
MIPS Assembly Language

Chapter 15
S. Dandamudi

Outline

- MIPS architecture
 - * Registers
 - * Addressing modes
- MIPS instruction set
 - * Instruction format
 - * Data transfer instructions
 - * Arithmetic instructions
 - * Logical/shift/rotate/compare instructions
 - * Branch and jump instructions
- SPIM system calls
- SPIM assembler directive
- Illustrative examples
- Procedures
- Stack implementation
- Illustrative examples

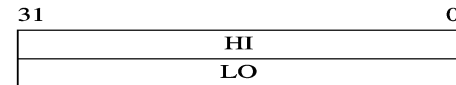
MIPS Processor Architecture

- MIPS follows RISC principles much more closely than PowerPC and Itanium
 - * Based on the load/store architecture
- Registers
 - * 32-general purpose registers (\$0 – \$31)
 - » \$0 – hardwired to zero
 - » \$31 – used to store return address
 - * Program counter (PC)
 - » Like IP in Pentium
 - * Two special-purpose registers (HI and LO)
 - » Used in multiply and divide instructions

MIPS Processor Architecture (cont'd)

	31	0
zero	0	
at	1	
v0	2	
v1	3	
a0	4	
a1	5	
a2	6	
a3	7	
t0	8	
t1	9	
t2	10	
t3	11	
t4	12	
t5	13	
t6	14	
t7	15	
s0	16	
s1	17	
s2	18	
s3	19	
s4	20	
s5	21	
s6	22	
s7	23	
t8	24	
t9	25	
k0	26	
k1	27	
gp	28	
sp	29	
fp	30	
ra	31	

General-purpose registers



Multiply and divide registers



Program counter

MIPS Processor Architecture (cont'd)

MIPS registers and their conventional usage

Register name	Number	Intended usage
zero	0	Constant 0
at	1	Reserved for assembler
v0, v1	2, 3	Results of a procedure
a0, a1, a2, a3	4–7	Arguments 1–4
t0–t7	8–15	Temporary (not preserved across call)
s0–s7	16–23	Saved temporary (preserved across call)
t8, t9	24, 25	Temporary (not preserved across call)
k0, k1	26, 27	Reserved for OS kernel
gp	28	Pointer to global area
sp	29	Stack pointer
fp	30	Frame pointer (if needed); otherwise, a saved register \$s8
ra	31	Return address (used by a procedure call)

MIPS Processor Architecture (cont'd)

MIPS addressing modes

- * Bare machine supports only a single addressing mode

disp (Rx)

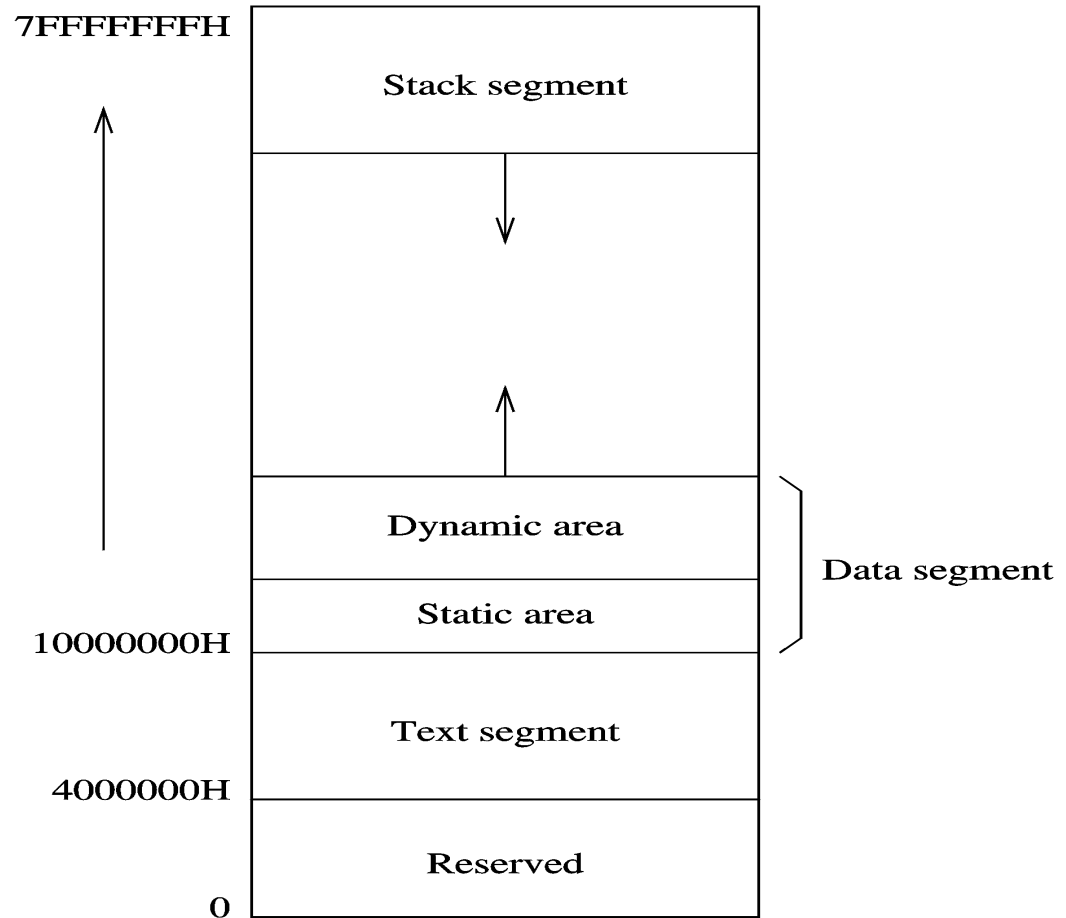
- * Virtual machine provides several additional addressing modes

Format	Address computed as
(Rx)	Contents of register Rx
imm	Immediate value imm
imm(Rx)	imm + contents of Rx
symbol	Address of symbol
symbol \pm imm	Address of symbol \pm imm
symbol \pm imm(Rx)	Address of symbol \pm (imm + contents of Rx)

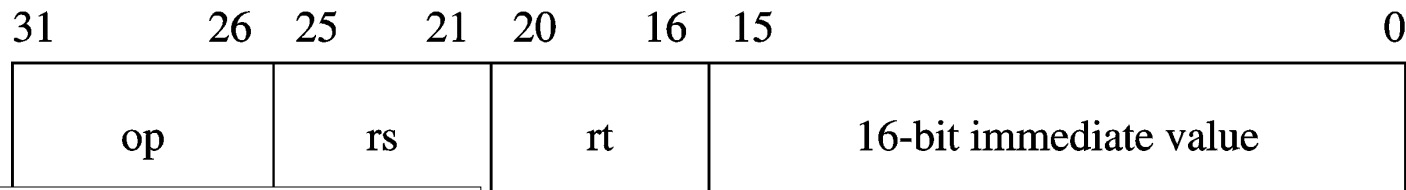
Memory Usage

Placement of segments allows sharing of unused memory by both data and stack segments

Memory addresses

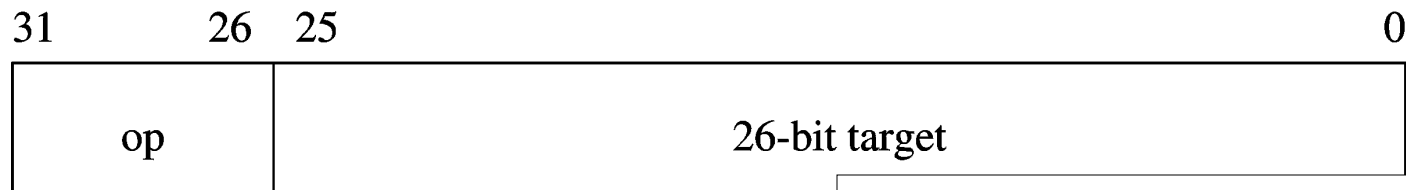


Instruction Format



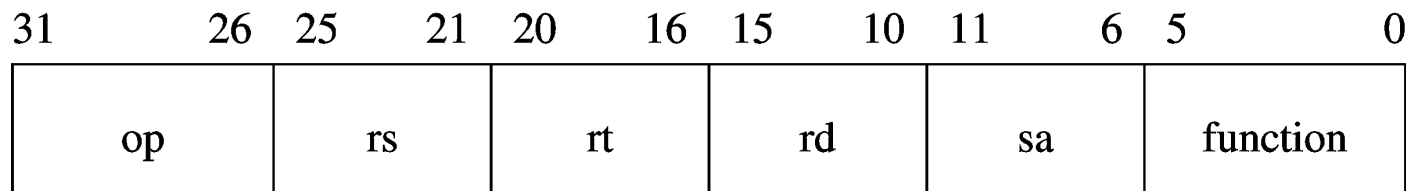
load, arithmetic/logical
with immediate operands

I-Type (Immediate)



J-Type (Jump)

Higher order bits from PC are
added to get absolute address



R-Type (Register)

MIPS Instruction Set

- Data transfer instructions
 - * Load and store instructions have similar format
 - ld Rdest, address**
 - » Moves a byte from **address** to **Rdest** as a signed number
 - Sign-extended to **Rdest**
 - » Use **ldu** for unsigned move (zero-extended)
 - * Use **lh**, **lhu**, **ld** for moving halfwords (signed/unsigned) and words
 - * Pseudoinstructions
 - la Rdest, address**
 - li Rdest, imm**
 - » Implemented as **ori Rdest, \$0, imm**

MIPS Instruction Set (cont'd)

- * Store byte

sb **Rsrc, address**

» Use **sh** and **sw** for halfwords and words

- * Pseudoinstruction

move **Rdest, Rsrc**

» Copies **Rsrc** to **Rdest**

- * Four additional data movement instructions are available

» Related to HI and LO registers

» Used with multiply and divide instructions

– Discussed later

MIPS Instruction Set (cont'd)

- Arithmetic instructions

- * Addition

add Rdest, Rsrc1, Rsrc2

- $R_{dest} \leftarrow R_{src1} + R_{src2}$
- Numbers are treated as signed integers
- Overflow: Generates overflow exception
- Use **addu** if the overflow exception is not needed

addi Rdest, Rsrc1, imm

- **imm**: 16-bit signed number

- * Pseudoinstruction

add Rdest, Rsrc1, Src2

Register or imm16



MIPS Instruction Set (cont'd)

* Subtract

sub **Rdest, Rsrc1, Rsrc2**

- $R_{dest} \leftarrow R_{src1} - R_{src2}$
- Numbers are treated as signed integers
- Overflow: Generates overflow exception
- Use **subu** if the overflow exception is not needed
- No immediate version
 - Use **addi** with negative **imm**

* Pseudoinstruction

sub **Rdest, Rsrc1, Src2**

Register or imm16



MIPS Instruction Set (cont'd)

* Pseudoinstructions

neg **Rdest, Rsrc**

- Negates **Rsrc** (changes sign)
- Implemented as

sub **Rdest, \$0, Rsrc**

abs **Rdest, Rsrc**

- Implemented as

bgez **Rsrc, skip**

sub **Rdest, \$0, Rsrc**

skip:

Constant 8
is used



MIPS Instruction Set (cont'd)

- * Multiply

- » **mult** (signed)

- » **multu** (unsigned)

mult **Rsrc1, Rsrc2**

- » 64-bit result in LO and HI registers

- » Special data move instructions for LO/HI registers

mfhi **Rdest**

mflo **Rdest**

- * Pseudoinstruction

mul **Rdest, Rsrc1, Rsrc2**

- 32-bit result in **Rdest**

- ➔ 64-bit result is not available

Register or imm



MIPS Instruction Set (cont'd)

* **mul** is implemented as

» If **Rsrc2** is a register

mult **Rsrc1, Src2**

mflo **Rdest**

» If **Rsrc2** is an immediate value (say 32)

ori **\$1, \$0, 32**

mult **\$5, \$1**

mflo **\$4**

a0 = \$4

a1 = \$5

at = \$1

MIPS Instruction Set (cont'd)

* Divide

» **div** (signed)

» **divu** (unsigned)

div **Rsrc1, Rsrc2**

» Result = **Rsrc1/Rsrc2**

» LO = quotient, HI = remainder

» Result undefined if the divisor is zero

* Pseudoinstruction

div **Rdest, Rsrc1, Src2**

– quotient in **Rdest**

rem **Rdest, Rsrc1, Src2**

– remainder in **Rdest**

Register or imm



MIPS Instruction Set (cont'd)

- Logical instructions

- * Support AND, OR, XOR, NOR

and **Rdest, Rsrc1, Rsrc2**

andi **Rdest, Rsrc1, imm16**

- * Also provides **or, ori, xor, xori, nor**

- * No **not** instruction

- » It is provided as a pseudoinstruction

not **Rdest, Rsrc**

- » Implemented as

nor **Rdest, Rsrc, \$0**

MIPS Instruction Set (cont'd)

- Shift instructions

- * Shift left logical

- sll** **Rdest, Rsrc1, count**

- » Vacated bits receive zeros
 - » Shift left logical variable

- sllv** **Rdest, Rsrc1, Rsrc2**

- » Shift count in **Rsrc2**

- * Two shift right instructions

- » Logical (**srl, srlv**)

- Vacated bits receive zeros

- » Arithmetic (**sra, srav**)

- Vacated bits receive the sign bit (sign-extended)

MIPS Instruction Set (cont'd)

- Rotate instructions
 - * These are pseudoinstructions

rol **Rdest, Rsrc1, Src2**

ror **Rdest, Rsrc1, Src2**

» Example:

ror **\$t2, \$t2, 31**

is translated as

sll **\$1, \$10, 31**

srl **\$10, \$10, 1**

or **\$10, \$10, \$1**

t2 = \$10

MIPS Instruction Set (cont'd)

- Comparison instructions

- * All are pseudoinstructions

- slt** **Rdest, Rsrc1, Rsrc2**

- » Sets **Rdest** to 1 if **Rsrc1 < Rsrc2**

- » Unsigned version: **sltu**

- » Others:

- **seq**

- **sgt, sgtu**

- **sge, sgeu**

- **sle, sleu**

- **sne**

MIPS Instruction Set (cont'd)

- Comparison instructions

» Example:

```
seq    $a0, $a1, $a2
```

is translated as

```
      beq    $6, $5, skip1  
      ori    $4, $0, 0  
      beq    $0, $0, skip2  
skip1:  
      ori    $4, $0, 1  
skip2:
```

a0 = \$4
a1 = \$5
a2 = \$6

MIPS Instruction Set (cont'd)

- Branch and Jump instructions

- * Jump instruction

j target

» Uses 26-bit absolute address

- * Branch pseudoinstruction

b target

» Uses 16-bit relative address

- * Conditional branches

beq Rsrc1, Rsrc2, target

» Jumps to **target** if **Rsrc1 = Rsrc2**

MIPS Instruction Set (cont'd)

* Other branch instructions

bne

blt, bltu

bgt, bgtu

ble, bleu

bge, bgeu

* Comparison with zero

beqz Rsrc, target

» Branches to **target** if **Rsrc = 0**

» Others

– **bnez, bltz, bgtz, blez, bgez**

» **b target** is implemented as **bgez \$0, target**

SPIM System Calls

- SPIM supports I/O through **syscall**
 - * Data types:
 - » **string, integer, float, double**
 - Service code: \$v0
 - Required arguments: \$a0 and \$a1
 - Return value: \$v0
 - * **print_string**
 - » Prints a NULL-terminated string
 - * **read_string**
 - » Takes a buffer pointer and its size *n*
 - » Reads at most *n-1* characters in NULL-terminated string
 - » Similar to **fgets**

SPIM System Calls (cont'd)

Service	System call code	Arguments	Result
print_int	1	\$a0 = integer	
print_float	2	\$f12 = oat	
print_double	3	\$f12 = double	
print_sting	4	\$a0 = string address	
read_int	5		Integer in \$v0
read_float	6		Float in \$f0
read_double	7		Double in \$f0
read_string	8	\$a0 = buffer address \$a1 = buffer size	
sbrk	9		Address in \$v0
exit	10		

SPIM System Calls (cont'd)

```
.DATA
prompt:
    .ASCIIZ    "Enter your name: "
in-name:
    .SPACE    31
    .TEXT


    . . .
la    $a0,prompt
li    $v0,4
syscall
la    $a0,in_name
li    $a1,31
li    $v0,8
syscall
```

SPIM Assembler Directives

- Segment declaration

- * Code: **.TEXT**
.TEXT <address>
- * Data: **.DATA**

Optional; if present,
segment starts at
that address



- String directives

- * **.ASCII**
 - » Not NULL-terminated
- * **.ASCIIZ**
 - » Null-terminated

Example:
ASCII “This is a very long string”
ASCII “spread over multiple
ASCIIZ “string statements.”

- Uninitialized space

.SPACE n

SPIM Assembler Directives (cont'd)

- Data directives

- * Provides four directives:

- **.HALF**, **.WORD**

- **.FLOAT**, **.DOUBLE**

- .HALF** **h1, h2, . . . , hn**

- Allocates 16-bit halfwords
 - Use **.WORD** for 32-bit words

- » Floating-point numbers

- Single-precision

- .FLOAT** **f1, f2, . . . , fn**

- Use **.DOUBLE** for double-precision

SPIM Assembler Directives (cont'd)

- Miscellaneous directives

- * Data alignment

- » Default:

- **.HALF**, **.WORD**, **.FLOAT**, **.DOUBLE** align data

- » Explicit control:

- .ALIGN** **n**

- aligns the next datum on a 2^n byte boundary

- » To turn off alignment, use

- .ALIGN** **0**

- * **.GLOBL** declares a symbol global

```
.TEXT
.GLOBL     main
main:
          . . .
```

Illustrative Examples

- Character to binary conversion
 - * `binch.asm`
- Case conversion
 - * `toupper.asm`
- Sum of digits – string version
 - * `addigits.asm`
- Sum of digits – number version
 - * `addigits2.asm`

Procedures

- Two instructions
 - * Procedure call
 - » `jal` (jump and link)
`jal proc_name`
 - * Return from a procedure
`jr $ra`
- Parameter passing
 - Via registers
 - Via the stack
- Examples
 - » `min_max.asm`
 - » `str_len.asm`

Stack Implementation

- No explicit support

- » No push/pop instructions
- » Need to manipulate stack pointer explicitly
 - Stack grows downward as in Pentium

- * Example: push registers **a0** and **ra**

```
sub    $sp, $sp, 8    #reserve 8 bytes of stack
sw     $a0, 0($sp)   #save registers
sw     $ra, 4($sp)
```

- * **pop** operation

```
lw     $a0, 0($sp)   #restore registers
lw     $a0, 4($sp)
addu   $sp, $sp, 8    #clear 8 bytes of stack
```


Illustrative Examples

- Passing variable number of parameters to a procedure

`var_para.asm`

- Recursion examples

`Factorial.asm`

`Quicksort.asm`

Last slide