

# MIPS Computer Architecture

## MIPS registers

- 32 registers: numbered from 0 up to 31
- each has a name and intended use

Register Name	Number	Usage
zero	0	Constant 0
at	1	Reserved for assembler
v0	2	Expression evaluation and
v1	3	results of a function
a0	4	Argument 1
a1	5	Argument 2
a2	6	Argument 3
a3	7	Argument 4
t0	8	Temporary (not preserved across call)
t1	9	Temporary (not preserved across call)
t2	10	Temporary (not preserved across call)
t3	11	Temporary (not preserved across call)

Register Name	Number	Usage
t4	12	Temporary (not preserved across call)
t5	13	Temporary (not preserved across call)
t6	14	Temporary (not preserved across call)
t7	15	Temporary (not preserved across call)
s0	16	Saved temporary (preserved across call)
s1	17	Saved temporary (preserved across call)
s2	18	Saved temporary (preserved across call)
s3	19	Saved temporary (preserved across call)
s4	20	Saved temporary (preserved across call)
s5	21	Saved temporary (preserved across call)
s6	22	Saved temporary (preserved across call)
s7	23	Saved temporary (preserved across call)
t8	24	Temporary (not preserved across call)
t9	25	Temporary (not preserved across call)
k0	26	Reserved for OS kernel
k1	27	Reserved for OS kernel
gp	28	Pointer to global area
sp	29	Stack pointer
fp	30	Frame pointer
ra	31	Return address (used by function call)

A RISC (Reduced Instruction Set Computer) machine

- memory access only through load and store
- computations only on values in registers
- single address mode for bare machine: imm (register). Address is  $\text{imm} + [\text{register}]$ .

Address modes in Assembly code

- six addressing modes

Format	Address Computation
(register)	contents of register
imm	immediate
imm (register)	immediate + contents of register
symbol	address of symbol
symbol $\pm$ imm	address of symbol $\pm$ immediate
symbol $\pm$ imm (register)	address of symbol $\pm$ (immediate + contents of register)

### Some Instructions from the Instruction Set

- mnemonics:
  - *Rdest*: destination register
  - *Rsrc1*: source register for the first operand
  - *Imm*: immediate value
  - *Src2t*: source for the second operand, can be either a register or an immediate value
  - *Label*: label for an address

- Load and Store Instructions

`la Rdest, address` *Load Address* †

Load computed *address*, not the contents of the location, into register `Rdest`.

`lw Rdest, address` *Load Word*

Load the 32-bit quantity (word) at *address* into register `Rdest`.

`sw Rsource, address` *Store Word*

Store the word from register `Rsource` at *address*.

- Exception and Trap Instructions

`rfe`

*Return From Exception*

Restore the Status register.

`syscall`

*System Call*

Register `$v0` contains the number of the system call.

`break n`

*Break*

Cause exception *n*. Exception 1 is reserved for the debugger.

`nop`

*No operation*

Do nothing.

- Constant-Manipulating Instructions

`li Rdest, imm`

*Load Immediate* †

Move the immediate into register `Rdest`.

- Arithmetic and Logical Instructions

`Src2` can either be a register or an immediate value (integer).

`abs Rdest, Rsource`

*Absolute Value* †

Put the absolute value of the integer from register `Rsource` in register `Rdest`.

<code>add Rdest, Rsrc1, Src2</code>	<i>Addition (with overflow)</i>
<code>addi Rdest, Rsrc1, Imm</code>	<i>Addition Immediate (with overflow)</i>
<code>addu Rdest, Rsrc1, Src2</code>	<i>Addition (without overflow)</i>
<code>addiu Rdest, Rsrc1, Imm</code>	<i>Addition Immediate (without overflow)</i>

Put the sum of the integers from register `Rsrc1` and `Src2` (or `Imm`) into register `Rdest`.

<code>and Rdest, Rsrc1, Src2</code>	<i>AND</i>
<code>andi Rdest, Rsrc1, Imm</code>	<i>AND Immediate</i>

Put the logical AND of the integers from register `Rsrc1` and `Src2` (or `Imm`) into register `Rdest`.

<code>neg Rdest, Rsource</code>	<i>Negate Value (with overflow)</i> †
<code>negu Rdest, Rsource</code>	<i>Negate Value (without overflow)</i> †

Put the negative of the integer from register `Rsource` into register `Rdest`.

- Comparison Instructions

`seq Rdest, Rsrc1, Src2` *Set Equal* †

Set register `Rdest` to 1 if register `Rsrc1` equals `Src2` and to be 0 otherwise.

- Branch and Jump Instructions

`b label` *Branch instruction* †

Unconditionally branch to the instruction at the label.

`beq Rsrc1, Src2, label` *Branch on Equal*

Conditionally branch to the instruction at the label if the contents of register `Rsrc1` equals `Src2`.

`j label` *Jump*

Unconditionally jump to the instruction at the label.

`jal label` *Jump and Link*

`jalr Rsource` *Jump and Link Register*

Unconditionally jump to the instruction at the label or whose address is in register `Rsource`. Save the address of the next instruction in register 31.

`jr Rsource` *Jump Register*

Unconditionally jump to the instruction whose address is in register `Rsource`.

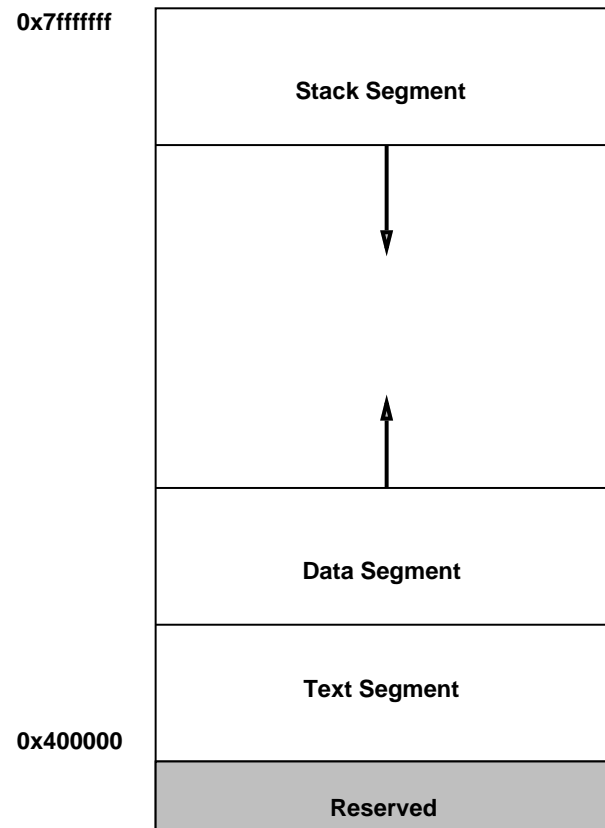
- Data Movement Instructions

`move Rdest, Rsource` *Move*

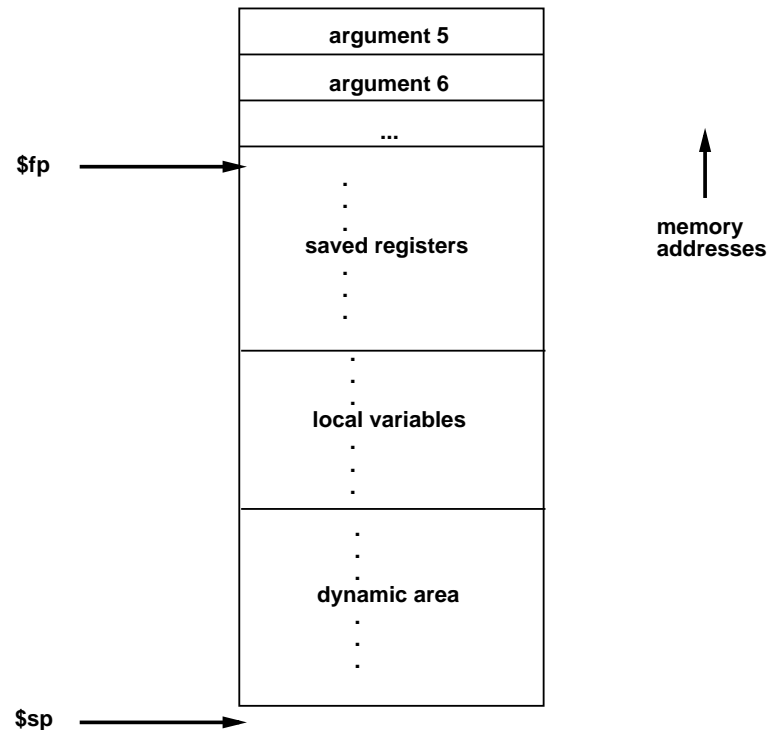
Move the contents of `Rsource` to `Rdest`.

### Memory Usage

- Address Space



- Layout of Stack Frame (gcc-UNIX)
  - Stack frame is the area between the locations pointed by FP and SP
  - FP points to the word immediately after the last argument passed
  - SP points to the first word of free space of the stack



- Calling Convention (UNIX)
  - Steps for Calling procedure
    1. pass arguments (first four in `a0 – a3`, the remaining into the stack)
    2. save caller-save registers such as `t0 – t7` if necessary
    3. call `jal`
  - Steps of called procedure
    1. `sp = sp - frame size`
    2. save callee-save registers: `fp`, `ra` (if the callee makes another call), and possible `s0 – s7`
    3. `fp = sp + frame size`
  - Steps for callee to return
    1. put return value in `v0`
    2. restore any saved register upon entry including old `fp`
    3. `sp = sp - frame size`

4. execute jal ra