

MIPS Architecture and Assembly Language Overview

Adapted from: [http://edge.mcs.dre.gel.edu/GICL/people/sevy/architecture/MIPSRef\(SPIM\).html](http://edge.mcs.dre.gel.edu/GICL/people/sevy/architecture/MIPSRef(SPIM).html)

[\[Register Description\]](#) [\[I/O Description\]](#)

Data Types and Literals

Data types:

- Instructions are all 32 bits
- byte(8 bits), halfword (2 bytes), word (4 bytes)
- a character requires 1 byte of storage
- an integer requires 1 word (4 bytes) of storage

Literals:

- numbers entered as is. e.g. 4
- characters enclosed in single quotes. e.g. 'b'
- strings enclosed in double quotes. e.g. "A string"

Registers

- 32 general-purpose registers
- register preceded by \$ in assembly language instruction
- two formats for addressing:
 - using register number e.g. \$0 through \$31
 - using equivalent names e.g. \$t1, \$sp
- special registers Lo and Hi used to store result of multiplication and division
 - not directly addressable; contents accessed with special instruction mfhi ("move from Hi") and mflo ("move from Lo")
- stack grows from high memory to low memory

This is from Figure 9.9 in the Goodman&Miller text

Register Number	Alternative Name	Description
0	zero	the value 0
1	\$at	(assembler temporary) reserved by the assembler
2-3	\$v0 - \$v1	(values) from expression evaluation and function results
4-7	\$a0 - \$a3	(arguments) First four parameters for subroutine. Not preserved across procedure calls
8-15	\$t0 - \$t7	(temporaries) Caller saved if needed. Subroutines can use w/out saving. Not preserved across procedure calls
16-23	\$s0 - \$s7	(saved values) - Callee saved. A subroutine using one of these must save original and restore it before exiting. Preserved across procedure calls
24-25	\$t8 - \$t9	(temporaries) Caller saved if needed. Subroutines can use w/out saving. These are in addition to \$t0 - \$t7 above. Not preserved across procedure calls.
26-27	\$k0 - \$k1	reserved for use by the interrupt/trap handler
28	\$gp	global pointer. Points to the middle of the 64K block of memory in the static data segment.
29	\$sp	stack pointer Points to last location on the stack.
30	\$s8/\$fp	saved value / frame pointer Preserved across procedure calls
31	\$ra	return address

See also Britton section 1.9, Sweetman section 2.21, Larus Appendix section A.6

Program Structure

- just plain text file with data declarations, program code (name of file should end in suffix .s to be used with SPIM simulator)
- data declaration section followed by program code section

Data Declarations

- placed in section of program identified with assembler directive **.data**
- declares variable names used in program; storage allocated in main memory (RAM)

Code

- placed in section of text identified with assembler directive **.text**
- contains program code (instructions)
- starting point for code e.g. execution given label **main**:
- ending point of main code should use exit system call (see below under System Calls)

Comments

- anything following # on a line
This stuff would be considered a comment
- Template for a MIPS assembly language program:

```
# Comment giving name of program and description of function
# Template.s
# Bare-bones outline of MIPS assembly language program

        .data        # variable declarations follow this line
        # ...

        .text        # instructions follow this line

main:   # indicates start of code (first instruction to execute)
        # ...

# End of program, leave a blank line afterwards to make SPIM happy
```

Data Declarations

format for declarations:

```
name:   storage_type   value(s)
```

- create storage for variable of specified type with given name and specified value
- value(s) usually gives initial value(s); for storage type `.space`, gives number of spaces to be allocated

Note: labels always followed by colon (:)

example

```
var1:    .word    3      # create a single integer variable with initial value 3
array1:  .byte    'a','b' # create a 2-element character array with elements initialized
                        # to a and b
array2:  .space   40     # allocate 40 consecutive bytes, with storage uninitialized
                        # could be used as a 40-element character array, or a
                        # 10-element integer array; a comment should indicate which!
```

Load / Store Instructions

- RAM access only allowed with load and store instructions
- all other instructions use register operands

load:

```
lw      register_destination, RAM_source

#copy word (4 bytes) at source RAM location to destination register.

lb      register_destination, RAM_source

#copy byte at source RAM location to low-order byte of destination register,
# and sign-e.g. tend to higher-order bytes
```

store word:

```
sw      register_source, RAM_destination

#store word in source register into RAM destination

sb      register_source, RAM_destination

#store byte (low-order) in source register into RAM destination
```

load immediate:

```
li      register_destination, value

#load immediate value into destination register
```

example:

```
.data
var1:   .word    23      # declare storage for var1; initial value is 23
```

```

        .text
__start:
    lw      $t0, var1      # load contents of RAM location into register $t0: $t0 = var1
    li      $t1, 5         # $t1 = 5 ("load immediate")
    sw      $t1, var1      # store contents of register $t1 into RAM: var1 = $t1
    done

```

Indirect and Based Addressing

- Used only with load and store instructions

load address:

```
la      $t0, var1
```

- copy RAM address of var1 (presumably a label defined in the program) into register \$t0

indirect addressing:

```
lw      $t2, ($t0)
```

- load word at RAM address contained in \$t0 into \$t2

```
sw      $t2, ($t0)
```

- store word in register \$t2 into RAM at address contained in \$t0

based or indexed addressing:

```
lw      $t2, 4($t0)
```

- load word at RAM address (\$t0+4) into register \$t2
- "4" gives offset from address in register \$t0

```
sw      $t2, -12($t0)
```

- store word in register \$t2 into RAM at address (\$t0 - 12)
- negative offsets are fine

Note: based addressing is especially useful for:

- arrays; access elements as offset from base address
- stacks; easy to access elements at offset from stack pointer or frame pointer

example

```

array1:    .data
           .space 12      # declare 12 bytes of storage to hold array of 3 integers
           .text
__start:   la      $t0, array1      # load base address of array into register $t0
           li      $t1, 5          # $t1 = 5 ("load immediate")
           sw      $t1, ($t0)       # first array element set to 5; indirect addressing
           li      $t1, 13         # $t1 = 13
           sw      $t1, 4($t0)      # second array element set to 13
           li      $t1, -7         # $t1 = -7
           sw      $t1, 8($t0)      # third array element set to -7
           done

```

Arithmetic Instructions

- most use 3 operands
- all operands are registers; no RAM or indirect addressing
- operand size is word (4 bytes)

```

add      $t0,$t1,$t2      # $t0 = $t1 + $t2; add as signed (2's complement) integers
sub      $t2,$t3,$t4      # $t2 = $t3 - $t4
addi     $t2,$t3, 5        # $t2 = $t3 + 5; "add immediate" (no sub immediate)
addu     $t1,$t6,$t7       # $t1 = $t6 + $t7; add as unsigned integers
subu     $t1,$t6,$t7       # $t1 = $t6 - $t7; subtract as unsigned integers

mult     $t3,$t4           # multiply 32-bit quantities in $t3 and $t4, and store 64-bit
                           # result in special registers Lo and Hi: (Hi,Lo) = $t3 * $t4
div      $t5,$t6           # Lo = $t5 / $t6 (integer quotient)
                           # Hi = $t5 mod $t6 (remainder)
mfhi     $t0               # move quantity in special register Hi to $t0: $t0 = Hi
mflo     $t1               # move quantity in special register Lo to $t1: $t1 = Lo
                           # used to get at result of product or quotient

move     $t2,$t3 # $t2 = $t3

```

Control Structures

Branches

- comparison for conditional branches is built into instruction

```

b      target      # unconditional branch to program label target
beq    $t0,$t1,target # branch to target if $t0 = $t1
blt    $t0,$t1,target # branch to target if $t0 < $t1
ble    $t0,$t1,target # branch to target if $t0 <= $t1
bgt    $t0,$t1,target # branch to target if $t0 > $t1
bge    $t0,$t1,target # branch to target if $t0 >= $t1
bne    $t0,$t1,target # branch to target if $t0 <> $t1

```

Jumps

```

j      target      # unconditional jump to program label target
jr     $t3          # jump to address contained in $t3 ("jump register")

```

Subroutine Calls

subroutine call: "jump and link" instruction

```
jal    sub_label    # "jump and link"
```

- copy program counter (return address) to register \$ra (return address register)
- jump to program statement at sub_label

subroutine return: "jump register" instruction

```
jr     $ra          # "jump register"
```

- jump to return address in \$ra (stored by jal instruction)

Note: return address stored in register \$ra; if subroutine will call other subroutines, or is recursive, return address should be copied from \$ra onto stack to preserve it, since jal always places return address in this register and hence will overwrite previous value

System Calls and I/O (SPIM Simulator)

- used to read or print values or strings from input/output window, and indicate program end
- use **syscall** operating system routine call
- first supply appropriate values in registers \$v0 and \$a0-\$a1
- result value (if any) returned in register \$v0

The following table lists the possible **syscall** services.

Service	Code in \$v0	Arguments	Results
print_int	1	\$a0 = integer to be printed	
print_float	2	\$f12 = float to be printed	
print_double	3	\$f12 = double to be printed	
print_string	4	\$a0 = address of string in memory	
read_int	5		integer returned in \$v0
read_float	6		float returned in \$v0
read_double	7		double returned in \$v0
read_string	8	\$a0 = memory address of string input buffer \$a1 = length of string buffer (n)	
sbrk	9	\$a0 = amount	address in \$v0
exit	10		

- The print_string service expects the address to start a null-terminated character string. The directive **.asciiz** creates a null-terminated character string.
- The read_int, read_float and read_double services read an entire line of input up to and including the newline character.
- The read_string service has the same semantics as the UNIX library routine fgets.
 - It reads up to n-1 characters into a buffer and terminates the string with a null character.
 - If fewer than n-1 characters are in the current line, it reads up to and including the newline and terminates the string with a null character.
- The sbrk service returns the address to a block of memory containing n additional bytes. This would be used for dynamic memory allocation.
- The exit service stops a program from running.

e.g. Print out integer value contained in register \$t2

```

li      $v0, 1          # load appropriate system call code into register $v0;
                        # code for printing integer is 1
move    $a0, $t2        # move integer to be printed into $a0: $a0 = $t2
syscall                          # call operating system to perform operation

```

e.g. Read integer value, store in RAM location with label int_value (presumably declared in data section)

```
li      $v0, 5          # load appropriate system call code into register $v0;
```

```
        syscall      # code for reading integer is 5
        sw          $v0, int_value    # call operating system to perform operation
                                         # value read from keyboard returned in register $v0;
                                         # store this in desired location
```

e.g. Print out string (useful for prompts)

```
        .data
string1 .asciiz "Print this.\n"      # declaration for string variable,
                                         # .asciiz directive makes string null terminated

        .text
main:   li          $v0, 4            # load appropriate system call code into register $v0;
                                         # code for printing string is 4
        la          $a0, string1     # load address of string to be printed into $a0
        syscall      # call operating system to perform print operation
```

e.g. To indicate end of program, use `exit` system call; thus last lines of program should be:

```
        li          $v0, 10          # system call code for exit = 10
        syscall      # call operating sys
```