

# 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.

## Basic features

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

## Load/store architecture

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

# MIPS Registers

## Real Names

\$0 . . . \$31

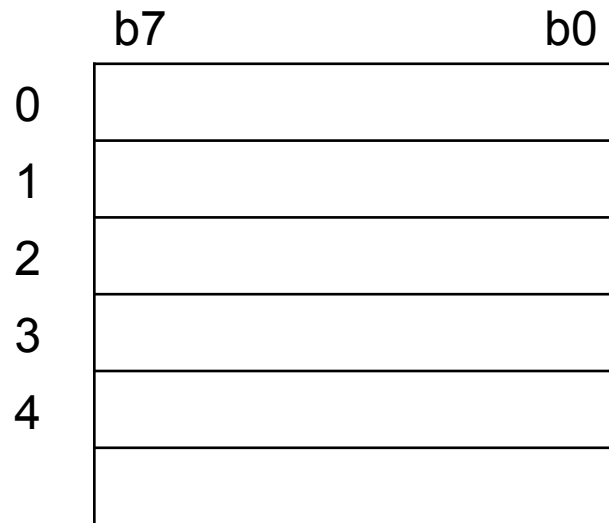
## Nicknames

\$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];
```

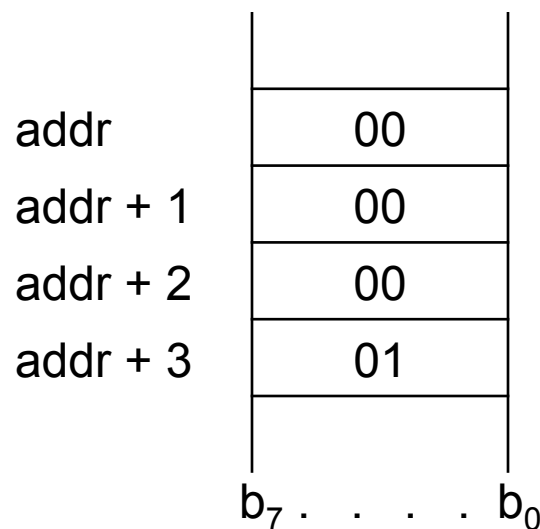
- `m[3]` refers to the 4<sup>th</sup> byte in memory
- 3 is the address of that byte

# MIPS Memory (Words)

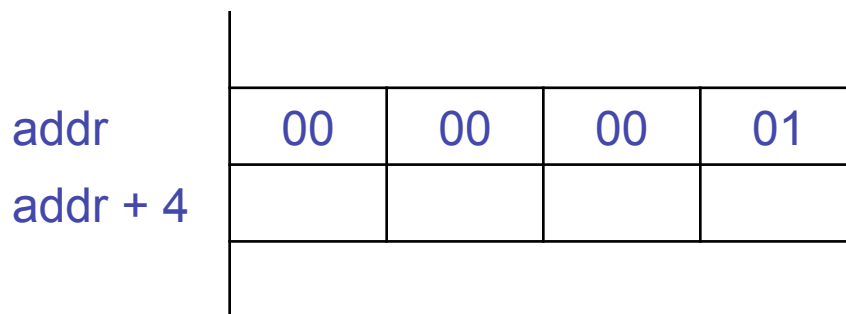
MIPS is an alignment machine

- word addresses are multiples of 4

MIPS is big-endian



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



# Specifying Addresses

## Addressing modes

label (direct): `foo`

“the contents of address `foo`”

constant (direct): `x00010004`

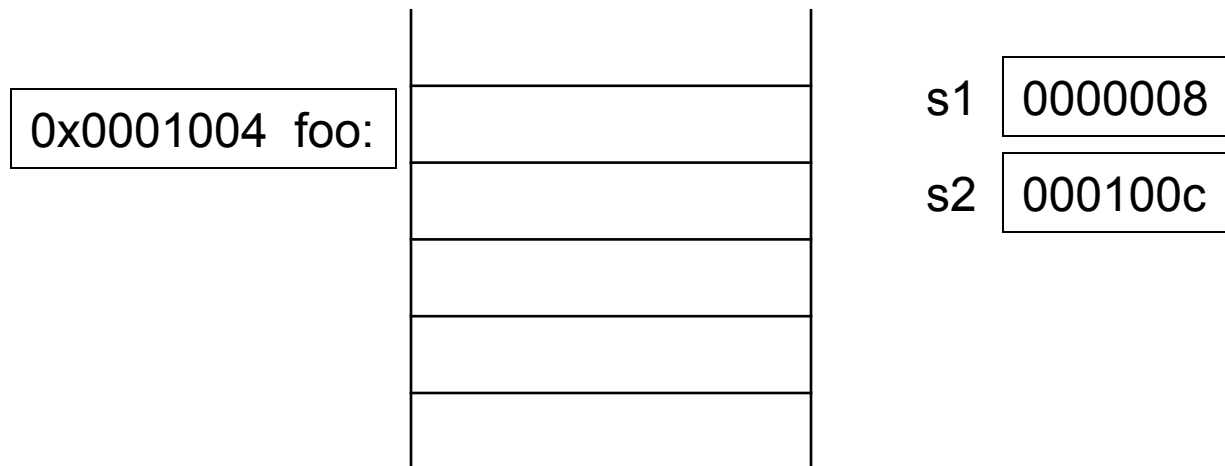
“the contents of address `x00010004`”

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`”



# Store Instructions

## Store Word

**sw R, address**

Store the contents of register R at address in memory. Address must be word-aligned.

## Store Byte

**sb R, address**

Store the low-order byte of register R at address in memory.

```
$s0  AB CD EF 02
sw   $s0, foo
sb   $s0, bar
la   $s1, foo
sw   $s0, 8($s1)
sw   $s0, 10($s1)
```

foo	AB	CD	EF	02
bar	02			
	AB	CD	EF	02

# Sample Program

## Average 3 integers:

```
                .data
avg:            .word          #integer average
i1:            .word 20        # first input
i2:            .word 13        # second input
i3:            .word 10        # third input
num = 3        # nb of nums

                .text
__start:       lw      $s1,i1
               lw      $s2,i2
               lw      $s3,i3
               add     $s0,$s1,$s2
               add     $s0,$s0,$s3
               div     $s0,$s0,num
               sw      $s0,avg
               done
```

# Branch Instructions

`bxx`                      `R1, R2, label`

- **R1** and **R2** must be registers.
- label can be a label (or a constant).
  - `beq`
  - `bgt`
  - `bge`
  - `blt`
  - `ble`
  - `bne`

Remember the short forms.

`bxx`                      `R1, label`

- `bltz`
- `bgez`
- `blez`
- `bgtz`
- `beqz`
- `bnez`

# Jump

## Unconditional Jump

```
j label
```

## Unconditional register jump

```
jr r
```

# Input / Output

We'll study this at great length later, but for now we'll use simple system calls.

Every system call has

- a call number
- arguments

You load these values into special registers, then do a magic `syscall` command.

Your code will continue where you left off.

# Printing

To print an integer:

```
lw      $a0,num    # the value to print
li      $v0,1     # code to print integer
syscall                # do it
```

To print a string:

```
.data
str:    .asciiz "Are we having fun yet?\n"
.text
la      $a0,str    #address of the string
li      $v0,4     # code to print string
syscall                # do it
```

All the characters will be printed until a NULL (0) is reached.

The **.asciiz** declaration automatically puts a NULL at the end of the string.

# A Stupid Program

```
# Loop, asking and answering silly questions
# until somebody says they're
# 999 years old and 999 inches tall.
#
# There are 3 years left between now and the
# year 2006, so the answer is
# height + (height/age)*3
    .data
age_question:
    .ascii "How old are you (in years)?"
height_quest:
    .ascii "How tall are you (in inches)?"
answer1:
    .ascii "Assuming linear growth, you'll be"
answer2:
    .ascii " inches in the year 2003.\n"
were_done:
    .ascii "We're outta here."

# Register Usage
#     s0 : age
#     s1 : height
#     s2 : calculation

    .text
__start:
ask:    la    $a0, age_question
        li    $v0, 4
        syscall                # Ask age question
```

# A Stupid Program

```
li      $v0,5
syscall                                # Read age
move    $s0,$v0
la      $a0,height_question
li      $v0,4
syscall                                # Ask height question
li      $v0,5
syscall                                # Read height
move    $s1,$v0
bne     $s0,999,cont                  # We're okay
                                           # if age<>999
beq     $s1,999,end                  # If height =
                                           # 999, exit
cont:   mul    $s2,$s1,3              # height * 3
div     $s2,$s2,$s0                  # height * 3 / age
add     $s2,$s2,$s1                  # height +
                                           # (height * 3/age)
la      $a0,answer1
li      $v0,4
syscall                                # Print first part of answer
move    $a0,$s2
li      $v0,1
```

# A Stupid Program

```
syscall                # Print number
la    $a0,answer2
li    $v0,4
syscall                # Print 2nd part of
                    # answer
j     ask              # Repeat this loop
end:  la    $a0,were_done
      li    $v0,4
      syscall                # Print end message

done
```

# Structured Coding

- JAVA has statements that allow structured control flow
- One way to write well-structured assembly code is to
  1. design your algorithm in terms of high-level control structures
  2. express those structures in assembly language

# IF Example

```
if (a > b)
    c = a
else
    c = b
```

## Version 1:

```
    bgt    a,b,lab    # if a > b else
    move   c,b        #     c = b
    j      end        #
lab:      #
    move   c, a       #     c = a
end:      #
```

## Version 2:

```
    ble   a,b,lab    # if a > b
    move  c,a        #     c = a
    j     end        #
lab:     # else
    move  c,b        #     c = b
end:     #
```

# IF Example

```
if ($s1 > $s2)
    $s3 = $s1
else
    $s2 = $s2
```

## Version 1:

```
    bgt    $s1, $s2, lab # if a > b else
    move   $s3, $s2     #      c = b
    j      end         #
lab:      #
    move   $s3, $s1    #      c = a
end:      #
```

## Version 2:

```
    ble    $s1, $s2, lab # if a > b
    move   $s3, $s1     #      c = a
    j      end         #
lab:      # else
    move   $s3, $s2    #      c = b
end:      #
```

# Alternative Example

```
if a > b
    c = a
else if a < b
    c = b
else
    c = 0
```

```
ble    a,b,l1    # if a > b
move   c,a       #     c = a
j      end       #
l1:    # else
bge    a,b,l2    # if a < b
move   c,b       #     c = b
j      end       #
l2:    # else
move   c,0       #     c = 0
end:   #
```

# FOR Example

```
sum = 0;
for(count = 1; count <= 10;count++)
    sum = sum + count;
```

```
move    sum, 0           # clear the sum
move    count,1         # initialize count
move    temp,10        # store upper limit
for:
    bgt    count,temp,end # for(count <= 10)
    add    sum,sum,count  # sum = sum + count
    add    count,count,1  # count++
    j      for            #
end:    #
```

See style guide for more clever ways to do FOR loops.

# FOR Example

```
sum = 0;
for(count = 1; count <= 10;count++)
    sum = sum + count;
```

```
move    $s1, 0           # clear the sum
move    $s2, 1           # initialize count
move    $s3, 10          # store upper limit
for:
        bgt    $s2, $s3, end    # for(count <= 10)
        add    $s1, $s1, $s2    # sum = sum + count
        add    $s2, $s2, 1      # count++
        j      for             #
end:    #
```

See style guide for more clever ways to do FOR loops.

# WHILE we're at it

```
remain = dividend;
result = 0;
while (remain >= divisor) {
    remain -= divisor;
    result++;
}
```

```
while:                # WHILE rem >= divis

blt    remain,divisor,end
sub    remain,remain,divisor
                                # subtract divisor
add    result,result,1    # increment result
j      while            #
end:    # END WHILE
```