

# MIPS

*CS31*

*Pascal Van Hentenryck*



# MIPS

## A real assembly language

- Basic architecture
- ALU operations
- Memory operations
- Branch
- I/O
- Control Structures

# MIPS Architecture

Used in several commercial workstations.

- including the machine which is displaying the beautiful slide you are looking at right now

## Basic features

- 32 registers
- Memory
- load/store architecture
- RISC (few instructions)

## Load/store architecture

- All ALU operations are performed on registers
- Load/store are used for memory accesses

# MIPS Registers

## Real Names

\$0 . . . \$31

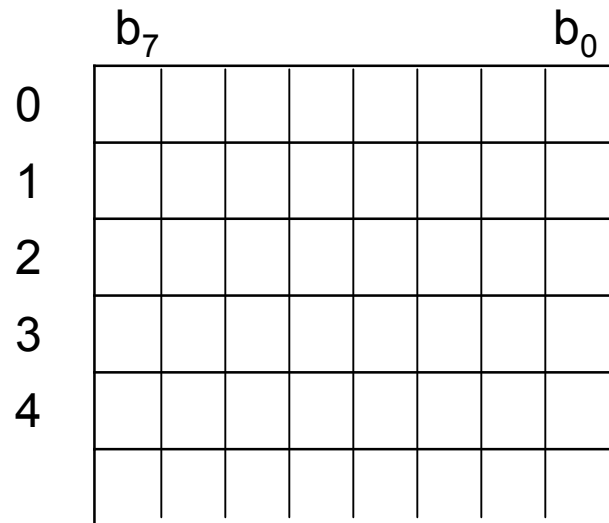
## Nicknames

```
$r0  $at  $v0  $v1
$a0  $a1  $a2  $a3
$t0  $t1  $t2  $t3
$t4  $t5  $t6  $t7
$s0  $s1  $s2  $s3
$s4  $s5  $s6  $s7
$t8  $t9  $k0  $k1
$gp  $sp  $s8  $ra
```

For now, we'll just use `$s0 - $s7`

# MIPS Memory

MIPS is byte addressable, i.e. its memory is a large array of bytes



Each time you increment an address, MIPS refers to the next 8-bit byte

In Java terms

```
bytes[] m = new bytes[size of memory];
```

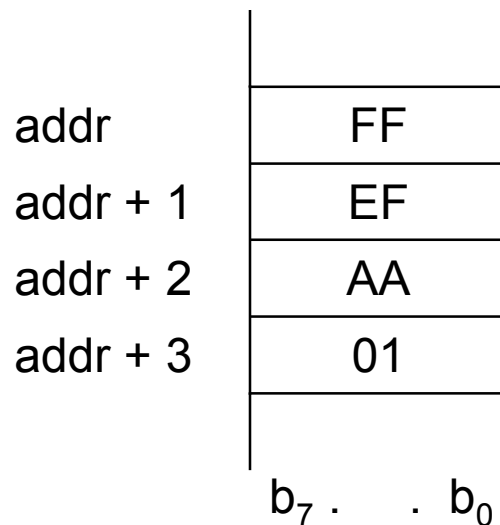
- `m[3]` refers to the 4<sup>th</sup> byte in memory
- 3 is the address of that byte

# MIPS Memory (Words)

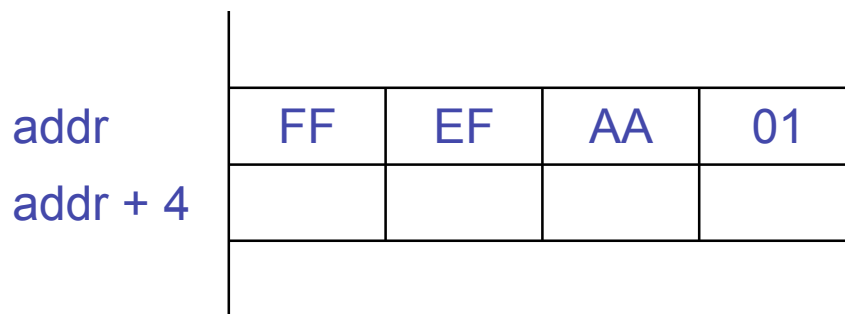
MIPS is an alignment machine

- word addresses are multiples of 4

MIPS is big-endian (versus little-endian)



If you examine a word (4 bytes) in memory, its high-order byte comes first



# MIPS Assembly Program

## Two main parts

- data section (declarations)
- text section (instructions)

## Two important elements

- `__start`: label of the first instruction
- `done` last instruction

# Variable Declaration

In Java:

```
int name;
```

In MAL:

```
{name:} .word {value}
```

```
ten: .word 10
```

```
eleven: .word xB
```

- label and value are optional
- no type checking

# Declaring Characters

## In Java:

```
char name;
```

## In MAL:

```
{name:}      .byte      {value}  
  
fizz:        .byte      'z'  
fuzz:        .byte      122  
linefeed:    .byte      '\n'
```

# Declaring Arrays

```
name: .type initial_value:num_elements
```

```
a:      .byte      1:7
```

```
b:      .word      1:3
```

a:	01	01	01	01
	01	01	01	??
b:	00	00	00	01
	00	00	00	01
	00	00	00	01

When you declare something of type word, it is automatically word-aligned (address is a multiple of 4)

# Constants

## In Java

```
static final int tax = 7;
```

## In MAL:

```
tax = 7
```

# ALU Operations

`opcode dest, source1, source2`

- `dest` must be a register
- `source1` must be a register
- `source2` can be a register or a constant

## Opcodes

- `add`                      `and`
- `sub`                        `or`
- `mul`                        `xor`
- `div`                        `nor`
- `rem`

`opcode dest, source`

## Opcodes

- `move`                      `not`

# Load Instructions

## Load Word

**lw**                      **R, address**

Takes a word (4 bytes) of data starting at address and loads it into register R.

Address *must* be word-aligned.

## Load Byte

**lb**                      **R, address**

Takes a byte of data starting at address and loads it into the low-order byte of register R, then sign-extends to 32 bits.

## Load Byte Unsigned

**lbu**                      **R, address**

Same as load byte, but instead of sign-extending, high-order 3 bytes of R are set to 0

# Load Practice

		00001000	01	FF	02	03
		foo	4A	BC	12	3F
lw	\$s0, 0x00001000					
s0						
lw	\$s0, 0x00001002					
s0						
lb	\$s0, foo					
s0						
lb	\$s0, 0x00001001					
s0						
lbu	\$s0, 0x00001001					
s0						

# Specifying Addresses

## Addressing modes

label (direct): `foo`

“the contents of address `foo`”

constant (direct): `x00010004`

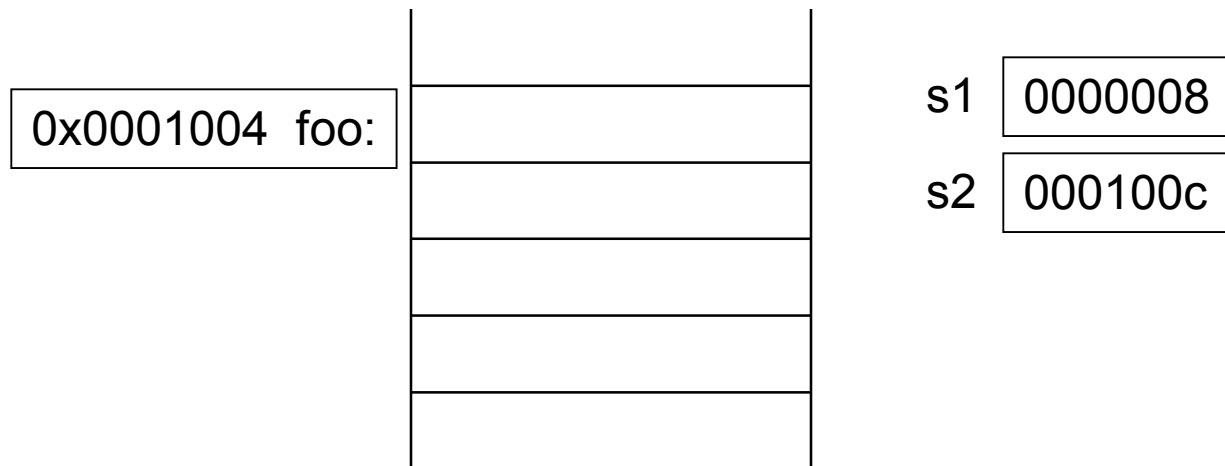
“the contents of address `x00010004`”

register (indirect): `($s2)`

“the contents of the address stored in `$s2`”

indexed (relative): `foo($s1)` or `4($s2)`

“the contents of the address obtained by adding the contents of `$s1` to `foo`”



# More Load Instructions

## Load Immediate

`li R, constant`

Loads the constant into register R

## Load Address

`la R, label`

Loads the address specified by label into register R.

These are two names for the same thing.

<code>la</code>	<code>\$s0, foo</code>	00001000	01	23	45	67
<code>s0</code>	<code>00001004</code>	foo	AB	CD	EF	01
<code>lw</code>	<code>\$s0, foo</code>					
<code>s0</code>	<code>ABCDEF01</code>					
<code>li</code>	<code>\$s0, 0x00001000</code>					
<code>s0</code>	<code>00001000</code>					