A MIPS processor consists of an integer processing unit and two coprocessors: coprocessor (0) handles traps, exceptions, and the virtual memory system; coprocessor (1) handles floating point processing.

### CPU Registers

<table>
<thead>
<tr>
<th>Name</th>
<th>Number</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>zero</td>
<td>0</td>
<td>Constant 0</td>
</tr>
<tr>
<td>at</td>
<td>$1</td>
<td>Reserved for assembler</td>
</tr>
<tr>
<td>v0</td>
<td>$2</td>
<td>Expression evaluation and results of a function</td>
</tr>
<tr>
<td>v1</td>
<td>$3</td>
<td>Argument 1-4</td>
</tr>
<tr>
<td>a0</td>
<td>$4–$7</td>
<td>Argument 1-4</td>
</tr>
<tr>
<td>t0</td>
<td>$8–$15</td>
<td>Temporary (not preserved across call)</td>
</tr>
<tr>
<td>s0</td>
<td>$16–$23</td>
<td>Saved temporary (preserved across call)</td>
</tr>
<tr>
<td>t8</td>
<td>$24–$25</td>
<td>Temporary (not preserved across call)</td>
</tr>
<tr>
<td>k0</td>
<td>$26–$27</td>
<td>Reserved for OS kernel</td>
</tr>
<tr>
<td>gp</td>
<td>$28</td>
<td>Pointer to global area</td>
</tr>
<tr>
<td>sp</td>
<td>$29</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>fp</td>
<td>$30</td>
<td>Frame pointer</td>
</tr>
<tr>
<td>ra</td>
<td>$31</td>
<td>Return address (used by function call)</td>
</tr>
</tbody>
</table>

- The CPU contains 32 general registers numbered 0–31.
- Register $0$ is always $\equiv 0$.

#### Register Use Conventions:

- Registers $at$ (1), $k0$ (26), and $k1$ (27) are reserved for use by the assembler and operating system.
- Registers $a0$–$a3$ (4–7) are used to pass the first four arguments to routines (remaining arguments are passed on the stack).
- Registers $v0$ and $v1$ (2, 3) are used to return values from functions.
- Registers $t0$–$t9$ (8–15, 24, 25) are caller-saved registers used for temporary quantities that do not need to be preserved across calls.
**CPU Registers...**

- Registers $s0$–$s7$ (16–23) are callee-saved registers that hold long-lived values that should be preserved across calls.
- Register $sp$ (29) is the stack pointer, which points to the first free location on the stack.
- Register $fp$ (30) is the frame pointer.
- Register $ra$ (31) is written with the return address for a call by the jal instruction.
- Register $gp$ (28) is a global pointer that points into the middle of a 64K block of memory in the heap that holds constants and global variables. The objects in this heap can be quickly accessed with a single load or store instruction.

**Addressing Modes**

- MIPS is a load/store architecture, which means that only load and store instructions access memory.
- Computation instructions operate only on values in registers.

<table>
<thead>
<tr>
<th>Format</th>
<th>Address Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(register)</td>
<td>contents of register</td>
</tr>
<tr>
<td>imm</td>
<td>immediate</td>
</tr>
<tr>
<td>imm (register)</td>
<td>immediate + contents of register</td>
</tr>
<tr>
<td>symbol</td>
<td>address of symbol</td>
</tr>
<tr>
<td>symbol ± imm</td>
<td>address of symbol + or − immediate</td>
</tr>
<tr>
<td>symbol ± imm</td>
<td>address of symbol + or − (immediate + contents of register)</td>
</tr>
</tbody>
</table>

**Integer Arithmetic I**

- $Src2$ is a register or an immediate value (a 16 bit integer).
- The assembler will translate the more general form of an instruction (e.g., add) into the immediate form (e.g., addi) if the second argument is constant.

```plaintext
abs Rdest, Rsrс
Add absolute Value
Put the absolute value of the integer from Rsrс in Rdest.
```

```plaintext
add Rdest, Rsrс1, Src2
Addition (with overflow)
Put the sum of the integers from register Rsrс1 and Src2 (or Imm) into register Rdest. Same for sub.
```

**Integers Arithmetic II**

```plaintext
move Rdest, Rsrс
Move
Move the contents of Rsrс to Rdest.
```

```plaintext
neg Rdest, Rsrс
Negate Value (with overflow)
Put the negative of the integer from register Rsrс into register Rdest.
```

```plaintext
div Rsrс1, Rsrс2
Divide (with overflow)
Divide the contents of the two registers. Leave the quotient in register lo and the remainder in register hi.
```
Integer Arithmetic III

\texttt{mult Rsrc1, Rsrc2} \quad \text{Multiply}
Multiply the contents of the two registers. Leave the low-order word of the product in register \texttt{lo} and the high-word in register \texttt{hi}.

\texttt{div Rdest, Rsrc1, Src2} \quad \text{Divide (with overflow)} \dagger
Put the quotient of the integers from register \texttt{Rsrc1} and \texttt{Src2} into register \texttt{Rdest}.

\texttt{mulo Rdest, Rsrc1, Src2} \quad \text{Multiply (with overflow)} \dagger
Put the product of the integers from register \texttt{Rsrc1} and \texttt{Src2} into register \texttt{Rdest}.

Shifts & Rotations I

\texttt{rol Rdest, Rsrc1, Src2} \quad \text{Rotate Left} \dagger
Rotate the contents of register \texttt{Rsrc1} left (right) by the distance indicated by \texttt{Src2} and put the result in register \texttt{Rdest}.

\texttt{ror Rdest, Rsrc1, Src2} \quad \text{Rotate Right} \dagger
Shift the contents of register \texttt{Rsrc1} left (right) by the distance indicated by \texttt{Src2} (\texttt{Rsrc2}) and put the result in register \texttt{Rdest}.

Logical Operations I

\texttt{and Rdest, Rsrc1, Src2} \quad \text{AND}
Put the logical AND of the integers from register \texttt{Rsrc1} and \texttt{Src2} (or \texttt{Imm}) into register \texttt{Rdest}. Same for \texttt{or}, \texttt{xor}, \texttt{nor}.

\texttt{andi Rdest, Rsrc1, Imm} \quad \text{AND Immediate}

\texttt{not Rdest, Rsrc} \quad \text{NOT} \dagger
Put the bitwise logical negation of the integer from register \texttt{Rsrc} into register \texttt{Rdest}.

Shifts & Rotations I

\texttt{sl} \texttt{l} \texttt{Rdest, Rsrc1, Src2} \quad \text{Shift Left Logical}
Shift the contents of register \texttt{Rsrc1} left (right) by the distance indicated by \texttt{Src2} (\texttt{Rsrc2}) and put the result in register \texttt{Rdest}.

\texttt{sra} \texttt{Rdest, Rsrc1, Src2} \quad \text{Shift Right Arithmetic}
Shift the contents of register \texttt{Rsrc1} left (right) by the distance indicated by \texttt{Src2} and put the result in register \texttt{Rdest}.

\texttt{srl} \texttt{Rdest, Rsrc1, Src2} \quad \text{Shift Right Logical}
Shift the contents of register \texttt{Rsrc1} left (right) by the distance indicated by \texttt{Src2} and put the result in register \texttt{Rdest}.

\texttt{li} \texttt{Rdest, imm} \quad \text{Load Immediate} \dagger
Move the immediate \texttt{imm} into register \texttt{Rdest}.

\texttt{lui} \texttt{Rdest, imm} \quad \text{Load Upper Immediate}
Load the lower halfword of the immediate \texttt{imm} into the upper halfword of register \texttt{Rdest}. The lower bits of the register are set to 0.
Comparison Instructions I

- Branch instructions use a signed 16-bit offset field; hence they can jump $2^{15} - 1$ instructions forward or $2^{15}$ instructions backwards.

- The Jump instruction contains a 26 bit address field.

seq Rdest, Rsrcl, Src2

Set Equal †
Set register Rdest to 1 if register Rsrcl equals Src2 and to be 0 otherwise. Same for sge, sgt, sle, slt, sne.

b label

Branch instruction †
Unconditionally branch to the instruction at the label.

Comparison Instructions II

bczf label

Branch Coprocessor z False
Conditionally branch to the instruction at the label if coprocessor z’s condition flag is true (false).

beq Rsrcl, Src2, label

Branch on Equal
Conditionally branch to the instruction at the label if the contents of register Rsrcl equals Src2. Same for bge, bgt, ble, blt, bne.

beqz Rsrcl, label

Branch on Equal Zero †
Conditionally branch to the instruction at the label if the contents of Rsrcl equals 0. Same for bgez, bgtz, blez, bltz, bnez.

Comparison Instructions III

j label

Jump
Unconditionally jump to the instruction at the label.

jal label

Jump and Link
Unconditionally jump to the instruction at the label.

jalr Rsrcl

Jump and Link Register
Unconditionally jump to the instruction at the label or whose address is in register Rsrcl. Save the address of the next instruction in register 31.

jr Rsrcl

Jump Register
Unconditionally jump to the instruction whose address is in register Rsrcl.

Load & Store Instructions

la Rdest, address

Load Address †
Load computed address, not the contents of the location, into register Rdest.

lb Rdest, address

Load Byte
Load the byte at address into register Rdest. The byte is sign-extended. Same for ld (Load Double-Word [64 bits]), lh (Load Half-Word [16 bits]), and (Load Word [32 bits]).

sb Rsrcl, address

Store Byte
Store the low byte from register Rsrcl at address. Same for sd (64 bits), sh (16 bits), and sw (32 bits).
Floating Point Instructions

- The MIPS has a floating point coprocessor that operates on single precision (32-bit) and double precision (64-bit) floating point numbers.
- This coprocessor has its own registers, numbered $f0–f31$.
- Two 32-bit registers are required to hold doubles. Floating point operations only use even-numbered registers—including instructions that operate on singles.
- Values are moved in or out of these registers a word (32-bits) at a time by lwc1, swc1, mtc1, and mfc1 l.s, l.d, s.s, and s.d pseudoinstructions.
- The flag set by floating point comparison operations is read by the CPU with its bc1t and bc1f instructions.

Floating Point Instructions I

- \texttt{abs.s FRdest, FRsrc} \quad Floating Point Absolute Value Single
  Compute the absolute value of the floating point double (single) in register FRsrc and put it in register FRdest.
- \texttt{neg.s FRdest, FRsrc} \quad Negate Single
  Negate the floating point single in register FRsrc and put it in register FRdest.
- \texttt{add.s FRdest, FRsrc1, FRsrc2} \quad Floating Point Addition Single
  Compute the sum of the floating point singles in registers FRsrc1 and FRsrc2 and put it in register FRdest. Same for \texttt{div.s}, \texttt{mul.s}, \texttt{sub.s}.

Floating Point Instructions II

- \texttt{c.eq.s FRsrc1, FRsrc2} \quad Compare Equal Single
  Compare the floating point single in register FRsrc1 against the one in FRsrc2 and set the floating point condition flag true if they are equal. Same for \texttt{c.le.s}.
- \texttt{cvt.s.w FRdest, FRsrc} \quad Convert Integer to Single
  Convert the integer in register FRsrc to a single precision number and put it in register FRdest.
- \texttt{cvt.w.s FRdest, FRsrc} \quad Convert Single to Integer
  Convert the single precision floating point number in register FRsrc to an integer and put it in register FRdest.

Floating Point Instructions III

- \texttt{mov.s FRdest, FRsrc} \quad Move Floating Point Single
  Move the floating float single from register FRsrc to register FRdest.
- \texttt{l.s FRdest, address} \quad Load Floating Point Single
  Load the floating float single at \texttt{address} into register FRdest.
- \texttt{s.s FRdest, address} \quad Store Floating Point Single
  Store the floating point single in register FRdest at \texttt{address}. 

**Procedure Call**

- The frame pointer points just below the last argument passed on the stack. The stack pointer points to the first word after the frame.
- A stack frame consists of the memory between the frame pointer ($fp$), and the stack pointer ($sp$).
- As typical of Unix systems, the stack grows down.

**Procedure Call—At the call site**

1. Pass the arguments. The first four arguments are passed in registers $a0$–$a3$. The remaining arguments are pushed on the stack.
2. Save the caller-saved registers. This includes registers $t0$–$t9$, if they contain live values.
3. Execute a jal instruction.

**Memory Layout**

- At the bottom of the user address space (0x400000) is the text segment (instructions).
- The program stack (0x7fffffff) grows down, towards the data segment.

**Memory Layout...**

- Above the text segment is the data segment (0x10000000). The static data portion contains objects whose size and address are known to the compiler and linker. Dynamic data is allocated by malloc through the sbrk system call.
Procedure Call—At the called routine

1. Build the stack frame by subtracting the frame size from the stack pointer.
2. Save the callee-saved registers in the frame. $fp$ is always saved. $ra$ needs to be saved if the routine itself makes calls. $s0$–$s7$ (if used by the callee) need to be saved.
3. Establish the frame pointer by adding the stack frame size to the address in $sp$.

System Calls

- You can communicate with the OS through the system call (syscall) instruction.
- Load the system call code into register $v0$ and the arguments into registers $a0$..$a3$.
- System calls return values in register $v0$.

<table>
<thead>
<tr>
<th>Service</th>
<th>Code</th>
<th>Args</th>
<th>Res</th>
</tr>
</thead>
<tbody>
<tr>
<td>print_int</td>
<td>1</td>
<td>$a0$</td>
<td></td>
</tr>
<tr>
<td>print_float</td>
<td>2</td>
<td>$f12$</td>
<td></td>
</tr>
<tr>
<td>print_string</td>
<td>4</td>
<td>$a0$</td>
<td>$v0$</td>
</tr>
<tr>
<td>read_int</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>read_float</td>
<td>6</td>
<td></td>
<td>$f0$</td>
</tr>
<tr>
<td>exit</td>
<td>10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Procedure Call—Returning from the call

1. Place the returned value into $v0$ (if a function).
2. Restore any callee-saved registers that were saved upon entry (including the frame pointer $fp$).
3. Pop the stack frame by adding the stack frame size to $sp$.
4. Return by jumping to the address in register $ra$.
**Assembler Directives**

- `.align n` Align the next datum on a $2^n$ byte boundary.
- `.asciiz str` Store the string in memory and null-terminate it.
- `.byte b1, ..., bn` Store the $n$ values in successive bytes of memory.
- `.data` The following data items should be stored in the data segment.

- `.float f1, ..., fn` Store the $n$ floating point single precision numbers in successive memory locations.
- `.space n` Allocate $n$ bytes of space in the data segment.
- `.text` The next items are put in the user text segment.
- `.word w1, ..., wn` Store the $n$ 32-bit quantities in successive memory words.
Example 2 – Branching

```c
main () {
    int i, p=1;
    for(i=1; i<=10; i++) { p = p * i; printf(p); }
}
```

Assembly and Execution:

(spim) reinitialize
(spim) load "fac.sp"
(spim) run

1
2
6
24
120
720
5040...

---

Example – System Calls

```c
.data
_a: .word 10
_b: .word 20
_c: .word 30
res: .asciiz "The result is: 
_nl: .asciiz "\n"

.text
.globl main
main:
```

```assembly
li $v0, 4 # syscall 4
la $a0, res # (print_str)
syscall
li $v0, 1 # syscall 1
lw $a0, _a # (print_int)
syscall
li $v0, 4 # syscall 4
la $a0, _nl # (print_str)
syscall
li $v0, 10 # syscall 10	syscall # (exit)
```

---

Example – System Calls...

```assembly
.data
_p: .word 0
_i: .word 0
_nl: .asciiz "\n"
.text
.globl main
main:
# Initialize p and i to 1
li $14, 1
sw $14, _p
sw $14, _i
# Beginning of loop: Load p and i. p = p * i.
$32: lw $24, _p
lw $25, _i
mul $8, $24, $25
sw $8, _p
```

---

Example – Branching

```assembly
main:lw $2, _b # load b
lw $3, _c # load c
addu $2, $2, $3 # add
sw $2, _a # store in a
li $v0, 4 # syscall 4
la $a0, res # (print_str)
syscall
li $v0, 1 # syscall 1
lw $a0, _a # (print_int)
syscall
li $v0, 4 # syscall 4
la $a0, _nl # (print_str)
syscall
li $v0, 10 # syscall 10	syscall # (exit)
```
Example 4 – Large Ints

```mips
main () {
    int b; b = b + 1234567;
}
```

The Virtual MIPS Code:
```
.globl main
.a: .word 0
.b: .word 0
main:
    lw $2,._b
    li $3,0x0012d687 # 1234567
    addu $2,$2,$3
    sw $2,._b
```

li $v0, 10
syscall

Example 4 – Large Ints...

- Some of the assembly instructions we’ve seen are actually virtual instructions – they are not actually implemented by the hardware. Instead, the assembler translates them into “real” instructions.

- There is no instruction to load a 32-bit address or literal integer into a register (why?). Instead, the assembler translates `li $3,0x0012d687` into two instructions: one loads the upper 16 bits, one the lower 16 bits.

- Similarly, each branch/jump/call instruction has a delayed branch slot. This is an instruction that in the program looks like it comes after the jump, but which actually gets executed before the jump is made. The assembler automatically fills these delay slots.

Example – Procedure Call
```
void P (a,b) int a,b; {int c=a+b;}
main () {P(5,6);}
```

The MIPS Code:
```
.text
P:
    subu $sp, 8
    addu $14, $4, $5
    sw $14, 4($sp)
    addu $sp, 8
    j $31
main:
    subu $sp, 32
    sw $31, 28($sp)
    li $4, 5
    li $5, 6
    jal P
```
Example 4 (C) – The Real MIPS Code

# Load the value of b. First the upper 16 bits of 
# b’s address is loaded, then the lower 16 bits are 
# added in, and b’s value is loaded into $2.  
lui $1, 64 ; lw $2, _b  
lw $2, 36($1)  

# Load 1234567 into $3. First the upper 16 bits, then the 
# lower.  
lui $1, 18 ; li $3, 0x0012d687 # 1234567  
ori $3, $1, -10617  

# Perform the add. Then load the address of b and store $2.  
addu $2, $2, $3 ; addu $2, $2, $3  
lui $1, 64 ; sw $2, _b  
sw $2, 36($1)  

Confused Student Email I

Dr. collberg: I am quiet confused about: `.*align 2’ usage. In 
manual it says `.*align n aligns the next datum on a 2^n byte 
boundary’. What does it really use for? (i.e. why we need it?) when 
we should use it?

On the Mips, every variable has to be aligned (start on) an 
address that is a multiple of the variable’s size. So, a 4 byte 
variable has to be allocated to an address that is a multiple of 4, 
e.g. 4, 8, 12, 16, 20, ..., 

Confused Student Email II

When SPIM was executing the assembly language instruction 
mul.s $f2, $f0, $f1 it give the message Bit 0 in FP reg 
spec and stopped execution. Could you tell me what this message 
means and why it is coming up? Thank you.

You can only use even-numbered floating-point registers (i.e. $f0 
$f2 $f4...) for single-precision numbers.

Can odd-numbered floating-point registers be used for some other 
kind of number? If so, what kind? If not, what is the point in 
having them?

Double precision (64-bit) FP numbers are stored in registers 
$f0+$f1, $f2-$f3, etc.

Confused Student Email V

What is the instruction for branch

    if a > b then ---
when a and b are floating point number? When I try "bczt label " 
or "bc1t label" I got spim parse error

spim: (parser) syntax error on line 21 
bczt L10 

Attempt to execute non-instruction at ...

You use ‘c.le.s’ etc to compare two floating point numbers. This 
sets the floating point coprocessor condition flag. See page 19 of 
the spim manual. Then use 'bc1t label' to jump (page 15 of the 
manual).
Confused Student Email VI

How could I load the value of Char from memory so I can do "<" operation? Because if I generate

```masm
.data
address: .asciiz "a"
lw $1, address
```

I got a error message from spim:

```plaintext
----Unaligned address in inst/data fetch:
Exception 4 [Unaligned address in inst/data fetch]
occurred and ignored---
And I have no idea about that.
```

---

Confused Student Email VIII

How do I load a floating point literal?

```masm
PROGRAM P1;
VAR F1 : REAL;
BEGIN
  F1 := 4.14;
END.
.data
_F1: .float 0.0
.data
main:
  la $8, _F1
.data
_F4.14: .float 4.14
.text
l.s $f2, _F4.14
swc1 $f2, ($8)
```

---

Homework

- Try out spim and xspim.
- Write some small MIPS programs and execute them using spim.
- Single step through a program. Try both the virtual and the bare (use spim -bare) machine.

---

Confused Student Email VI...

CHARs are one byte long. Use 'lb' and 'sb' (load and store byte) when you’re working on CHARs. 'lw' loads a word (4 bytes). On the Mips words have to be aligned on a 4 byte boundary, which is why Spim complains.