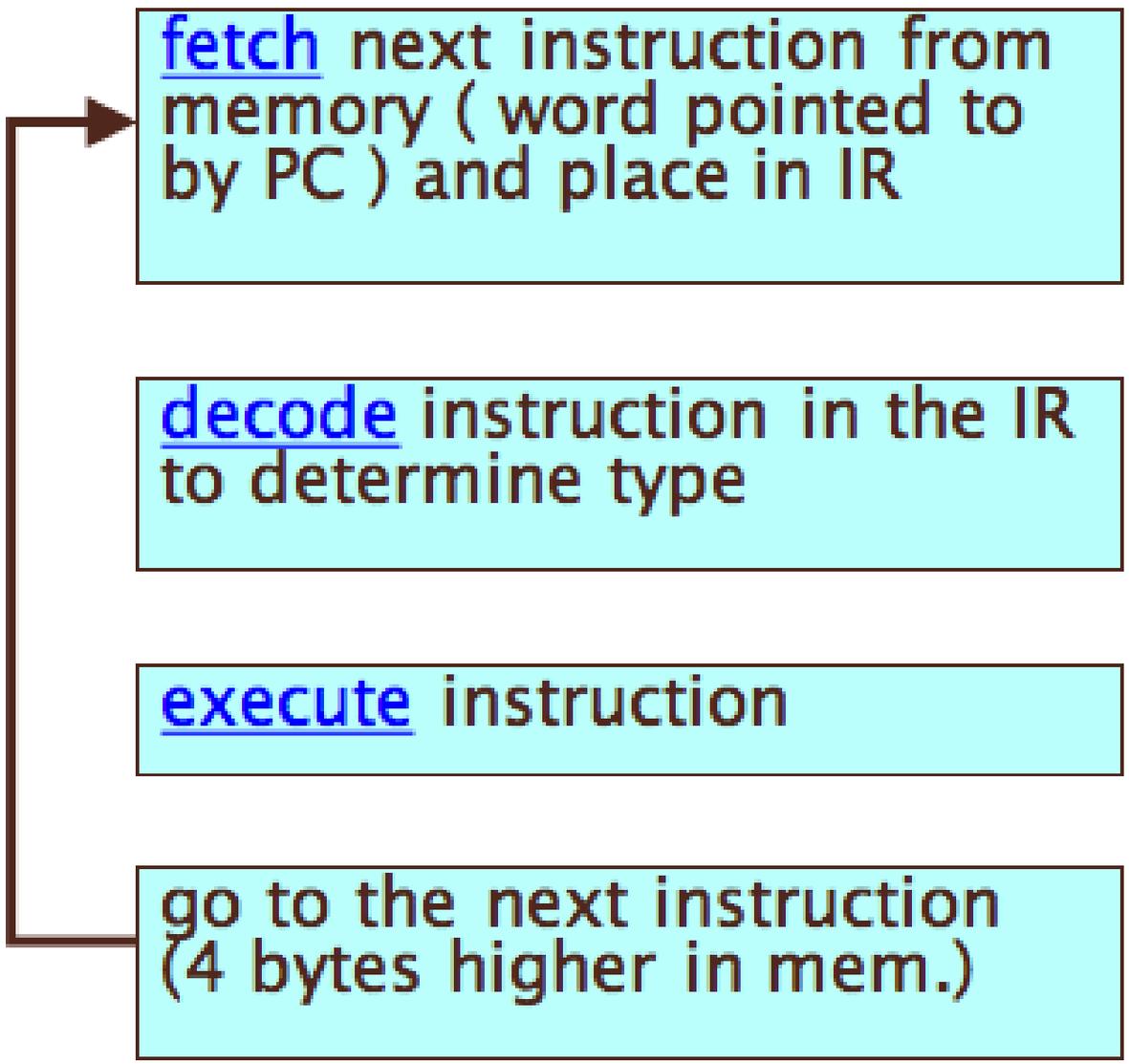
- The Instruction Register stores the instruction currently being executed (32 bits)
- Some bits encode the kind of instruction
- The rest of the info depends on the opcode's value
- The opcode tells the control circuitry which set of microinstructions needs to be followed

PC Register

- The Program Counter acts as a bookmark:
  - Tells the computer where is it up to
- Holds the memory address of the machine instruction currently being executed
- Advanced (PC=PC+4) by most machine instructions to point to next instruction
- Jump instructions write a new value to PC to move execution to a new place in program (loops, calls)
- Branch instructions do this only if a given condition is met (if statements)

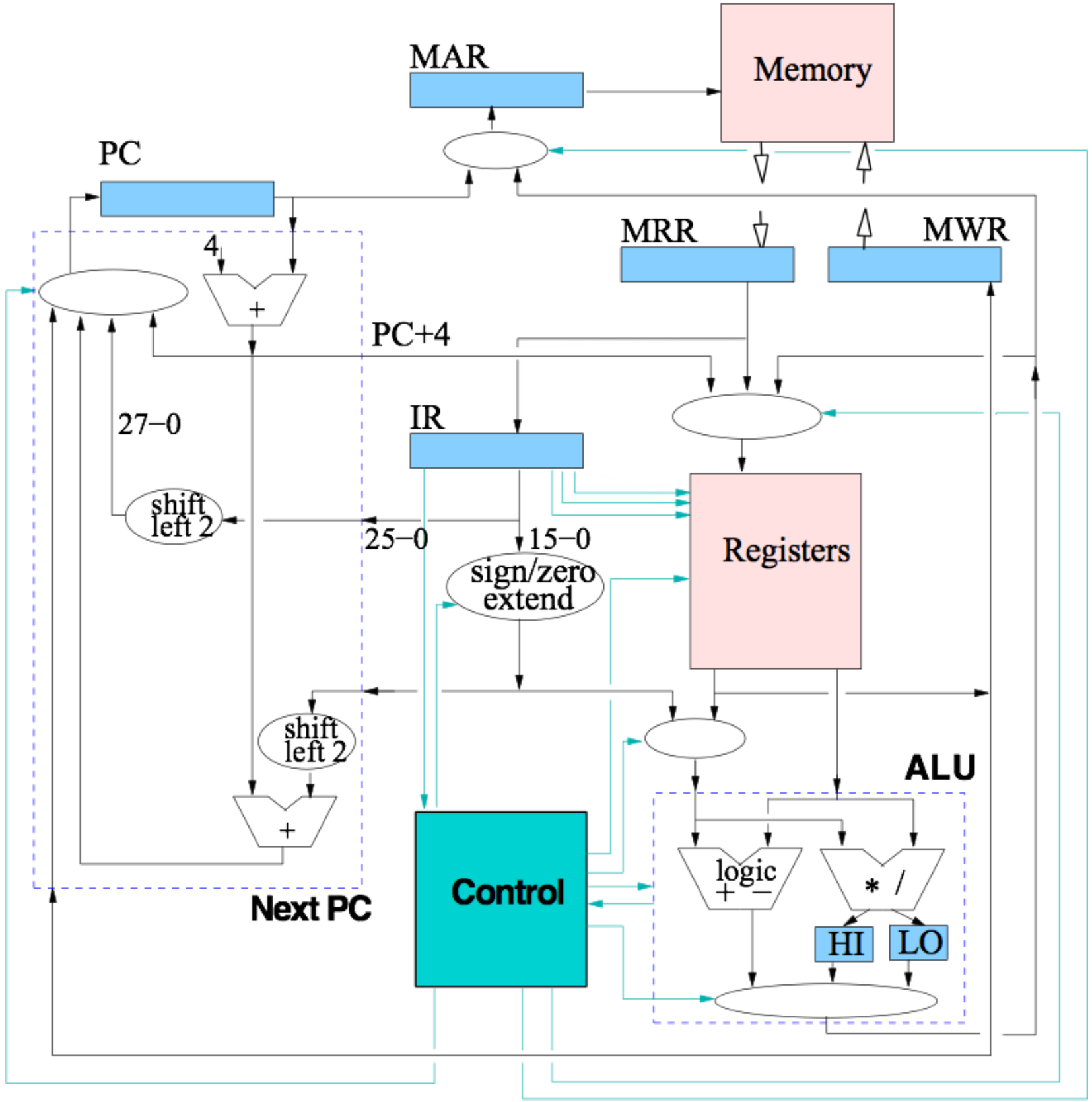
MIPS Instruction Execution

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



von Neumann architectures

- Separate modules for Processing and Main memory
- Programs are stored in main memory
- Each instruction must be loaded into the processor before execution


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

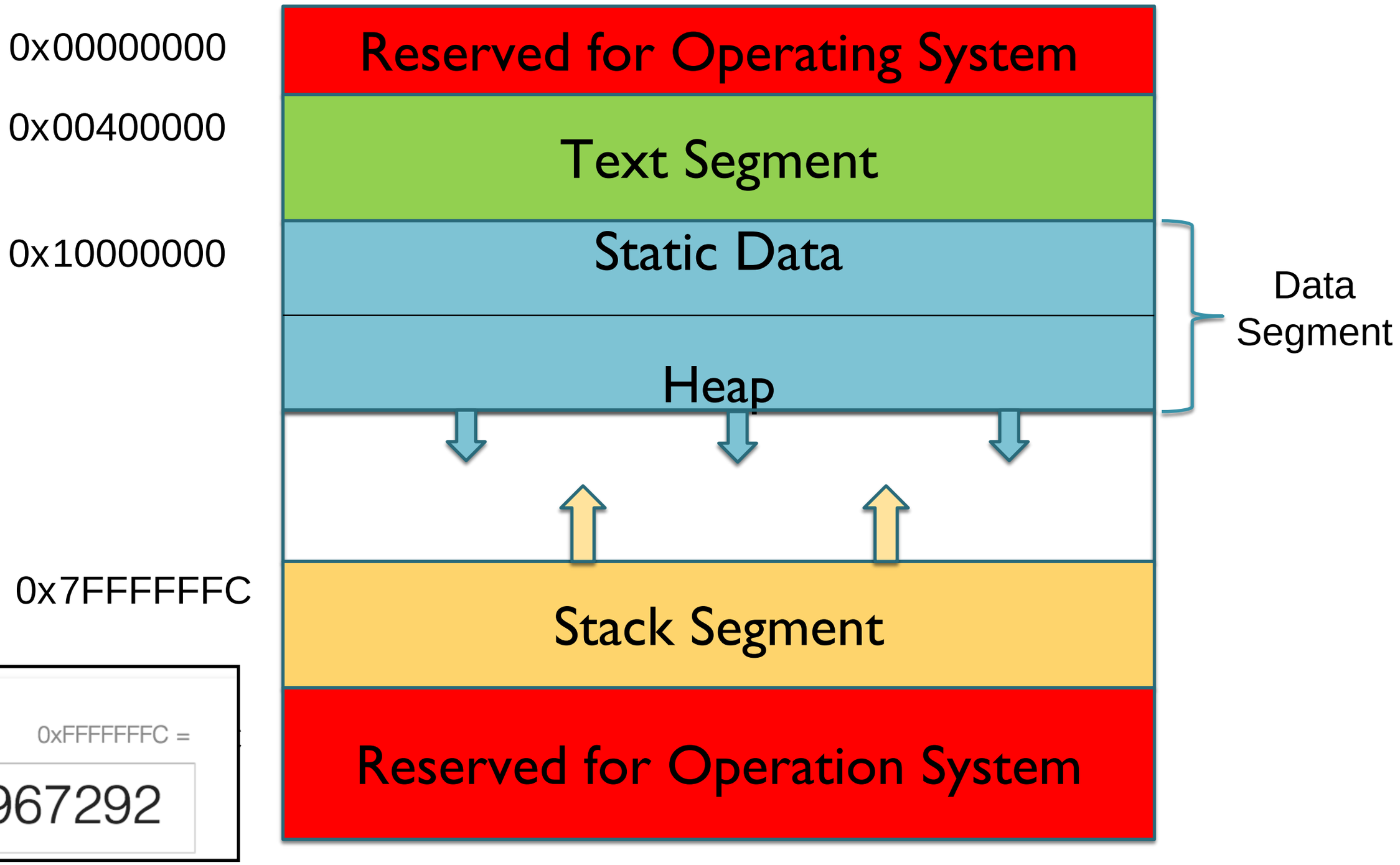
Accessing Main memory

- MIPS is a load-store architecture:
  Computations take place in registers

- Programs and their data live in main memory, not CPU
  - We need 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 Architecture: Memory



0xFFFFFFFC =

4294967292

Memory addresses

- Memory addresses in MIPS are 32 bits long and unsigned

- Each address refers to one byte of memory. Total potential address space: 4 Gb ≅4294967296 bytes ≅ 2^{32}.

- Usually only use addresses for words. These addresses are multiples of 4.

Getting data out of main memory

- We use a load instructions:
  - We can load 1, 2, or 4 bytes at a time
  - In this unit, usually 4: **“load word” or lw**

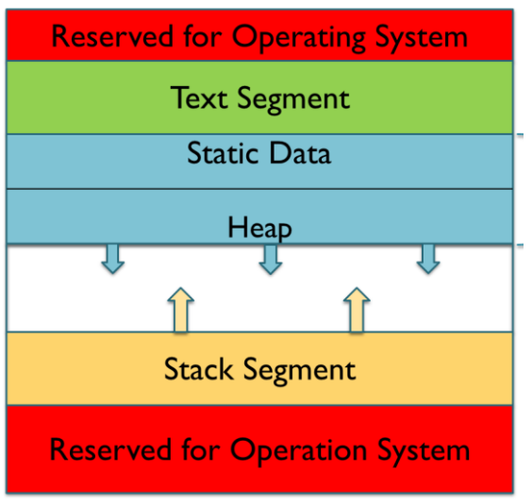
- To execute a load instruction:
  - The address to be loaded from goes into the MAR
  - The memory controller gets told to do a read
  - The data at that address goes into the MRR
  - The MRR is copied to the destination register (GPR) specified in the instruction

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 4: **“store word”** or sw

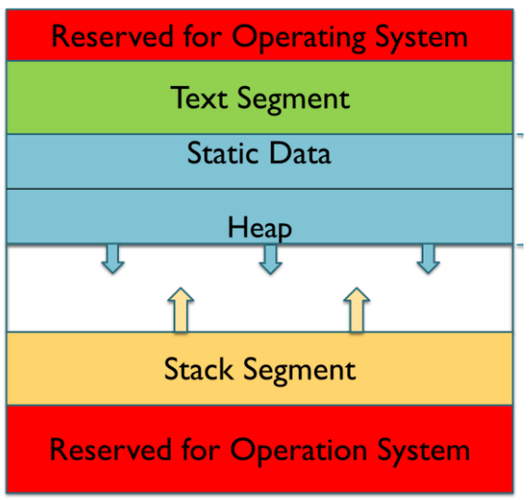
- To execute a store instruction:
  - The contents of the GPR specified in the instruction are copied to the MWR
  - The address to be stored to goes into the MAR
  - The memory controller gets told to do a write
  - The data in the MWR gets written to memory

The Text Segment



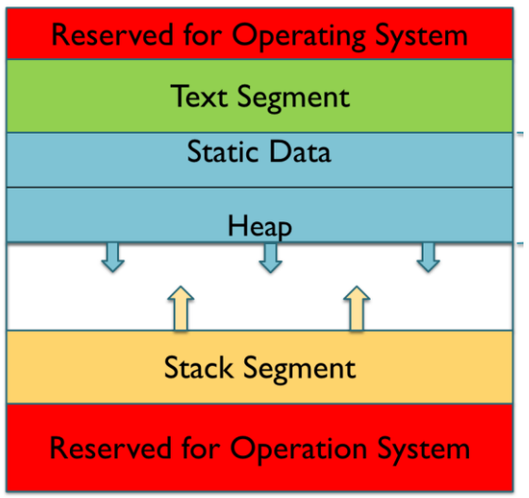
- Starts at 0x00400000 (that's 4194304 in decimal)
- Executable code goes here. In machine-readable format (remember?)
- PC register value is effectively a CPU "reference" into the text segment.
- How does code get here? For compiled programs, the OS puts it there when you tell it to run a program. This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

The Data Segment



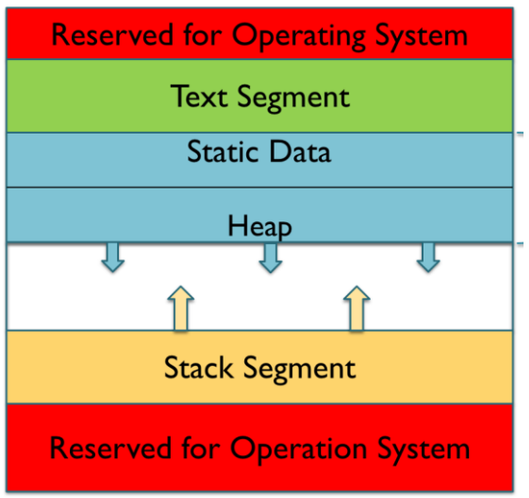
- Starts at 0x10000000
- Contains programs static data
  - Global variables
  - String constants
- Contains program dynamically allocated memory
  - Heap segment

The Heap Segment



- Grows “downwards” as more memory is allocated.
- 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 0x7FFFFFFC
- Grows “upwards” towards the Data segment
- Contains System stack
  - Local variables in functions
  - Function arguments
  - Function return address
  - Saved registers
  - Frame pointer

```
1               .data
2   str:        .asciiz "Hello World!”
3
4               .text
5               la      $a0, str            # print str
6               addi    $v0, $0, 4
7               syscall
8               addi    $v0, $0, 10         # exit
9               syscall
```

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