MIPS Assembly Language Guide

MIPS is an example of a Reduced Instruction Set Computer (RISC) which was designed for easy instruction pipelining. MIPS has a "Load/Store" architecture since all instructions (other than the load and store instructions) must use register operands. MIPS has 32 32-bit "general purpose" registers ($0, $1, $2, ... , $31), but some of these have special uses (see MIPS Register Conventions table).

Common MIPS Instructions (and **psuedo-instructions**)		
Type of Instruction	**MIPS Assembly Language**	**Register Transfer Language Description**
Memory Access (Load and Store)	lw $4, Mem	$4← [Mem]
	sw $4, Mem	Mem←$4
	lw $4, 16($3)	$4← [Mem at address in $3 + 16]
	sw $4, 16($3)	[Mem at address in $3 + 16]← $4
Move	**move** $4, $2	$4← $2
	li $4, 100	$4← 100
Load Address	**la** $5, mem	$4← load address of mem
Arithmetic Instruction (reg. operands only)	add $4, $2, $3	$4← $2 + $3
	mul $10, $12, $8	$10← $12 * $8 (32-bit product)
	sub $4, $2, $3	$4← $2 - $3
Arithmetic with Immediates (last operand must be an integer)	addi $4, $2, 100	$4← $2 + 100
	mul $4, $2, 100	$4← $2 * 100 (32-bit product)
Conditional Branch	**bgt** $4, $2, LABEL (**bge, blt, ble**, beq, bne)	Branch to LABEL if $4 > $2
Unconditional Branch	j LABEL	Always Branch to LABEL

A simple MIPS assembly language program to sum the elements in an array A is given below:

```
      .data
array:      .word 5, 10, 20, 25, 30, 40, 60
length:     .word 7
sum:        .word 0

# Algorithm being implemented to sum an array
#     sum = 0                           (use $8 for sum)
#     for i := 0 to length-1 do         (use $9 for i)
#           sum := sum + array[i]       (use $10 for length-1)
#     end for                           (use $11 for base addr. of array)

      .text
      .globl main
main:
      li    $8, 0              # load immediate 0 in reg. $8 (sum)
      la    $11, array         # load base addr. of array into $11
for:
      lw    $10, length        # load length in reg. $10
      addi  $10, $10, -1       # $10 = length - 1
      li    $9, 0              # initialize i in $9 to 0
for_compare:
      bgt   $9, $10, end_for   # drop out of loop when i > (length-1)
      mul   $12, $9, 4         # mult. i by 4 to get offset within array
      add   $12, $11, $12      # add base addr. of array to $12 to get addr. of array[i]
      lw    $12, 0($12)        # load value of array[i] from memory into $12
      add   $8, $8, $12        # update sum
      addi  $9, $9, 1          # increment i
      j     for_compare
end_for:

      sw    $8, sum

      li    $v0, 10            # system code for exit
      syscall
```

MIPS Logical Instructions	
and $4, $5, $6	$4←$5 (bit-wise AND) $6
andi $4, $5, 0x5f	$4←$5 (bit-wise AND) $5f_{16}$
or $4, $5, $6	$4←$5 (bit-wise OR) $6
ori $4, $5, 0x5f	$4←$5 (bit-wise OR) $5f_{16}$
xor $4, $5, $6	$4←$5 (bit-wise Exclusive-OR) $6
xori $4, $5, 0x5f	$4←$5 (bit-wise Exclusive-OR) $5f_{16}$
nor $4, $5, $6	$4←$5 (bit-wise NOR) $6
not $4, $5	$4←NOT $5 #inverts all the bits

MIPS Shift and Rotate Instructions	
sll $4, $5, 3	$4←shift left $5 by 3 positions. Shift in zeros (only least significant 5-bits of immediate value are used to shift)
sllv $4, $5, $6	Similar to sll, but least significant 5-bits of $6 determine the amount to shift.
srl $4, $5, 3	$4← shift right $5 by 3 positions. Shift in zeros
srlv $4, $5, $6	Similar to srl, but least significant 5-bits of $6 determine the amount to shift.
sra $4, $5, 3	$4←shift right $5 by 3 positions. Sign-extend (shift in sign bit)
srav $4, $5, $6	Similar to sra, but least significant 5-bits of $6 determine the amount to shift.
rol $4, $5, 3	$4←rotate left $5 by 3 positions
rol $4, $5, $6	Similar to above, but least significant 5-bits of $6 determine the amount to rotate.
ror $4, $5, 3	$4←rotate right $5 by 3 positions
ror $4, $5, $6	Similar to above, but least significant 5-bits of $6 determine the amount to rotate.

Common usages for shift/rotate and logical instructions include:

1. To calculate the address of element array[i], we calculate (base address of array) + i * 4 for an array of words. Since multiplication is a slow operation, we can shift the value left two bit positions. For example:

```
la $3, array            # load base address of array into $3
sll $10, $2, 2          # logical shift i's value in $2 by 2 to multiply its value by 4
add  $10, $3, $10       # finish calculation of the address of element array[i]
lw  $4, 0($10)          # load the value of array[i] into $4
```

2. Sometimes you want to manipulate individual bits in a "string of bits". For example, you can represent a set of letters using a bit-string. Each bit in the bit-string is associated with a letter: bit position 0 with 'A', bit position 1 with 'B', ..., bit position 25 with 'Z'. Bit-string bits are set to '1' to indicate that their corresponding letters are in the set. For example, the set { 'A', 'B', 'D', 'Y' } would be represented as:

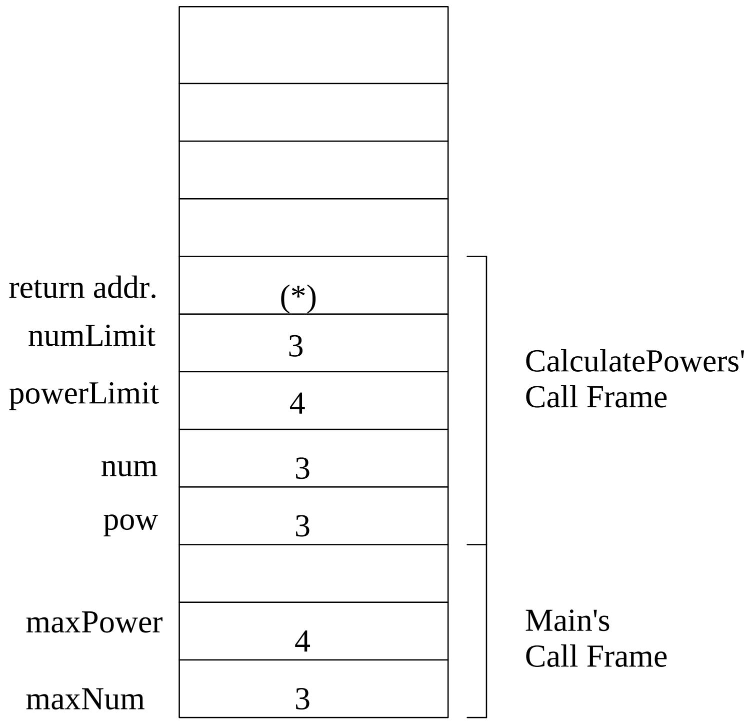
	unused	'Z'	'Y'	'X'	. . .	'E'	'D'	'C'	'B'	'A'
{ 'A', 'B', 'D', 'Y' } is	0 0 0 0 0 0	0	1	0		0	1	0	1	1
bit position:		25	24	23		4	3	2	1	0

To determine if a specific ASCII character, say 'C' (67_{10}) is in the set, you would need to build a "mask" containing a single "1" in bit position 2. The sequence of instructions "li $3, 1" followed by "sll $3, $3, 2" would build the needed mask in $3. If the bit-string set of letters is in register $5, then we can check for the character 'C' using the mask in $3 and the instruction "and $6, $5, $3". If the bit-string set in $5 contained a 'C', then $6 will be non-zero; otherwise $6 will be zero.

High-level Language Programmer's View

main:
 	
maxNum = 3
maxPower = 4
 	
CalculatePowers(maxNum, maxPower)
(*)
. . .
end main
	CalculatePowers(**In:** integer numLimit,
	integer powerLimit)
	
	integer num, pow
	
	for num := 1 to numLimit do
	for pow := 1 to powerLimit do
	
	print num " raised to " pow " power is "
	Power(num, pow)
	end for pow
	end for num
		integer **Power**(**In:** integer n, integer e)
		
		integer result
		if e = 0 then
		result = 1
		else if e = 1 then
		result = n
		else
		result = Power(n, e - 1)* n
		end if
		return result
		end Power

HLL View of Run-time Stack



Compiler uses registers to avoid accessing the run-time stack in memory as much as possible. Registers can be used for local variables, parameters, return address, function-return value.

```
AL code for subprogram "caller"

  <code using some registers>

 call subprogram

  <wants used registers to be unchanged>
```

When a subprogram is called, some of the register values might need to be saved ("spilled") on the stack to free up some registers for the subprogram to use.

Standard conventions for spilling registers:
1) caller save - before the call, caller saves the register values it needs after execution returns from the subprogram
2) callee save - subprogram saves and restores any register it uses in its code
3) some combination of caller and callee saved (USED BY MIPS)

MIPS Register Conventions			
Reg. #	**Convention Name**	**Role in Procedure Calls**	**Comments**
$0	$zero	constant value zero	Cannot be changed
$1	$at	Used by assembler to implement psuedoinstructions	DON'T USE
$2, $3	$v0, $v1	Results of a function	
$4 - $7	$a0 - $a3	First 4 arguments to a procedure	
$8 - $15, $24, $25	$t0 - $t9	Temporary registers (not preserved across call)	Caller-saved registers - subprogram can use them as scratch registers, but it must also save any needed values before calling another subprogram.
$16 - $23	$s0 - $s7	Saved temporary (preserved across call)	Callee-saved registers - it can rely on an subprogram it calls not to change them (so a subprogram wishing to use these registers must save them on entry and restore them before it exits)
$26, $27	$k0, $k1	Reserved for the Operating System Kernel	DON'T USE
$28	$gp	Pointer to global area	
$29	$sp	Stack pointer	Points to first free memory location above stack
$30	$fp/$s8	Frame pointer (if needed) or another saved register	$fp not used so use as $s8
$31	$ra	Return address (used by a procedure call)	Receives return addr. on *jal* call to procedure

Using MIPS Calling Convention	
Caller Code	**Callee Code**
. . . 1) save on stack any $t0 - $t9 and $a0 - $a3 that are needed upon return 2) place arguments to be passed in $a0 - $a3 with additional parameters pushed onto the stack 3) jal ProcName # saves return address in $ra 4) restore any saved registers $t0 - $t9 and $a0 - $a3 from stack	. . . 1) allocate memory for frame by subtracting frame size from $sp 2) save callee-saved registers ($s0 - $s7) if more registers than $t0 - $t9 and $a0 - $a3 are needed 3) save $ra if another procedure is to be called **. . . code for the callee** 4) for functions, place result to be returned in $v0 - $v1 5) restore any callee-saved registers ($s0 - $s7) from step (2) above 6) restore $ra if it was saved on the stack in step (3) 7) pop stack frame by adding frame size to $sp 8) return to caller by "jr $ra" instruction

main:	**CalculatePowers(In:** integer numLimit, integer powerLimit)	integer **Power**(**In:** integer n, integer e)
maxNum = 3		integer result
maxPower = 4	integer num, pow	if e = 0 then
		result = 1
CalculatePowers(maxNum, maxPower)	for num := 1 to numLimit do	else if e = 1 then
(*)	for pow := 1 to powerLimit do	result = n
. . .		else
end main	print num " raised to " pow " power is "	result = Power(n, e - 1)* n
	Power(num, pow)	end if
	end for pow	return result
	end for num	**end Power**
	end CalculatePowers	

a) Using the MIPS register conventions, what registers would be used to pass each of the following parameters to CalculatePowers:

maxNum	maxPower

b) Using the MIPS register conventions, which of these parameters ("numLimit", "powerLimit", or both of them) should be moved into s-registers?
(**NOTE: Use an s-register for any value you still need after you come back from a subprogram/function/procedure call, e.g., call to "Power")**

c) Using the MIPS register conventions, what registers should be used for each of the local variables:

num	pow

d) Using the MIPS register conventions, what registers would be used to pass each of the following parameters to Power:

num	pow

e) Using the MIPS register conventions, which of these parameters ("n", "e", or both of them) should be moved into s-registers?

f) Using the MIPS register conventions, what register should be used for the local variable:

result

g) Write the code for main, CalculatePowers, and Power in MIPS assembly language.

main:	**InsertionSort**(numbers - address to integer array, length - integer)	**Insert**(numbers - address to integer array, elementToInsert - integer, lastSortedIndex - integer) {

integer scores [100];
integer n; // # of elements
 	
InsertionSort(scores, n)
 	
(*)
. . .
 	
end main
	integer firstUnsortedIndex
	for firstUnsortedIndex = 1 to (length-1) do
	Insert(numbers, numbers[firstUnsortedIndex], firstUnsortedIndex-1);
	end for
	
	end InsertionSort
		integer testIndex;
		testIndex = lastSortedIndex;
		while (testIndex >=0) AND (numbers[testIndex] > elementToInsert) do
		numbers[testIndex+1] = numbers[testIndex];
		
		testIndex = testIndex - 1;
		end while
		numbers[testIndex + 1] = elementToInsert;
		end Insert

a) Using the MIPS register conventions, what registers would be used to pass each of the following parameters to InsertionSort:

scores	n

b) Using the MIPS register conventions, which of these parameters ("numbers", "length", or both of them) should be moved into s-registers?

c) Using the MIPS register conventions, what registers should be used for the local variable "firstUnsortedIndex"?

d) Using the MIPS register conventions, what registers would be used to pass each of the following parameter values to Insert:

numbers	numbers[firstUnsortedIndex]	firstUnsortedIndex-1

e) Using the MIPS register conventions, which of these parameters ("numbers", "elementToInsert", or "lastSortedIndex") should be moved into s-registers?

f) Using the MIPS register conventions, what registers should be used for the local variable "testIndex"?

g) Write the code for main, InsertionSort, and Insert in MIPS assembly language.

PCSpim I/O Support

Access to Input/Output (I/O) devices within a computer system is generally restricted to prevent user programs from directly accessing them. This prevents a user program from accidentally or maliciously doing things like:

- reading someone else's data file from a disk
- writing to someone else's data file on a disk
- etc.

However, user programs need to perform I/O (e.g., read and write information to files, write to the console, read from the keyboard, etc.) if they are to be useful. Therefore, most computer systems require a user program to request I/O by asking the operating system to perform it on their behalf.

PCSpim uses the "syscall" (short for "system call") instruction to submit requests for I/O to the operating system. The register $v0 is used to indicate the type of I/O being requested with $a0, $a1, $f12 registers being used to pass additional parameters to the operating system. Integer results and addresses are returned in the $v0 register, and floating point results being returned in the $f0 register. The following table provides details of the PCSpim syscall usage.

Service Requested	**System call code passed in $v0**	**Registers used to pass additional arguments**	**Registers used to return results**
print_int	1	$a0 contains the integer value to print	
print_float	2	$f12 contains the 32-bit float to print	
print_double	3	$f12 (and $f13) contains the 64-bit double to print	
print_string	4	$a0 contains the address of the .asciiz string to print	
read_int	5		$v0 returns the integer value read
read_float	6		$f0 returns the 32-bit floating-point value read
read_double	7		$f0 and $f1 returns the 64-bit floating-point value read
read_string	8	$a0 contains the address of the buffer to store the string $a1 contains the maximum length of the buffer	
sbrk - request a memory block	9	$a0 contains the number of bytes in the requested block	$v0 returns the starting address of the block of memory
exit	10		

CalculatePowers subprogram example using MIPS register conventions and PCSpim syscalls

```
                .data
maxNum:         .word 3
maxPower:       .word 4
str1:           .asciiz " raised to "
str2:           .asciiz " power is "
str3:           .asciiz  "\n"                # newline character

                .text
                .globl main
main:
                lw      $a0, maxNum          # $a0 contains maxNum
                lw      $a1, maxPower        # $a1 contains maxPower
                jal     CalculatePower

                li      $v0, 10              # system code for exit
                syscall


################################################################################
CalculatePower:                              # $a0 contains value of numLimit
                                             # $a1 contains value of powerLimit

                addi    $sp, $sp, -20        # save room for the return address
                sw      $ra, 4($sp)          # push return address onto stack
                sw      $s0, 8($sp)
                sw      $s1, 12($sp)
                sw      $s2, 16($sp)
                sw      $s3, 20($sp)

                move    $s0, $a0             # save numLimit in $s0
                move    $s1, $a1             # save powerLimit in $s1

for_1:
                li      $s2, 1               # $s2 contains num
for_compare_1:
                bgt     $s2, $s0, end_for_1
for_body_1:

for_2:
                li      $s3, 1               # $s3 contains pow
for_compare_2:
                bgt     $s3, $s1, end_for_2
for_body_2:
                move    $a0, $s2             # print num
                li      $v0, 1
                syscall
```

```
                la      $a0, str1               # print " raised to "
                li      $v0, 4
                syscall

                move    $a0, $s3                # print pow
                li      $v0, 1
                syscall

                la      $a0, str2               # print " power is "
                li      $v0, 4
                syscall

                move    $a0, $s2                # call Power(num, pow)
                move    $a1, $s3
                jal     Power

                move    $a0, $v0                # print result

                li      $v0, 1
                syscall

                la      $a0, str3               # print new-line character
                li      $v0, 4
                syscall

                addi    $s3, $s3, 1
                j       for_compare_2
end_for_2:

                addi    $s2, $s2, 1
                j       for_compare_1
end_for_1:

                lw      $ra, 4($sp)             # restore return addr. to $ra
                lw      $s0, 8($sp)             # restore saved $s registers
                lw      $s1, 12($sp)
                lw      $s2, 16($sp)
                lw      $s3, 20($sp)

                addi    $sp, $sp, 20            # pop call frame from stack
                jr      $ra
end_CalculatePowers:
```

```
##############################################################################
Power:                                  # $a0 contains n (we never change it during the
                                        #      recursive calls so we don't need to save it)
                                        # $a1 contains e
            addi    $sp, $sp, -4
            sw      $ra, 4($sp)         # save $ra on stack

if:
            bne     $a1, $zero, else_if
            li      $v0, 1              # $v0 contains result
            j       end_if
else_if:
            bne     $a1, 1, else
            move    $v0, $a0
            j       end_if
else:                                   # first parameter is still n in $a0
            addi    $a1, $a1, -1        # put second parameter, e-1, in $a1
            jal     Power               # returns with value of Power(n, e-1) in $v0
            mul     $v0, $v0, $a0       # result = Power(n, e-1) * n
end_if:
            lw      $ra, 4($sp)         # restore return addr. to $ra
            addi    $sp, $sp, 4         # pop call frame from stack
            jr      $ra
end_Power:
```

Snap-shot of the Console window after the program executes: