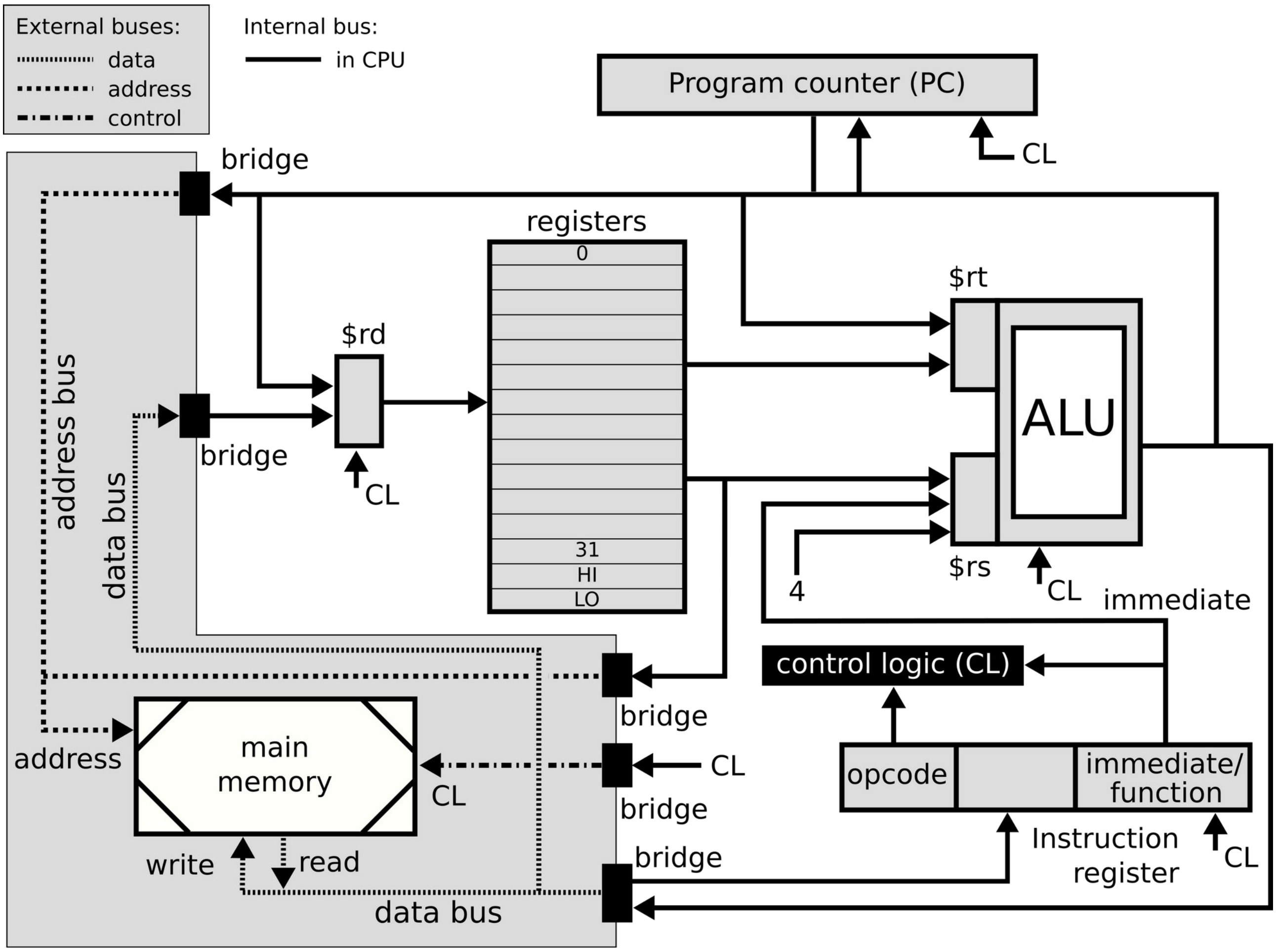
Computer Architecture Exercise 3 MIPS Architecture Peter Stallinga Universidade do Algarve 2024-2025	

The basic data path (data flow) of MIPS is given below:

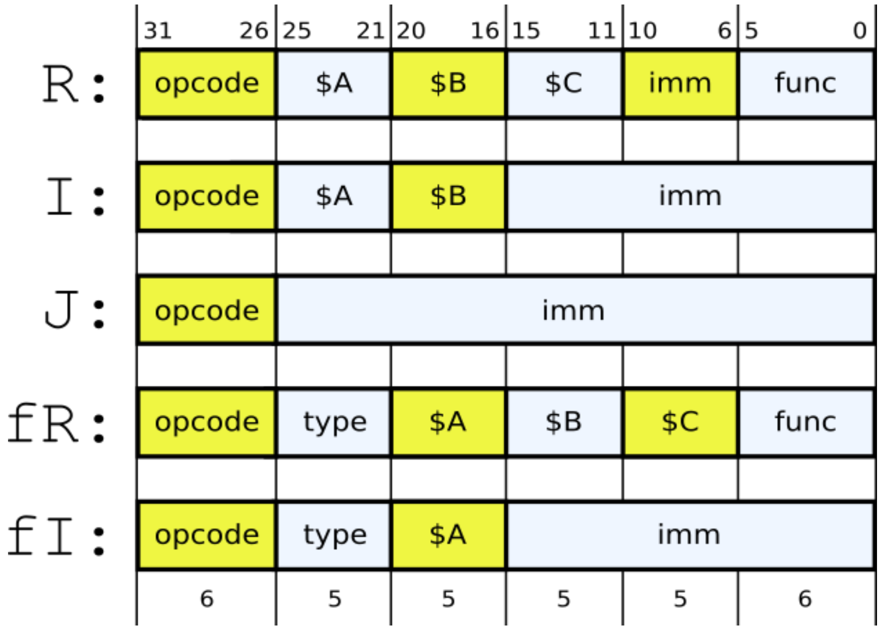


In each cycle:

- An instruction pointed at by the **program counter** is 'fetched' (copied from **main memory** to the **instruction register**).
- The **program counter** is increased by **4**.
- The **control logic** decodes the **opcode** and loads the operands (either from **main memory**, the **registers** or the **program counter**) into the **ALU** registers **$rs** ('source') and **$rt** ('target').
- The ALU performs the action chosen by the **control logic** (**CL**).
- The **control logic** stores the **ALU** result (either in **main memory**, a **register** or the **program counter**).

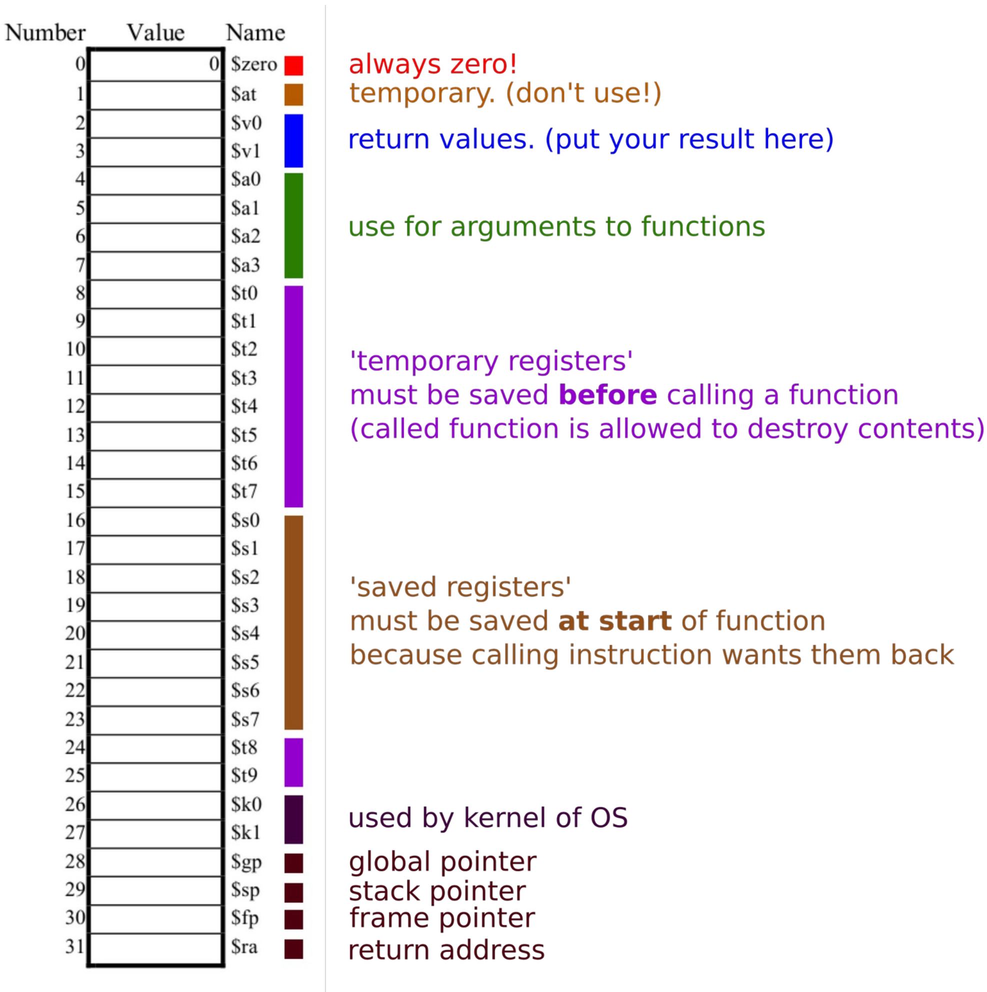
The most important to remember are: the program counter, the registers, and memory.

The MIPS instructions have the following format:



The code on the left indicates Register, Immediate, Jump, floating-point Register, and floating-point Immediate instruction types.

MIPS has 32 32-bit registers and are given as follows:

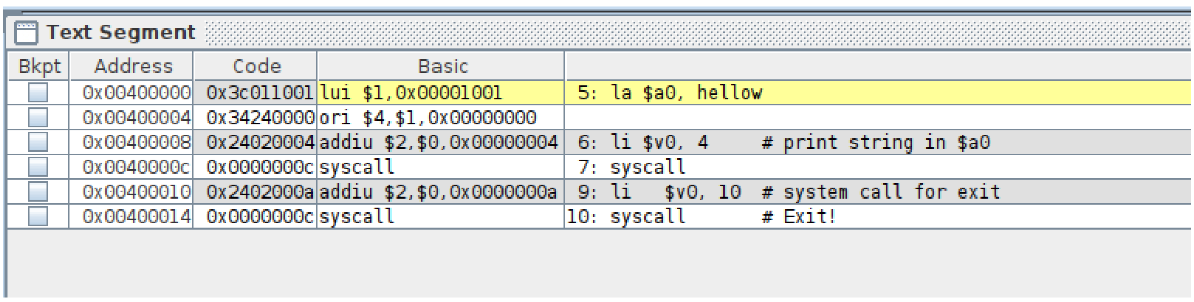


- Note the peculiar always-zero register `$0`, so that unary operations, like copy to another registerm can be done by binary operations (`$0`+`$1` --> `$2`, etc).
- Do not use the `$at` (`$1`) register. It is for the compiler
- Put the (final) results in `$v0` and `$v1`. These are like the return values in C
- Likewise the `$a` registers are used for passing arguments to functions
- The `$t` (temporary) registers are for our calculations
- The `$s` (saved) registers are for used inside functions
- The `$k` (kernel) registers are for the operating system
- The global pointer and frame pointer are for paging, the stack pointer is for the stack (duh) (important, later for functions).
- The return address is for using function calls (to remember where we have to jump back to after the function call finished)

Load the program `hellow.asm` into MARS

```
.data

hellow: .asciiz "Hello World!"

.text

la $a0, hellow
li $v0, 4       # print string in $a0
syscall

li   $v0, 10   # system call for exit
syscall        # Exit!
```

Compile the program and find:



Text Segment

Bkpt	Address	Code	Basic	
	0x00400000	0x3c011001	lui $1,0x00001001	5: la $a0, hellow
	0x00400004	0x34240000	ori $4,$1,0x00000000	
	0x00400008	0x24020004	addiu $2,$0,0x00000004	6: li $v0, 4 # print string in $a0
	0x0040000c	0x0000000c	syscall	7: syscall
	0x00400010	0x2402000a	addiu $2,$0,0x0000000a	9: li $v0, 10 # system call for exit
	0x00400014	0x0000000c	syscall	10: syscall # Exit!

First of all, note that the data segment does not have variables. (Variables do not exist in Assembly!) It only has labels (like `hellow`), which are simply ways for the assembler to remember the memory addresses. Everytime it sees a new label (in the data segment or code segment) it will remember that in a compiler table and then in the code segment when the label is used, it will look up the value in the table and substitute it. So

```
hellow: asciiz "Hello World!"
```

will make MARS remember that `hellow` has the immutable (constant, not variable) value of `0x10010000`. Then, in the code segment

```
la $a0, hellow
```

will be translated by MARS at compiletime into

```
la $a0, 0x10010000
```

before it translates the instruction into machine language. At run-time the label `hellow` does not exist! The text `"Hello World!"` does exist at run time in memory.

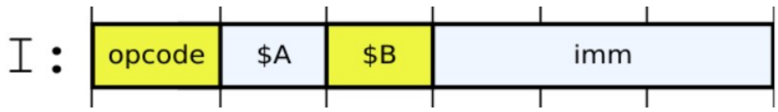
Furthermore, note how the loading of a memory address, `la $a0, hellow`, has been translated into *two* (2) instructions

```
lui $1, 0x00001001
ori $4, $1, 0x00000000
```

That is because a 32-bit address plus the opcode and the register specification (see the I instruction type above) can never fit into a 32-bit instruction. That is why it is loaded in two steps, each step loading a halfword (16 bit). Load-upper-immediate (`lui`) and then OR-immediate (`ori`).
That is why `la` is a so-called pseudo-instruction, not part of MIPS. But our MARS compiler can take care of it.

The MIPS Reference Card is in the annex.

This is how it works: We see that the data start at address `0x10010000` and there is where our data `"hello world!"` will be placed. This adress has to be loaded into register `$4` (`$a0`). The first step is to load (immediate) the high halfword (`0x1001`) into `$1` (`$at`) with `lui`.
`lui` is a I-type instruction which has format:



In the reference card we find that `lui` has the opcode `001111`. The `lui` instruction does not use the 5-bit $A field (thus: `00000`), but for the `$B` field we specify register `$1` (thus 00001) as destination. The final instruction thus becomes

opcode	$A	$B	imm
`0011 11`	`00 000`	`0 0001`	`0001 0000 0000 0001`

And that is `0x3c011001`, as can be seen in the machine-language column of the compiled program

Show that `ori $a0, $at, 0x00000000` is indeed instruction `0x34240000`.

What is the hexadecimal machine code for `add $s0, $t0, $t1`?

What is the MIPS Assembly for MIPS machine language `0x71288002`?