1) Assign variables to registers and then translate the following high-level language code segment to MIPS assembly language.

\[
\text{if } X < Y \text{ then}
\]  
\[
\quad \text{min} = X
\]  
\[ \text{else} \]
\[
\quad \text{min} = Y
\]  
\[ \text{end if} \]

2. Translate the following high-level language code segment to MIPS assembly language. Use the registers indicated in the code.

a) for $4 = 0$ to 100 by steps of size 10 do
\[
\text{if } (S3 < S4) \text{ AND } (S2 \geq 50) \text{ then}
\]  
\[
\quad S2 = S2 + S3
\]  
\[ \text{end if} \]
\[ \text{end for} \]

b) while $(S8 > 20)$ do
\[
\text{if } (S8 < 100) \text{ OR } (S8 > 200) \text{ then}
\]  
\[
\quad S7 = S8
\]  
\[
\quad S8 = S8 - 10
\]  
\[ \text{else} \]
\[
\quad S8 = S8 - S7
\]  
\[ \text{end if} \]
\[
\quad S7 = S6 + 4
\]  
\[ \text{end while} \]
3) Use a laptop and MIPS simulator (PCSpim, QtSpim, MARS, etc.) to enter and run the following MIPS program.

```
.data
x: .word -3
y: .word 5
product: .word 0
sum: .word 0

.text
.globl main

# High-level language program
#    product = x * y
#    sum = x + y

# Register usage
#    x is in $2
#    y is in $3
#    product is in $4
#    sum is in $5
main:
    lw $2, x
    lw $3, y
    mul $4,$2,$3
    add $5,$2,$3
    sw $4, product
    sw $5, sum

    li $v0, 10     # system call code to exit program
    syscall
```
RISC Arch. (CISC - Intel x86)

Load/Store machine

add $2, $3, $4 # $2 ← $3 + $4

lw $t0, 0($sp) # load $t0, src
lw $t1, 0($sp) # load $t1, src
add $2, $t0, $t1 # $2 ← $t0 + $t1

Reg. File 32 reg. 32-bit

0: Don't Use
1: $200
2: $1
3: $2

4GB (2^32) 32-bit
byte addr. addr.

Lecture 16-1
HLL:
if \( x < y \) then

"then body"

else

"else body"

end if

HLL:
for \( i = 1 \) to 10 do

for_init: \( i \leftarrow 1 \)

for_compare: \( i \leq 10 \)

body

end_for:
# 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).

<table>
<thead>
<tr>
<th>Common MIPS Instructions (and psuedo-instructions)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type of Instruction</strong></td>
</tr>
<tr>
<td>Memory Access (Load and Store)</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Move</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Load Address</td>
</tr>
<tr>
<td>Arithmetic Instruction (reg. operands only)</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Arithmetic with Immediates (last operand must be an integer)</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Conditional Branch</td>
</tr>
<tr>
<td>Unconditional Branch</td>
</tr>
</tbody>
</table>

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

```mips
.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
# for i := 0 to length-1 do
# sum := sum + array[i]
# end for

.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     # multi. 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
    for_compare
end_for:
    sw $8, sum
    li $v0, 10         # system code for exit
syscall
```
MIPS Logical Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>and $4, $5, $6</code></td>
<td>$4 ← $5 (bit-wise AND) $6</td>
</tr>
<tr>
<td><code>andi $4, $5, 0xf</code></td>
<td>$4 ← $5 (bit-wise AND) $5f_6</td>
</tr>
<tr>
<td><code>or $4, $5, $6</code></td>
<td>$4 ← $5 (bit-wise OR) $6</td>
</tr>
<tr>
<td><code>ori $4, $5, 0xf</code></td>
<td>$4 ← $5 (bit-wise OR) $5f_6</td>
</tr>
<tr>
<td><code>xor $4, $5, $6</code></td>
<td>$4 ← $5 (bit-wise Exclusive-OR) $6</td>
</tr>
<tr>
<td><code>xori $4, $5, 0xf</code></td>
<td>$4 ← $5 (bit-wise Exclusive-OR) $5f_6</td>
</tr>
<tr>
<td><code>nor $4, $5, $6</code></td>
<td>$4 ← $5 (bit-wise NOR) $6</td>
</tr>
<tr>
<td><code>not $4, $5</code></td>
<td>$4 ← NOT $5 # inverts all the bits</td>
</tr>
</tbody>
</table>

MIPS Shift and Rotate Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sll $4, $5, 3</code></td>
<td>$4 ← shift left $5 by 3 positions. Shift in zeros (only least significant 5-bits of immediate value are used to shift)</td>
</tr>
<tr>
<td><code>sllv $4, $5, $6</code></td>
<td>Similar to sll, but least significant 5-bits of $6 determine the amount to shift.</td>
</tr>
<tr>
<td><code>srl $4, $5, 3</code></td>
<td>$4 ← shift right $5 by 3 positions. Shift in zeros</td>
</tr>
<tr>
<td><code>srlv $4, $5, $6</code></td>
<td>Similar to srl, but least significant 5-bits of $6 determine the amount to shift.</td>
</tr>
<tr>
<td><code>sra $4, $5, 3</code></td>
<td>$4 ← shift right $5 by 3 positions. Sign-extend (shift in sign bit)</td>
</tr>
<tr>
<td><code>srav $4, $5, $6</code></td>
<td>Similar to sra, but least significant 5-bits of $6 determine the amount to shift.</td>
</tr>
<tr>
<td><code>rol $4, $5, 3</code></td>
<td>$4 ← rotate left $5 by 3 positions</td>
</tr>
<tr>
<td><code>rol $4, $5, $6</code></td>
<td>Similar to above, but least significant 5-bits of $6 determine the amount to rotate.</td>
</tr>
<tr>
<td><code>ror $4, $5, 3</code></td>
<td>$4 ← rotate right $5 by 3 positions</td>
</tr>
<tr>
<td><code>ror $4, $5, $6</code></td>
<td>Similar to above, but least significant 5-bits of $6 determine the amount to rotate.</td>
</tr>
</tbody>
</table>

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

```
<table>
<thead>
<tr>
<th>'A', 'B', 'D', 'Y'</th>
<th>is</th>
<th>0 0 0 0 0 0 1 0 0 0 0 0 1 0 1 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit position:</td>
<td></td>
<td>25 24 23 4 3 2 1 0</td>
</tr>
</tbody>
</table>
```

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

```plaintext
main:
maxNum = 3
maxPower = 4
CalculatePowers(maxNum, maxPower)
(*)
...
end main
```

```plaintext
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
del end Power
```

HLL View of Run-time Stack

<table>
<thead>
<tr>
<th>return addr.</th>
<th>(*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>numLimit</td>
<td>3</td>
</tr>
<tr>
<td>powerLimit</td>
<td>4</td>
</tr>
<tr>
<td>num</td>
<td>3</td>
</tr>
<tr>
<td>pow</td>
<td>3</td>
</tr>
</tbody>
</table>

| maxPower     | 4   |
| maxNum       | 3   |

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"

```plaintext
<code using some registers>
call subprogram
```

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

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

### Using MIPS Calling Convention

**Caller 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

**Callee Code**

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:

<table>
<thead>
<tr>
<th>CalculatePowers(In: integer numLimit, integer powerLimit)</th>
<th>integer Power(In: integer n, integer e)</th>
</tr>
</thead>
<tbody>
<tr>
<td>maxNum = 3</td>
<td></td>
</tr>
<tr>
<td>maxPower = 4</td>
<td></td>
</tr>
<tr>
<td>CalculatePowers(maxNum, maxPower)</td>
<td></td>
</tr>
<tr>
<td>for num := 1 to numLimit do</td>
<td></td>
</tr>
<tr>
<td>(*)</td>
<td></td>
</tr>
<tr>
<td>for pow := 1 to powerLimit do</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>end main</td>
<td></td>
</tr>
<tr>
<td>end for pow</td>
<td></td>
</tr>
<tr>
<td>end for num</td>
<td></td>
</tr>
<tr>
<td>print num &quot; raised to &quot; pow &quot; power is &quot;</td>
<td></td>
</tr>
<tr>
<td>Power(num, pow)</td>
<td></td>
</tr>
<tr>
<td>result = Power(n, e - 1)* n</td>
<td></td>
</tr>
<tr>
<td>end if</td>
<td></td>
</tr>
<tr>
<td>return result</td>
<td></td>
</tr>
<tr>
<td>end Power</td>
<td></td>
</tr>
<tr>
<td>end CalculatePowers</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>maxNum</th>
<th>maxPower</th>
</tr>
</thead>
</table>

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:

<table>
<thead>
<tr>
<th>num</th>
<th>pow</th>
</tr>
</thead>
</table>

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

<table>
<thead>
<tr>
<th>num</th>
<th>pow</th>
</tr>
</thead>
</table>

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:

integer scores [100];
integer n; // # of elements

InsertionSort(scores, n)

(*)

end main

Insert(numbers - address to integer array,
       length - integer)

integer firstUnsortedIndex
for firstUnsortedIndex = 1 to (length-1) do
   Insert(numbers, numbers[firstUnsortedIndex],
          firstUnsortedIndex-1);
end for

end InsertionSort

Insert(numbers - address to integer array,
        elementToInsert - integer,
        lastSortedIndex - integer) {
   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:

<table>
<thead>
<tr>
<th>scores</th>
<th>n</th>
</tr>
</thead>
</table>

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:

<table>
<thead>
<tr>
<th>numbers</th>
<th>numbers[firstUnsortedIndex]</th>
<th>firstUnsortedIndex-1</th>
</tr>
</thead>
</table>

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.

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

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:

addi $a1, $a1, -1 # first parameter is still n in $a0
jal Power # put second parameter, e-1, in $a1
mul $v0, $v0, $a0 # returns with value of Power(n, e-1) in $v0

end_if:

lw $ra, 4($sp) # result = Power(n, e-1) * n
addi $sp, $sp, 4 # restore return addr. to $ra
jr $ra # pop call frame from stack

end_Power:

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

1 raised to 2 power is 1
1 raised to 3 power is 1
1 raised to 4 power is 1
2 raised to 1 power is 2
2 raised to 2 power is 4
2 raised to 3 power is 8
2 raised to 4 power is 16
3 raised to 1 power is 3
3 raised to 2 power is 9
3 raised to 3 power is 27
3 raised to 4 power is 81
1) Consider the following array scores:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>scores:</td>
<td>10</td>
<td>30</td>
<td>45</td>
<td>20</td>
<td>80</td>
<td>20</td>
<td>70</td>
<td>30</td>
<td>50</td>
</tr>
<tr>
<td>length</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

a) What MIPS instruction would load the base (starting) address of scores into register $12$?

b) If register $5$ contained the index $i$'s value, what MIPS instructions would calculate the address of scores[$i$] into register $13$?

c) What MIPS instruction would load the value of scores[$i$] into register $14$?

2) Complete the MIPS sequential search program that finds the index location of the first occurrence of targetValue in the array scores. If the targetValue is not in the array, then foundIndex should be set to -1.

```plaintext
# MIPS sequential search program -- High-level algorithm:
# foundIndex = -1
# for i = 0 to (length-1) do
#    if scores[i] == targetValue then
#      foundIndex = i
#      break out of loop
#    end if
# end for

.data
scores: .word 10, 30, 45, 20, 80, 20, 70, 30, 50
length: .word 9
targetValue: .word 20
foundIndex: .word 0

.text
.globl main

main:
```
integer firstUnsortedIndex, testIndex, elementToInsert;
for firstUnsortedIndex = 1 to (length-1) do
    testIndex = firstUnsortedIndex-1;
    elementToInsert = numbers[firstUnsortedIndex];
    while (testIndex >=0) AND (numbers[testIndex] > elementToInsert ) do
        numbers[ testIndex + 1 ] = numbers[ testIndex ];
        testIndex = testIndex - 1;
    end while
    numbers[ testIndex + 1 ] = elementToInsert;
end for

3. Write MIPS Assembly Language code for the above insertion sort algorithm

.data
numbers:    .word 20, 30, 10, 40, 50, 60, 30, 25, 10, 5
length:     .word 10

.text
.globl main

main:
li $v0, 10
syscall # system call to exit
Lecture 17  1-D array MIPS Examples

1-D arrays

array:

\[
\begin{array}{ccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 \\
5 & 10 & 20 & 25 & 30 & 40 & 60 \\
\end{array}
\]

HLL:

\[
x = \text{array}[3] \\
\text{array}[3] = 23
\]

RAM access \(O(1) \) "constant time"

"n" items in array

addr. of array\[i\] = (base addr. of array) + (i \times (elt. size))

\[
\begin{array}{c}
\text{addr.} \\
\text{of array[i]} \\
$12 \\
\end{array}
\]

la $11, array

\[
\text{#i is } 9
\]

mul $12, $9, 4

add $12, $11, $12

lw $14, 0($12)
Bubble Sort: ascending sort smaller → larger

Simple Sort:

numbers:

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

Outer loop - track the dividing line between the sorted and unsorted part of the array.

Inner loop - extend the sorted by one item.

HLL:

For lastUnsorted = (length - 1) do
  for test = 0 to (lastUnsorted - 1) do
    if numbers[test] > numbers[test + 1] then
      temp = numbers[test]
      numbers[test] = numbers[test + 1]
      numbers[test + 1] = temp
    end if
  end for
end for

Lecture 17-2
for_init_1:  lw $9, length
            addi $8, $9, -1

for_compare_1:  blt $8, 1, end_for_1

for_init_2:  li $10, 0
            addi $11, $8, -1

for_compare_2:  bgt $10, $11, end_for_2

if:
    mul $13, $10, 4
    add $13, $12, $13  # addr_numbersitize
    lw $14, 0($13)  # value_numbersitize
    lw $15, 4($13)  # value_numbersitize
    ble $14, $15, end_if
    sw $14, 4($13)
    sw $15, 0($13)
end_if:
    addi $10, $10, 1
    j for_compare_2

end_for_2:
    addi $8, $8, -1
    j for_compare_1

end_for_1:
lecture 17-3
lecl7_bubble_sort.s

# Bubble sort code in MIPS
# HLL:
# for lastUnsorted = (length-1) downto 1 do
#   for test = 0 to (lastUnsorted -1) do
#     if numbers[test] > numbers[test + 1] then
#       temp = numbers[test]
#       numbers[test] = numbers[test + 1]
#       numbers[test + 1] = temp
#     end if
#   end for
# end for
#
# Register Usage:
# $8 is lastUnsorted
# $10 is test
# $11 is (lastUnsorted - 1)
# $12 is base address of array numbers
# $13 is address of numbers[test]
# $14 is value of numbers[test]
# $15 is value of numbers[test + 1]

.data
numbers: .word 30, 10, 20, 5, 90, 40, 60
length: .word 7

.text
.globl main
main:
  la $12, numbers
for_init_1:  
    lw $9, length
    addi $8, $9, -1
for_compare_1: 
    blt $8, 1, end_for_1
for_init_2:  
    li $10, 0
    addi $11, $8, -1
for_compare_2: 
    bgt $10, $11, end_for_2
if:  
    mul $13, $10, 4     # addr. of numbers[test]
    add $13, $12, $13   # value of numbers[test]
    lw $14, 0($13)      # value of numbers[test+1]
    lw $15, 4($13)
    ble $14, $15, end_if
    sw $14, 4($13)
    sw $15, 0($13)
end_if:     
    addi $10, $10, 1

Page 1
```
end_for_2:
    addi $8, $8, -1
    j for_compare_1
end_for_1:
    li $v0, 10  # system call code for exit program
    syscall
```
MIPS Assembly Language Supplement (Section 4.14 of the textbook)

Section 4.14 does a good job emphasizing that MARIE is a toy architecture that lacks key feature of real-world computer architectures. Most noticeable, MARIE only supports a limited subroutine call, but it lacks a run-time stack to support more powerful subprograms: procedure, function, or method calls. Additionally, MARIE has only one “general-purpose” register, the ACCumulator, for temporary storage within the CPU, so it is difficult and inefficient to program with lots of LOADs and STOREs for intermediate results while processing. Real-world computers have many (10s or 100s) general-purpose registers to store temporary results while calculating. Both the Intel and MIPS architectures discussed in the textbook support subprograms and have multiple registers, but Intel is a Complex Instruction Set Computer (CISC) architecture while MIPS is a Reduced Instruction Set Computer (RISC) architecture.

A CISC approach to instruction set design was the traditional approach through the early 1980’s. The main philosophy was to make assembly language (AL) as much like a high-level language (HLL) as possible to reduce the “semantic gap” between AL and HLL. The rational for CISC at the time was to:
- reduce compiler complexity and aid assembly language programming. Compilers were not too good during the 50’s to 70’s, (e.g., they made poor use of general purpose registers so code was inefficient) so some programs were written in assembly language.
- reduce the program size. More powerful/instructional instructions reduced the number of instructions necessary in a program. Memory during the 50’s to 70’s was limited and expensive.
- improve code efficiency by allowing complex sequence of instructions to be implemented in microcode. For example, the Digital Equipment Corporation (DEC) VAX computer had an assembly-language instruction “MATCHC substrLength, substr, strLength, str” that looks for a substring within a string.

The architectural characteristics of CISC machines include:
- complex, high-level like AL instructions
- variable format machine-language instructions that execute using a variable number of clock cycles
- many addressing modes (e.g., the DEC VAX had 22 addressing modes)

By the early 1980’s, some computer engineers were seeing some problems with the CISC approach:
- complex CPU hardware, including a microprogrammed control unit, was needed to implement more and complex instructions which slows the execution of simpler instructions
- high-level language compilers could rarely figure out when to use complex instructions. Assembly-language programmer could may use of complex instructions, but compilers had become more efficient with respect to register usage, and programs were larger and harder to write in assembly language. Thus, fewer programs were written in assembly language.
- variability in instruction format and instruction execution time made CISC hard to pipeline. Instruction pipelining through the CPU speeds up program execution like an assembly-line speeds up manufacturing of a car. A car assembly line might split up building a car into four stages: chassis, motor, interior, and exterior.

Assume that the whole car assembly process takes 4 hours. If you divide the process into four equal stages of an hour each, then ideally we can complete a car every hour. Problems occur if the stages are not equally balanced for all cars. If someone ordered the deluxe interior package that takes two hours to install, then the chassis, motor, and exterior workers get an hour break during the second hour of the deluxe interior package installation.
MIPS Assembly Language Supplement (Section 4.14 of the textbook)

The main RISC philosophy (mid-80’s and after) is to design the assembly language (AL) to optimize the instruction pipeline to speed program execution. One possible break down of an instruction execution into stages would be:

<table>
<thead>
<tr>
<th>Stage</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch</td>
<td>Read next instruction into CPU and increment PC to next instruction</td>
</tr>
<tr>
<td>Decode</td>
<td>Determine opcode, and read register operands from the register file</td>
</tr>
<tr>
<td>Execution</td>
<td>Calculate using register operands read in the Decode stage. The ALU calculation depends on the type instruction being performed:</td>
</tr>
<tr>
<td></td>
<td>- memory reference (load/store): calculate the effective memory address of the operand</td>
</tr>
<tr>
<td></td>
<td>- arithmetic operation (add, sub, etc.) with two register operands</td>
</tr>
<tr>
<td></td>
<td>- arithmetic operation with a register and an immediate constant</td>
</tr>
<tr>
<td>Memory</td>
<td></td>
</tr>
<tr>
<td>access</td>
<td>- load: read memory from effective address into a temporary pipeline register</td>
</tr>
<tr>
<td></td>
<td>- store: write register value from Decode stage to memory at effective address</td>
</tr>
<tr>
<td>Write-back</td>
<td>- ALU or load instruction: write result into register file</td>
</tr>
</tbody>
</table>

The architectural characteristics of RISC machines include:

- one instruction completion per clock cycle. This means that each stage needs fit in one clock cycle.
- large number of registers with register-to-register operations (e.g., “ADD R2, R3, R4,” where R2 gets the results of R3 + R4). Register operands are already in the CPU so they are fast to access.
- simple addressing modes because complex address calculations might take longer than one clock cycle
- simple, fixed-length instruction formats. Fixed-length instructions require a fixed amount of time to fetch. Simple instruction formats can be decoded in a clock cycle. MIPS instruction formats are all 32-bits, and are as follows:

**Arithmetic:** add R1, R2, R3

<table>
<thead>
<tr>
<th>opcode</th>
<th>dest</th>
<th>operand1</th>
<th>operand2</th>
<th>unused</th>
</tr>
</thead>
</table>

**Unconditional Branch/"jump":** j someLabel

| opcode | large offset from PC or absolute address |

**Arithmetic with immediate:** addi R1, R2, 8

<table>
<thead>
<tr>
<th>opcode</th>
<th>operand1</th>
<th>operand2</th>
<th>immediate value</th>
</tr>
</thead>
</table>

**Conditional Branch:** beq R1, R2, end_if

<table>
<thead>
<tr>
<th>opcode</th>
<th>operand1</th>
<th>operand2</th>
<th>PC-relative offset to label</th>
</tr>
</thead>
</table>

**Load/Store:** lw R1, 16(R2)

| opcode | operand1 | base | offset from base |

- hardwired control unit. The simple instructions can be performed using hardwired control unit that allows for a fast clock cycle

Most high-level programming languages (C, C++, Ada, Cobol, Java, Python, etc.) enable programs to be written in small reusable sections of code called subprograms that perform a specific task. A subprogram can be invoked using different actual parameters to allow them to perform their task on different data values. When writing the subprogram, formal parameters are used to describe the task. When a subprogram is called, the actual parameter values are passed to the formal parameters. This is called parameter passing.
MIPS Assembly Language Supplement (Section 4.14 of the textbook)

To help manage memory for subprograms, a run-time stack is used to provide memory space for a subprogram when it is called and delete it when it completes/returns. Specifically, when a subprogram is called, a call-frame (activation record) is pushed on top of the run-time stack which contains:

- the return address - where to return execution after the the subprogram returns
- space for the formal parameters - these get initialized to the value of their corresponding actual parameter from the subprogram call
- space for local variables - temporary variables allocated within the subprogram

After the call-frame is setup, the execution begins at the beginning of the subprogram. When the subprogram completes/returns, its call-frame is popped off the run-time stack and execution resumes at the return address. If the subprogram is a function, then a return value will be returned to the return address. Consider the scenario of subprogram A calling subprogram B, then subprogram B calling subprogram C.

Snapshots of the run-time stack over time:
MIPS Assembly Language Supplement (Section 4.14 of the textbook)

Consider the more realistic program that calculates the values:

\[
\begin{array}{cccc}
1^1 & 1^2 & 1^3 & 1^4 \\
2^1 & 2^2 & 2^3 & 2^4 \\
3^1 & 3^2 & 3^3 & 3^4 \\
\end{array}
\]

where the number and exponent ranges start at 1, but their upper bound are parameters to CalculatePowers. CalculatePowers uses a recursive function Power to calculate a number raised to an exponent. Recursion plays by the same rules as any other subprogram.

**main:**

```
maxNum = 3
maxPower = 4

CalculatePowers(maxNum, maxPower)
(*)
...
end main
```

**CalculatePowers( 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
```

**end CalculatePowers**

**integer function Power( 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
```
MIPS Assembly Language Supplement (Section 4.14 of the textbook)

The figure below shows a trace of the run-time stack after CalculatePowers called Power when num = 1 and pow = 1. When Power is called with n = 1 and e = 1, a base case of the recursion assigns result to 1 and then returns. The circled numbers indicate the order of events in the trace.

Run-time Stack

The figure below shows a trace of the run-time stack after CalculatePowers called Power when num = 3 and pow = 4. Notice that Power follows the same rules as any other subprogram with respect to parameter passing and the run-time stack. The circled numbers indicate the order of events in the trace.
MIPS Assembly Language Supplement (Section 4.14 of the textbook)

One of the main goals of the course is for you to learn how to program in assembly language. MARIE was a good introduction, and simple enough to discuss its control unit's implementation. However, its simplicity made it very difficult to actually write useful assembly-language programs in MARIE, so the next three assignments will focus on MIPS assembly-language programming.

Today very few people actually write code in assembly language, but knowing how is necessary to understanding how high-level language programming languages are implemented with respect to the run-time stack and built-in data structures such as arrays and records. In the past, people wrote in assembly-language for several reasons:

- to improve the speed of the program
- to decrease the amount of memory used to store the program
- to gain access to low-level features of the machine that are difficult from a high-level programming language (you might be writing a device driver for a new peripheral and need to fiddle with individual bits of data)

Fortunately, today's compilers generate extremely efficient machine-language code, so it is unlikely that the first two reasons apply. Plus, programs are generally much larger than in the past, so writing them all in assembly language would be difficult.

If you did want to speed up a high-level program by using assembly language, you would compile the program with a profiling option, and then run the program with real data. Having profiling turned on causes the program to track where it spends its execution time, and generates a report of the program's profile. Usually, over 85% of a program's time is spent executing a single subprogram. Thus, you can write just this subprogram in assembly language and leave the rest of the program in the high-level language (HLL). To correctly have the HLL program call your assembly-language subprogram, your assembly-language subprogram must follow the run-time stack and register conventions established for the processor. The register conventions are the rules about how the registers should be used. Before looking at the MIPS subprograms and its register convention, we must first learn to write simple main programs.

MIPS Assembly Language Guide:

As discussed earlier, 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). For now it's enough to know that register $0 always contains the value 0, $1 should never be used, and registers $2 to $25 can be used for writing simple main programs. (MIPS also has floating point registers and instruction, but we'll only focus on integer instructions)

Memory is byte addressable, but your can also access a 16-bit halfword, or a 32-bit word. The load word instruction, lw, reads a 32-bit word from memory into a register. An example might be "lw $4, X" where X is a label for a variable in memory. We'll mostly deal with signed 32-bit word data, but some times 8-bit byte data with be used for ASCII characters. The load byte, lb, instruction will be used to read character data. Store instructions are used to write a register value back to memory. For example, "sw $5, Y" would store register $5 to the label Y in memory.

The MIPS assembler is fairly helpful and provides the program with not only assembly instructions, but also psuedo-instructions that can be implemented by the assembler with a couple actual assembler instructions. The following table is list of common MIPS instructions and psuedo-instructions.
### MIPS Assembly Language Supplement (Section 4.14 of the textbook)

#### Common MIPS Instructions (and pseudo-instructions)

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

Let's look at a simple, complete MIPS program to calculate: \( \text{result} = (x + y) \times (10 - z) \), where \( x \) is 1, \( y \) is 2, and \( z \) is 3.

```mips
# Simple program to calculate: \( \text{result} = (x + y) \times (10 - z) \).

.data
x:    .word 1  # variable x initialized to 1 before program starts to execute
y:    .word 2  # variable y initialized to 2 before program starts to execute
z:    .word 3  # variable z initialized to 3 before program starts to execute
result: .word 0  # variable result initialized to 0 before program starts to execute

.text
.globl main
# main is global so it can be found at start
main:
lw $2, x # load values into registers
lw $3, y
lw $4, z
add $5, $2, $3 # $5 gets the value of \( x + y \)
li $6, 10 # load the value 10 into register $6
sub $6, $6, $4 # $6 gets the value of \( 10 - z \)
mul $6, $5, $6 # $6 gets the result
sw $6, result # save the result to memory
li $v0, 10 # system code for exit
syscall # call the operating system
```

A MIPS program needs a data segment (started with ".data") and a text segment (started with ".text") for the program. Execution of the program starts at the global "main" label and terminates with a system call to the operating system in the last two lines of the program. Notice that labels (named spots in memory) for variables and in the code (e.g., "main") end with a colon, ".".
MIPS Assembly Language Supplement (Section 4.14 of the textbook)

Let's look at how MIPS can be used to implement various HLL control structures. For example, consider the following IF-THEN-ELSE statement and corresponding flow-chart:

<table>
<thead>
<tr>
<th>HLL statement</th>
<th>Flow chart</th>
<th>Assembly Language</th>
</tr>
</thead>
</table>
| if X < Y then | ![Flow chart diagram](image) | lw $8, X  
lw $9, Y  
bge $8,$9, ELSE  
THEN: |
| ...          | False             | J END_IF          |
| else         | True              | ELSE: |
| ...          | then body         | END_IF: |
| end if       | else body         |               |

Since we want to conditionally jump over the THEN part when X < Y is False, the branch condition we check is the opposite of less-than, i.e., greater-than-or-equal (bge). If the THEN part is executed, then we jump to the END_IF.

For a loop example, consider the following FOR-loop and corresponding flow-chart:

<table>
<thead>
<tr>
<th>HLL statement</th>
<th>Flow chart</th>
<th>Assembly Language</th>
</tr>
</thead>
</table>
| for I = 1 to 10 do | ![Flow chart diagram](image) | FOR_INIT: li $5,1  
FOR_COND: bgt $5,10,END_FOR  
FOR_BODY: |
| ...          | I = 1             | END_FOR: |
|               | I < 10?           |               |
|               | False             |               |
|               | True              |               |
|               | for body          |               |
|               | I = I + 1         |               |
| end for       |                   | addi $5,$5,1  
j FOR_COND |

Register $5$ is used to store I in this example. We can initialize $5$ to $1$ by using the “load immediate” instruction: li $5, 1$. If I ≤ 10 is False, then we want to drop out of the loop. Since I ≤ 10 is False when I > 10, use the conditional branch instruction: bgt $5,10,END_FOR$ to drop out of the FOR loop. After the for-body executes and the loop-control variable I is incremented, the j FOR_COND loops back to recheck the loop control variable.
MIPS Assembly Language Supplement (Section 4.14 of the textbook)

Most high-level programming languages have an array data structure for storing a collection of same type elements. We generally view an array as a rectangle divide into smaller cells that can be access by specifying an index. Consider an array scores with room for 15 element, but only containing 7 items.

<table>
<thead>
<tr>
<th>scores:</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5</td>
<td>10</td>
<td>20</td>
<td>25</td>
<td>30</td>
<td>40</td>
<td>60</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In most HLL you use square-brackets, [ ], to access individual elements in the array. For example, scores[3] has the value 25. If we want to change the element at index 3, would assign “scores[3] = 23.” Arrays are implemented as a contiguous block of memory with a known starting location, called the base address. Because array elements are all the same size, we can calculate the address of some index “i” by:

address of array[i] = base address + (i * element size in bytes)

In a HLL the compiler generates code to perform this addressing calculation, but in assembly language its the programmer’s job. The MIPS code looks something like:

```assembly
.data
array:   .word 5, 10, 20, 25, 30, 40, 60, 0, 0, 0, 0, 0, 0, 0, 0
.
.
# code to access array[i], where i’s value is in register $5
la $4, array          # load the base address of the array in register $4

mul $6, $5, 4          # calculates i * the element size of 4 bytes
add $7, $4, $6         # $7 contains the complete address of array[i]
lw $8, 0($7)           # load the value of element array[i] to register $8
```

The above load instruction “lw $8, 0($7)” loads register $8 with the address specified by $0 ($7) where 0 is a displacement added to the address in $7. Since we calculated the exact address of array[i] in $7, adding 0 is what we want to do. A displacement is useful in an array if you are accessing nearby elements. For example, if we want to perform the assignment: array[i+1] = array[i], we could use the above code which reads the value of array[i] into $8, and then store $8 to at 4 bytes from where $7 points in memory, i.e., sw $8, 4($7).
A simple MIPS assembly language program to sum the elements in an array is given below:

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

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

.text
.globl main
main:
    li $8, 0
    la $11, array
    # load immediate 0 in reg. $8 (sum)
    # load base addr. of array into $11
    lw $10, length
    # load length in reg. $10
    addi $10, $10, -1
    li $9, 0
    # $10 = length - 1
    # initialize i in $9 to 0

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

end_for:
    sw $8, sum

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

In the above code each array element access involves one addition and one multiplication. One way to speed up this code is by walking pointers. Because of the regular access pattern of the array element, i.e., start at the beginning and move down the array sequentially on each iteration of the loop. Since the elements are words and each that up 4 bytes, we can just add 4 to the pointer register $11 on each iteration. Plus, we can eliminate the loop-control variable i if we calculate the stopping address and use it to compare to register $11 as we “walk” it down the array.

![Diagram showing array access and walking pointer](image)

The following program is the walking-pointer version. Walking a pointer reduces the number of calculations per iteration of the loop by one addition and one multiplication.
MIPS Assembly Language Supplement (Section 4.14 of the textbook)

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

# Algorithm being implemented to sum an array, but we are walking $11 down the array
# sum - 0 (use $8 for sum).
# for i := 0 to length-1 do (use $11 for the address of array[i])
# sum := sum + array[i] (use $10 for the stopping address, i.e., addr.
# of array[length])
#
end for

.text
globl main

main:
li $8, 0 # load immediate 0 in reg. $8 (sum)
la $11, array # load base addr. of array into $11, i.e., addr. of array[0]
for:
    lw $10, length # load length in reg. $10
    mul $10, $10, 4 # calculate the stopping address of array[length] in $10
    add $10, $11, $10 #
    for_compare:
        bge $11, $10, end_for # drop out of loop when $11 gets to stopping address in $10
        lw $12, 0($11) # load value of array[i] from memory into $12
        add $8, $8, $12 # update sum
        addi $11, $11, 4 # walk the pointer $11 to the next array element
        j for_compare
    end_for:
    sw $8, sum
li $v0, 10 # system code for exit
syscall

Multi-dimensional Arrays:
Consider a two-dimensional array M with 10 rows x 20 columns, we need to “unfold” this two-dimensional array into the one-dimensional memory. Two possible approaches could be taken:

- **column-major order** (see diagram below) where column 0 is followed by column 1 in memory, and column 1 is followed by column 2, etc.
- **row-major order** (see diagram below) where row 0 is followed by row 1 in memory, and row 1 is followed by row 2, etc.

![Column-major order diagram](attachment:column-major秩序.png)

Some high-level languages use one approach and some use the other. The choice is somewhat arbitrary, since access to an element requires the same type of calculations.

Let’s examine how row-major order would be packed into memory to develop the address calculation for an element M[r][c], i.e., row r and column c.
MIPS Assembly Language Supplement (Section 4.14 of the textbook)

**Row-Major Order**

Consider accessing element $M[3][2]$

- Start at the base address of $M$.
- Skip to the start of the correct Row:
  - For row 3, skip 3 rows: row 0, row 1, and row 2.
- Skip elements within the correct Row:
  - For column 2, skip 2 elements: column 0 and column 1.

To calculate the address of some element $M[r][c]$, we perform the calculation:

- Address of $M[r][c] = \text{base address} + r \times \text{size in a row} + c \times \text{size of an element}$
- Address of $M[r][c] = \text{base address} + r \times \# \text{ of columns} \times \text{size of an element} + c \times \text{size of an element}$
- Address of $M[r][c] = \text{base address} + (r \times \# \text{ of columns} + c) \times \text{size of an element}$
MIPS Assembly Language Supplement (Section 4.14 of the textbook)

The MIPS code to access $M[r][c]$ where $M$ has 10 rows and 20 columns and is stored in row-major order:

```
    # code to access $M[r][c]$, where $r$'s value is in register $s5$ and $c$'s is in $s6$
    la $s4, M$  # load the base address of the array in register $s4$
    mul $s7, s5, 20  # calculates $r \times \#$ of columns
    add $s7, s7, s6  # calculates $r \times \#$ of columns + $c$
    mul $s7, s7, 4  # calculates $(r \times \#$ of columns + $c) \times \$size of an element
    add $s8, s4, s7  # complete address calculation for $M[r][c]$
    lw $s8, 0(s7)  # load the value of element $M[r][c]$ to register $s8$
```

For a two-dimensional array, the address calculation takes 2 additions and 2 multiplications.

If we wanted to “walk” a pointer down a single column, say column 2, then we would just need to perform one addition to increment the pointer by the size of a row to move it from one element to the next, i.e., $M[0][2]$, $M[1][2]$, $M[2][2]$, $M[3][2]$, etc. Thus, a pointer would eliminate one addition and 2 multiplications per element access.
1) Using the idea of "walking pointers" re-do the MIPS sequential search program that finds the index location of the first occurrence of `targetValue` in the array `scores`. If the `targetValue` is not in the array, then `foundIndex` should be set to -1.

```plaintext
# MIPS sequential search program -- High-level algorithm:
# foundIndex = -1
# for i = 0 to (length-1) do
#   if scores[i] == targetValue then
#     foundIndex = i
#     break out of loop
# end if
# end for

.data
scores:     .word 10, 30, 45, 20, 80, 20, 70, 30, 50, 0, 0, 0, 0
length:     .word 9
targetValue: .word 20
foundIndex: .word 0

.text
.globl main

main:
```

2) For a 2D array `M` stored using column-major order, complete the formula to calculate the address of some element `M[r][c]`:

```
address of M[r][c] = base address +
```

Column-major order

start here

rows

```
columns

Lecture 18 Page 1`
integer firstUnsortedIndex, testIndex, elementToInsert;
for firstUnsortedIndex = 1 to (length-1) do
    testIndex = firstUnsortedIndex - 1;
    elementToInsert = numbers[firstUnsortedIndex];
    while (testIndex >= 0) AND (numbers[testIndex] > elementToInsert ) do
        numbers[ testIndex + 1 ] = numbers[ testIndex ];
        testIndex = testIndex - 1;
    end while
    numbers[ testIndex + 1 ] = elementToInsert;
end for

3. Write MIPS Assembly Language code for the above insertion sort algorithm by walking pointers.

    .data
    numbers:  .word 20, 30, 10, 40, 50, 60, 30, 25, 10, 5
    length:   .word 10

    .text
    .globl main

    main:

        li   $v0, 10
        syscall # system call to exit
Lecture 18 "Walking pointers" in an array and 2D arrays
A simple MIPS assembly language program to sum the elements in an array is given below:

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

# Algorithm being implemented to sum an array
# sum = 0
# for i := 0 to length-1 do
#   sum := sum + array[i]
# end for

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

In the above code each array element access involves one addition and one multiplication. One way to speed up this code is by walking pointers. Because of the regular access pattern of the array element, i.e., start at the beginning and move down the array sequentially on each iteration of the loop. Since the elements are words and each that up 4 bytes, we can just add 4 to the pointer register $11 on each iteration. Plus, we can eliminate the loop-control variable i if we calculate the stopping address and use it to compare to register $11 as we "walk" it down the array.

<table>
<thead>
<tr>
<th>array:</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5</td>
<td>10</td>
<td>20</td>
<td>25</td>
<td>30</td>
<td>40</td>
<td>60</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>+4</td>
<td>+4</td>
<td>+4</td>
<td>+4</td>
<td>+4</td>
<td>+4</td>
<td>+4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

la $11, array

The following program is the walking-pointer version. Walking a pointer reduces the number of calculations per iteration of the loop by one addition and one multiplication.
Multi-dimensional Arrays:
Consider a two-dimensional array M with 10 rows x 20 columns, we need to "unfold" this two-dimensional array into the one-dimensional memory. Two possible approaches could be taken:

- **column-major order** (see diagram below) where column 0 is followed by column 1 in memory, and column 1 is followed by column 2, etc.
- **row-major order** (see diagram below) where row 0 is followed by row 1 in memory, and row 1 is followed by row 2, etc.

Some high-level languages use one approach and some use the other. The choice is somewhat arbitrary, since access to an element requires the same type of calculations.

Let's examine how row-major order would be packed into memory to develop the address calculation for an element M[r][c], i.e., row r and column c.
Row-Major Order

Consider accessing element M[3][2]

```
M[0][0]
M[0][1]
M[0][2]
...
M[0][19]
M[1][0]
M[1][1]
M[1][2]
...
M[1][19]
M[2][0]
M[2][1]
M[2][2]
...
M[2][19]
M[3][0]
M[3][1]
M[3][2]
...
```

Start at the base address of M

3 rows

\[ 3 \times \text{(# of columns)} \times (\text{size}) \]

Skip to the start of the correct Row
For row 3, skip 3 rows: row 0, row 1, and row 2.

Skip elements within the correct Row
For column 2, skip 2 elements: column 0 and column 1.

To calculate the address of some element M[r][c], we perform the calculation:

- Address of M[r][c] = base address + \( r \times \text{size in a row} + c \times \text{size of an element} \)
- Address of M[r][c] = base address + \( r \times \text{# of columns} \times \text{size of an element} + c \times \text{size of an element} \)
- Address of M[r][c] = base address + \( (r \times \text{# of columns} + c) \times \text{size of an element} \)
MIPS Assembly Language Supplement (Section 4.14 of the textbook)

The MIPS code to access M[r][c] where M has 10 rows and 20 columns and is stored in row-major order:

```mips
la $4, M          # load the base address of the array in register $4
mul $7, $5, 20    # calculates r * # of columns
add $7, $7, $6    # calculates r * # of columns + c
mul $7, $7, 4     # calculates (r * # of columns + c)* size of an element
add $7, $4, $7    # complete address calculation for M[r][c]
lw $8, 0($7)      # load the value of element M[r][c] to register $8
```

For a two-dimensional array, the address calculation takes 2 additions and 2 multiplications.

If we wanted to "walk" a pointer down a single column, say column 2, then we would just need to perform one addition to increment the pointer by the size of a row to move it from one element to the next, i.e., M[0][2], M[1][2], M[2][2], M[3][2], etc. Thus, a pointer would eliminate one addition and 2 multiplications per element access.

\[
\text{sum} = 0 \\
\text{for } r = 0 \text{ to } 9 \text{ do } t + \\
\text{ for } c = 0 \text{ to } 19 \text{ do } 2x \\
\text{sum} = \text{sum} + M[r][c]
\]
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
end CalculatePowers

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

1) Trace the next execution (e.g., num is 3 and pow is 3) of the recursive function Power by showing the run-time stack.

2) What is the most number of call frames on the stack at any one time for the whole program?
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>main:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CalculatePowers(In: integer numLimit, integer powerLimit)</td>
</tr>
<tr>
<td></td>
<td>integer Power(In: integer n, integer e)</td>
</tr>
<tr>
<td>maxNum = 3</td>
<td>integer num, pow</td>
</tr>
<tr>
<td>maxPower = 4</td>
<td>if e = 0 then</td>
</tr>
<tr>
<td></td>
<td>result = 1</td>
</tr>
<tr>
<td></td>
<td>else if e = 1 then</td>
</tr>
<tr>
<td></td>
<td>result = n</td>
</tr>
<tr>
<td></td>
<td>else</td>
</tr>
<tr>
<td></td>
<td>print num &quot;raised to&quot; pow &quot;power is&quot;</td>
</tr>
<tr>
<td></td>
<td>Power(num, pow)</td>
</tr>
<tr>
<td></td>
<td>end if</td>
</tr>
<tr>
<td></td>
<td>end for pow</td>
</tr>
<tr>
<td></td>
<td>end for num</td>
</tr>
<tr>
<td></td>
<td>Power</td>
</tr>
<tr>
<td></td>
<td>end CalculatePowers</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>maxNum</td>
<td>maxNum</td>
</tr>
<tr>
<td>maxPower</td>
<td>maxPower</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>num</td>
<td>num</td>
</tr>
<tr>
<td>pow</td>
<td>pow</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>num</td>
<td>num</td>
</tr>
<tr>
<td>pow</td>
<td>pow</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>result</td>
</tr>
</tbody>
</table>

g) Write the code for main, CalculatePowers, and Power in MIPS assembly language.
Lecture 19 Run-time Stack in HLL
To help manage memory for subprograms, a run-time stack is used to provide memory space for a subprogram when it is called and delete it when it completes/returns. Specifically, when a subprogram is called, a call-frame (activation record) is pushed on top of the run-time stack which contains:

- the return address - where to return execution after the the subprogram returns
- space for the formal parameters - these get initialized to the value of their corresponding actual parameter from the subprogram call
- space for local variables - temporary variables allocated within the subprogram

After the call-frame is setup, the execution begins at the beginning of the subprogram. When the subprogram completes/returns, its call-frame is popped off the run-time stack and execution resumes at the return address. If the subprogram is a function, then a return value will be returned to the return address. Consider the scenario of subprogram A calling subprogram B, then subprogram B calling subprogram C.

Snapshots of the run-time stack over time:
Consider the more realistic program that calculates the values:

\[
\begin{array}{cccc}
1^1 & 1^2 & 1^3 & 1^4 \\
2^1 & 2^2 & 2^3 & 2^4 \\
3^1 & 3^2 & 3^3 & 3^4 \\
\end{array}
\]

where the number and exponent ranges start at 1, but their upper bound are parameters to CalculatePowers. CalculatePowers uses a recursive function Power to calculate a number raised to an exponent. Recursion plays by the same rules as any other subprogram.

**main:**

```
maxNum = 3
maxPower = 4

CalculatePowers(maxNum, maxPower)
(*)

...

end main
```

**CalculatePowers**

```
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

end CalculatePowers
```

```
integer function Power( 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
```

Base cases:

\[
\begin{align*}
\text{if } e = 0 & : n^0 = 1 \\
\text{if } e = 1 & : n^1 = n \\
\end{align*}
\]

\[
\frac{3^4}{3^3} = 3 \times \frac{3^3}{3^3} = 3 \times 3^1 = 3^2
\]
MIPS Assembly Language Supplement (Section 4.14 of the textbook)

The figure below shows a trace of the run-time stack after CalculatePowers called Power when num = 1 and pow = 1. When Power is called with n = 1 and e = 1, a base case of the recursion assigns result to 1 and then returns. The circled numbers indicate the order of events in the trace.

The figure below shows a trace of the run-time stack after CalculatePowers called Power when num = 3 and pow = 4. Notice that Power follows the same rules as any other subprogram with respect to parameter passing and the run-time stack. The circled numbers indicate the order of events in the trace.
MIP Calling-Conventions Supplement 2 for Section 4.14 of the textbook

If you did want to speed up a high-level program by using assembly language, you would compile the program with a profiling option, and then run the program with real data. Having profiling turned on causes the program to track where it spends its execution time, and generates a report of the program's profile. Usually, over 85% of a program's time is spent executing a single subprogram. Thus, you can write just this subprogram in assembly language and leave the rest of the program in the high-level language (HLL). To correctly have the HLL program call your assembly-language subprogram, your assembly-language subprogram must follow the run-time stack and register conventions established for the processor. The register conventions are the rules about how the registers should be used.

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

Different machines use one of several 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)

The following table shows the MIPS register conventions. Each register can be referenced to by its number or its convention name, e.g., $4 as $s0 for an "argument"/parameter register. The caller of the subprogram would place the parameter value in $s0 and call the subprogram. The subprogram has the parameter value since it is in the register. Thus, avoiding pushing it on the run-time stack in the slow memory.

<table>
<thead>
<tr>
<th>Reg. #</th>
<th>Convention Name</th>
<th>Role in Procedure Calls</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0</td>
<td>$zero</td>
<td>constant value zero</td>
<td>Cannot be changed</td>
</tr>
<tr>
<td>$1</td>
<td>$at</td>
<td>Used by assembler to implement pseudoinstructions</td>
<td>DON'T USE</td>
</tr>
<tr>
<td>$2, $3</td>
<td>$v0, $v1</td>
<td>Results of a function</td>
<td></td>
</tr>
<tr>
<td>$4 - $7</td>
<td>$a0 - $a3</td>
<td>First 4 arguments to a procedure</td>
<td></td>
</tr>
<tr>
<td>$8 - $15, $24, $25</td>
<td>$t0 - $t9</td>
<td>Temporary registers (not preserved across call)</td>
<td>Caller-saved registers - subprogram can use them as scratch registers, but it must also save any needed values before calling another subprogram.</td>
</tr>
<tr>
<td>$16 - $23</td>
<td>$s0 - $s7</td>
<td>Saved temporary (preserved across call)</td>
<td>Callee-saved registers - it can rely on a 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)</td>
</tr>
<tr>
<td>$26, $27</td>
<td>Sk0, Sk1</td>
<td>Reserved for the Operating System Kernel</td>
<td>DON'T USE</td>
</tr>
<tr>
<td>$28</td>
<td>$gp</td>
<td>Pointer to global area</td>
<td></td>
</tr>
<tr>
<td>$29</td>
<td>$sp</td>
<td>Stack pointer</td>
<td>Points to first free memory location above stack</td>
</tr>
<tr>
<td>$30</td>
<td>$fp/$s8</td>
<td>Frame pointer (if needed) or another saved register</td>
<td>$fp not used so use as $s8</td>
</tr>
<tr>
<td>$31</td>
<td>$ra</td>
<td>Return address (used by a procedure call)</td>
<td>Receives return addr. on jal call to procedure</td>
</tr>
</tbody>
</table>
MIP Calling-Conventions Supplement 2 for Section 4.14 of the textbook

The general steps for using the MIPS register conventions are listed below.

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

Let's reconsider the CalculatePowers program and examine how to apply the MIPS register conventions:

### High-level Language Programmer's View of CalculatePowers

```plaintext
main:
maxNum = 3
maxPower = 4
CalculatePowers(maxNum, maxPower) (*)
...
end main
```

```plaintext
CalculatePowers( 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
```

```plaintext
integer function Power( 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
```

I would recommend asking yourself the following questions to determine which registers to use when applying the register conventions.

1) Using the MIPS register conventions, what registers would be used to pass the parameters (maxNum and maxPower) to CalculatePowers?

The first parameter is always passed in $a0, the second parameter in $a1, etc. If there are more than four parameters, then additional parameters...
are pushed onto the run-time stack. Thus, we'll use the following register allocation:

<table>
<thead>
<tr>
<th>maxNum</th>
<th>maxPower</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a0</td>
<td>$a1</td>
</tr>
</tbody>
</table>

The main program code that calls CalculatePowers would be:

```assembly
main:
    ...
    lw  $a0, maxNum    # $a0 contains maxNum
    lw  $a1, maxPower  # $a1 contains maxPower
    jal CalculatePowers
```

The jump-and-link (jal) instruction acts an unconditional jump instruction, but it also saves the address of the instruction after the jal to register $ra so execution can return there when CalculatePowers returns. Having the return-address saved to a register avoids the need to save it to the run-time stack.

When CalculatePowers starts execution its formal parameters, numLimit and powerLimit, will be in registers $a0 and $a1, respectively. Since CalculatePowers calls the Power function which takes two parameters, both register $a0 and $a1 must eventually be used for this purpose. You want to decide if either numLimit or powerLimit is needed across the call to Power. If so, we must save their value(s) before calling Power. One way to save their value is to move them into an s-register which is maintained across the call to a subprogram (i.e., the subprogram will not change the s-registers if it is following the register conventions). In writing the code for a subprogram, the second question I ask myself is:

2) 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")

The call to Power is part of the inner-for-loop body with numLimit and powerLimit both being needed after the call (so they can be compared to the loop control variables). Thus, both should be saved to s-registers: $a0 can be moved to $s0 and $a1 can be moved to $s1.

For the local variables, num and pow, we need to ask ourselves a similar question:

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

Since both variables are used as loop-control variables with Power being called as part of the inner-for-loop body, both variable must maintain their value across the call to Power. Thus, s-registers should be used for both. Since $s0 and $s1 are already being used for numLimit and powerLimit, we can use $s2 and $s3 as:

<table>
<thead>
<tr>
<th>num</th>
<th>pow</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s2</td>
<td>$s3</td>
</tr>
</tbody>
</table>
MIP Calling-Conventions Supplement 2 for Section 4.14 of the textbook

Before we can use $s0 to $s3 in CalculatePowers, we need to save their values to the run-time stack. After all, "main" might be storing something in these s-registers, and by convention CalculatePowers should not be allowed to change "main's" values in s-registers. The code that starts the CalculatePowers subprogram would be:

```
CalculatePowers:  # parameters:  $a0 contains numLimit, and $a1 contains powerLimit

sub  $sp, $sp, -20  # move stack pointer, $sp, "up" to make room for the call-frame
sw   $ra, 4($sp)   # push return address onto stack
sw   $s0, 8($sp)   # push the caller's s-register values onto the stack
sw   $s1, 12($sp)
sw   $s2, 16($sp)
sw   $s3, 20($sp)  # save a-registers to s-registers so they don't get wiped out
move $s0, $a0      # save numLimit in $s0
move $s1, $a1      # save powerLimit in $s1
```

To create room on the run-time stack for CalculatePowers' call-frame, we subtract 20 bytes from the stack-pointer, $sp register, which is enough for 5 registers. The registers saved are the s-registers $s0 to $s3 and the $ra register which contains the return address back in the "main." We save the $ra on the stack since the "jal  Power" instructions in the subprogram body would wipe it out.

Since CalculatePowers calls Power, we start over with the same set of questions for the call to Power and the code for Power.

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

<table>
<thead>
<tr>
<th>num</th>
<th>pow</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a0</td>
<td>$a1</td>
</tr>
</tbody>
</table>

The CalculatePowers' code that calls Power would be:

```
CalculatePowers:

...  
move $a0, $s2  # call Power(num, pow), where num is in $s2 and pow is in $s3
move $a1, $s3
jal  Power
...  
```

Writing the Power function is a little trick. Since it is recursive and it only calls itself, we can "bend the register conventions" a little bit to improve efficiency.

2) Using the MIPS register conventions, which of these parameters ("n", "e", or both of them) should be moved into s-registers?
Normally, I'd look at the recursive call "result = Power(n, e - 1)* n", and think that parameter n is needed after we return from Power so we can multiple it. Thus, I'd want to move the original parameter n in $a0 to an s-register, so the call to Power does not wipe it out. However, the first parameter in the recursive call is the value of n, so we can just leave $a0 as the value of n throughout subprogram Power. If this is confusing, you might look at the HLL run-time stack diagram on page 6 of the “MIPS Supplement”. Notice that n's value is unchanged in each call-frame.

The parameter e is initially in $a1. Since e's value is not needed after the recursive call to Power, it does not need to be saved to an s-register.

3) Using the MIPS register conventions, what register should be used for the local variable "result"?

Since "result" has no value before the recursive call to Power, we don't need to use a s-register. The value of "result" is returned as the function value so using $v0 makes the most sense. The flow-chart of the Power function clearly shows that "result" does not have a value before the recursive call. Suppose e = 1, then the "result = n" assignment statement would be executed, but the recursive call would not be performed.

```
HLL Power code:
integer function Power( 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
```
MIP Calling-Conventions Supplement 2 for Section 4.14 of the textbook

The Power function in MIPS assembly language using the above decisions is given below. Since neither parameter nor the local variable "result" needs to be saved to an s-register, the only thing to save on the run-time stack is the return-address register, $ra.

Power:

```
# $a0 contains n (we never change it during the recursive calls so we don't need to save it)
# $a1 contains e
sub $sp, $sp, -4  # make room for the call-frame
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:
    addi $a1, $a1, -1  # first parameter is still n in $a0
    jal Power  # put second parameter, e-1, in $a1
    mul $v0, $v0, $a0  # returns with value of Power(n, e-1) in $v0
end_if:
    lw $ra, 4($sp)  # restore return addr. to $ra
    addi $sp, $sp, 4  # pop call frame from stack
    jr $ra

end_Power:
```

Notice at the end of Power, we must
- restore the saved register(s) (here only the $ra register),
- restore the stack pointer to its original position before the call, and
- jump-register (jr $ra) back to the return address in register $ra.

Complete code for the above example, including printing of the output, is included at the end of this Supplement. Before we look at this code, let's practice using the register conventions on the following Insertion sort example.

Insertion sort is a simple sort. All simple sorts consist of two nested loops where:
- the outer loop keeps track of the dividing line between the sorted and unsorted part with the sorted part growing by one in size each iteration of the outer loop.
- the inner loop's job is to do the work to extend the sorted part's size by one.
After several iterations of the outer loop, an array might look like:

<table>
<thead>
<tr>
<th>Sorted Part</th>
<th>Unsorted Part</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5</td>
<td>6 7 8</td>
</tr>
<tr>
<td>10 20 35 40</td>
<td>45 60 25 50 90</td>
</tr>
</tbody>
</table>

Insertion sort takes the "first unsorted element" (25 at index 6 in the above example) and "inserts" it into the sorted part of the list "at the correct spot." After 25 is inserted into the sorted part, the array would look like:

<table>
<thead>
<tr>
<th>Sorted Part</th>
<th>Unsorted Part</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5</td>
<td>6 7 8</td>
</tr>
<tr>
<td>10 20 25 35</td>
<td>40 45 60 50 90</td>
</tr>
</tbody>
</table>

The below code, splits off the inner-loop into its own Insert subprogram. The array is passed by sending its starting/base address as a parameter.

```
main:
  integer scores [100];
  integer n; // # of elements
  InsertionSort(scores, n)

  (*)
  ...

end main

InsertionSort(numbers - address to integer array, 
               length - integer)
  integer firstUnsortedIndex
  for firstUnsortedIndex = 1 to (length-1) do
    Insert(numbers, numbers[firstUnsortedIndex], 
            firstUnsortedIndex-1);
  end for

end InsertionSort

Insert(numbers - address to integer array, 
        elementToInsert - integer, 
        lastSortedIndex - integer) {
  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
```
MIP Calling-Conventions Supplement 2 for Section 4.14 of the textbook

Try to apply the MIPS register conventions by answering the "standard" questions. On the next page, I'll supply my answers to these questions.

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

<table>
<thead>
<tr>
<th>scores</th>
<th>n</th>
</tr>
</thead>
</table>

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

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

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

<table>
<thead>
<tr>
<th>numbers</th>
<th>numbers[firstUnsortedIndex]</th>
<th>firstUnsortedIndex-1</th>
</tr>
</thead>
</table>

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

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

Try to write the code for main, InsertionSort, and Insert in MIPS assembly language.
MIP Calling-Conventions Supplement 2 for Section 4.14 of the textbook

My answers to these questions:

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

<table>
<thead>
<tr>
<th>scores</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a0</td>
<td>$a1</td>
</tr>
</tbody>
</table>

Because the first parameter always is passed in $a0, and the second parameter is always passed in $a1. The array scores is passed by sending the base address in $a0.

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

Both numbers and (length-1) need to be stored in s-registers since their values are needed each iteration of the for-loop, i.e., their values are needed across the call to Insert. We’ll use $s0 for "numbers" and $s1 for the value of (length-1).

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

The local variable firstUnsortedIndex needs to be stored in an s-register since its value is needed each iteration of the for-loop, i.e., its value is needed across the call to Insert. We’ll use $s2 for "firstUnsortedIndex".

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

<table>
<thead>
<tr>
<th>numbers</th>
<th>numbers[firstUnsortedIndex]</th>
<th>firstUnsortedIndex-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a0</td>
<td>$a1</td>
<td>$s2</td>
</tr>
</tbody>
</table>

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

Since Insert does not call any subprograms, we can leave the parameters in the a-registers.

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

Since Insert does not call any subprograms, we can use a "temporary" register, say $t0 for testIndex. Thus, Insert does not need to store anything on the run-time stack making it very efficient. The beauty of a combined caller-saved with callee-saved register convention.
main:

integer scores [100];
integer n; // # of elements

InsertionSort(scores, n)

(*)
...

end main

Insert(numbers - address to integer array, elementToInsert - integer, lastSortedIndex - integer) { 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

The code for main, InsertionSort, and Insert in MIPS assembly language is given below.

.data
scores: .word 20, 30, 10, 40, 50, 60, 30, 25, 10, 50
n: .word 10

.text
.globl main
main:
la $a0, scores
lw $a1, n
jal insertionSort
li $v0, 10
syscall

insertionSort:
sub $sp, $sp, 16 # setup call-frame
sw $ra, 4($sp) # save the return addr to get back to main
sw $s0, 8($sp) # save main's s-registers if it is
sw $s1, 12($sp) # using any
sw $s2, 16($sp)
move $s0, $a0 # save address of numbers in $s0
sub $s1, $a1, 1 # save (length-1) in $s1
for_init:
  li $s2, 1 # save firstUnsortedIndex in $s2
for_loop:
  bgt $s2, $s1, end_for
move $a0, $s0 # fill actual parameters
MIP Calling-Conventions Supplement 2 for Section 4.14 of the textbook

```
mul $t0, $s2, 4
add $t0, $s0, $t0
lw $a1, 0($t0)
sub $a2, $s2, 1
jal insert
  # call insert subpgm
add $s2, $s2, 1
j for_loop
end_for:
  lw $ra, 4($sp)
  # restore return addr. and
lw $s0, 0($sp)
  # restore main's s-registers
lw $s1, 12($sp)
lw $s2, 16($sp)
addi $sp, $sp, 16
  # remove call-frame
jr $ra
  # 'jump register' back to main
end_insertionSort:

insert:  # insert does not call any subpgms so we'll use $a and $t registers
  # nothing to save on the run-time stack
  move $t0, $a2
  # use $t0 for testIndex
while:
  blt $t0, 0, end_while
  mul $t1, $t0, 4
  add $t1, $a0, $t1
  lw $t2, 0($t1)
  ble $t2, $a1, end_while
  sw $t2, 4($t1)
  sub $t0, $t0, 1
  j while
end_while:
  mul $t1, $t0, 4
  add $t1, $a0, $t1
  sw $a1, 4($t1)
  jr $ra
  # 'jump register' back to insertionSort
end_insert:
```

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 accidently 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.

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

Below is the complete CalculatePowers program example using MIPS register conventions and PCSpim syscalls. Notice that strings use the "asciiz" assembler directive to generate an array of ASCII characters that are "null" terminated (ASCII value 0).
# 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
lw $a1, maxPower
jal CalculatePowers

li $v0, 10
syscall

CalculatePowers:

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

sub $sp, $sp, -20
sw $ra, 4($sp)
sw $s0, 8($sp)
sw $s1, 12($sp)
sw $s2, 16($sp)
sw $s3, 20($sp)

move $s0, $a0
move $s1, $a1

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
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

sub $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:
   addi $a1, $a1, -1 # first parameter is still n in $a0
   jal Power # put second parameter, e-1, in $a1
   mul $v0, $v0, $a0 # returns with value of Power(n, e-1) in $v0
   # 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:

```
1 raised to 2 power is 1
1 raised to 3 power is 1
1 raised to 4 power is 1
2 raised to 1 power is 2
2 raised to 2 power is 4
2 raised to 3 power is 8
2 raised to 4 power is 16
3 raised to 1 power is 3
3 raised to 2 power is 9
3 raised to 3 power is 27
3 raised to 4 power is 81
```
main:

integer scores [100];
integer n; // # of elements

InsertionSort(scores, n)

end main

InsertionSort(numbers - address to integer array,
length - integer)

integer firstUnsortedIndex
for firstUnsortedIndex = 1 to (length-1) do
    Insert(numbers, numbers[firstUnsortedIndex],
            firstUnsortedIndex-1);
end for

end InsertionSort

Insert(numbers - address to integer array,
elementToInsert - integer,
lastSortedIndex - integer) {
    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:

<table>
<thead>
<tr>
<th>scores</th>
<th>n</th>
</tr>
</thead>
</table>

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:

<table>
<thead>
<tr>
<th>numbers</th>
<th>numbers[firstUnsortedIndex]</th>
<th>firstUnsortedIndex-1</th>
</tr>
</thead>
</table>

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.
Lecture 20  Practice MIPS calling

n = 0, 1, 2, 3, 4, 5, 6
fibonacci: 0, 1, 1, 2, 3, 5, 8, 13, 21...

Main:
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)

result = fib(5) jal fib

"qadr" of next instr. in main in draq
#define
"base cases"
defib(n) = \begin{cases}
0 & \text{if } n = 0 \\
1 & \text{if } n = 1 \\
fib(n-1) + fib(n-2) & \text{if } n \geq 2
\end{cases}

if n == 0 then
    li $v0, 0
    jr $ra
else if n == 1 then
    li $v0, 1
    jr $ra
else
    li $v0, fib(n-1)
    jr $ra
    li $v0, fib(n-2)
    jr $ra

Pre-lec. 20-1
data
reslt: word 0
.globl main
main:
    li $a0,5  # n is 5
    jal fib
    sw $v0, reslt

    li $v0, 10
    syscall
fib:  # $a0 is n
    if: bne $a0, 0, else-if
        li $v0, 0
        jr $ra
    else-if:
        bne $a0, 1, else
        li $v0, 1
        jr $ra
    else:
    # Code for else case
    jr $ra
else:
  addi $sp, $sp, -12
  sw $ra, 4($sp)
  sw $s0, 8($sp)
  sw $s1, 12($sp)
  move $s0, $a0
  addi $a0, $s0, -1
  jal fib
  move $s1, $v0
  addi $v0, $s1, $v0
  lw $ra, 4($sp)
  lw $s0, 8($sp)
  lw $s1, 12($sp)
  addi $sp, $sp, 12
  jr $ra
# Recursive fibonacci example

.data
result: .word 0

.text
.globl main
main:

    # Fibonacci seq.: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...
    li    $a0, 7          # fib(7) is 13 in decimal or D in hex.
    jal   fib
    sw    $v0, result

    li    $v0, 10
    syscall

fib:

    # $a0 is n

if:
    bne  $a0, 0, else_if
    li    $v0, 0
    jr    $ra

else_if:
    bne  $a0, 1, else
    li    $v0, 1
    jr    $ra

else:
    addi  $sp, $sp, -12
    sw    $ra, 4($sp)   # save return addr.
    sw    $s0, 8($sp)   # save n in $s0
    sw    $s1, 12($sp)  # save result of fib(n-1) in $s1
    move  $s0, $a0

    addi  $a0, $s0, -1
    jal   fib
    move  $s1, $v0

    addi  $a0, $s0, -2
    jal   fib
    add   $v0, $s1, $v0  # calculate fib(n-1) + fib(n-2)

    lw    $ra, 4($sp)   # restore return addr.
    lw    $s0, 8($sp)   # restore $s0
    lw    $s1, 12($sp)  # restore $s1
    addi  $sp, $sp, 12  # "pop" call frame
    jr    $ra
1. If $5$ contains $0xAF00A5D$ and $6$ contained $0x6$, what hexadecimal value would be in $4$ after each of the following?
   a) slt $4$, $5$, $3$
   b) sltv $4$, $5$, $6$
   c) sra $4$, $5$, $3$
   d) ror $4$, $5$, $6$

2. If $5$ contains $0xA5D$ and $6$ contained $0x63C$, what hexadecimal value would be in $4$ after each of the following?
   a) and $4$, $5$, $6$
   b) ori $4$, $5$, $0xBF$
   c) xor $4$, $5$, $6$
   d) nor $4$, $5$, $6$
   e) not $4$, $5$

3. 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:

    \[
    \begin{array}{cccccccc}
    \text{unused} & Z & Y & X & \ldots & E & D & C & B & A \\
    \text{bit position:} & 25 & 24 & 23 & & 4 & 3 & 2 & 1 & 0 \\
    \end{array}
    \]

    in $5$

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.
   a) What instruction(s) could we use to build the mask needed for ‘C’ in $3$?

   b) If a bit-string set of letters is in register $5$, then what instruction(s) can be used to check if the character ‘C’ (using the mask in $3$) is in the set contained in $5$?

   c) If a bit-string set of letters is in register $5$ and another bit-string set of letters is in $6$, then what instruction(s) can be used to calculate in $7$ the union of sets in $5$ and $6$? (i.e., all elements in either set)

   d) If a bit-string set of letters is in register $5$ and another bit-string set of letters is in $6$, then what instruction(s) can be used to calculate in $7$ the intersection of sets in $5$ and $6$? (i.e., all elements in both sets)

   e) If a bit-string set of letters is in register $5$ and another bit-string set of letters is in $6$, then what instruction(s) can be used to calculate in $7 the set difference: set in $5$ - set in $6$? (i.e., all elements in the left-hand set that are not in the right-hand set)
4. Write MIPS code for the set-of-letters abstract-data type (ADT) using a bit string. The bit string representation for the set of letters can use a 32-bit word with the least-significant bit associated with the letter 'A', etc.

\[
\begin{array}{ccccccc}
\text{ unused } & \text{ 'Z' } & \text{ 'Y' } & \text{ 'X' } & \ldots & \text{ 'E' } & \text{ 'D' } & \text{ 'C' } & \text{ 'B' } & \text{ 'A' } \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\
\end{array}
\]

bit position: 25 24 23 4 3 2 1 0

The set of letters ADT should have the following operations as subprograms, so use appropriate register conventions:

<table>
<thead>
<tr>
<th>Subprogram Name</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bitString</td>
<td>pass in a pointer to the an .ASCIIZ string returns a word containing the set of letters as a bitString</td>
<td>Returns a bit string corresponding to the set of letters in the .ASCIIZ string. Non-letter characters are ignored, and both upper and lower-case letters should be represented as letters in the set.</td>
</tr>
<tr>
<td>union</td>
<td>passed two set bitStrings returns the set union of the two sets</td>
<td>The resulting set should contain the elements that are in one or both of the input sets.</td>
</tr>
<tr>
<td>intersection</td>
<td>passed two set bitStrings returns the set intersection of the two sets</td>
<td>The resulting set should contain the elements that are in both of the input sets.</td>
</tr>
<tr>
<td>difference</td>
<td>passed two set bitStrings returns the set difference of the first set - second set</td>
<td>The resulting set should contain the elements that are in the first set, but not also in the second set.</td>
</tr>
<tr>
<td>contains</td>
<td>passed an .ASCII character and a set bitString returns a Boolean (0 for false or 1 for true)</td>
<td>Returns 1 (true) if the .ASCII character is in the bitString set; otherwise return 0 (false).</td>
</tr>
<tr>
<td>print</td>
<td>passed an set bitString</td>
<td>Prints the bitString to the console using print_string system calls. The set should be printed in the conventional format, i.e., &quot;( E, G, T, Y )&quot;</td>
</tr>
</tbody>
</table>
Lecture 8128 PC Spin I/O + MIPS Logical + Shift Instructions

11 $4, 0x B9, #4
11 $5, 0x 35, #5

and $6, $4, $5, #6

or $7, $4, $5

logical-shift-left

$8, $4, 2

rot $8, $4, 2

ror $9, $4, 2

0x 40000002 E16

0x 8421 2 E 4 16

0100 00 0010 1110

0x 0011 0001

0x 0011 0001

0x B D 16

0x 0010 1110 0100

0x 0011 1101

0x 0011 1101

0x 0011 1101
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.

<table>
<thead>
<tr>
<th>Service Requested</th>
<th>System call code passed in $v0</th>
<th>Registers used to pass additional arguments</th>
<th>Registers used to return results</th>
</tr>
</thead>
<tbody>
<tr>
<td>print_int</td>
<td>1</td>
<td>$a0 contains the integer value to print</td>
<td></td>
</tr>
<tr>
<td>print_float</td>
<td>2</td>
<td>$f12 contains the 32-bit float to print</td>
<td></td>
</tr>
<tr>
<td>print_double</td>
<td>3</td>
<td>$f12 (and $f13) contains the 64-bit double to print</td>
<td></td>
</tr>
<tr>
<td>print_string</td>
<td>4</td>
<td>$a0 contains the address of the .asciiz string to print</td>
<td></td>
</tr>
<tr>
<td>read_int</td>
<td>5</td>
<td>$v0 returns the integer value read</td>
<td></td>
</tr>
<tr>
<td>read_float</td>
<td>6</td>
<td>$f0 returns the 32-bit floating-point value read</td>
<td></td>
</tr>
<tr>
<td>read_double</td>
<td>7</td>
<td>$f0 and $f1 returns the 64-bit floating-point value read</td>
<td></td>
</tr>
<tr>
<td>read_string</td>
<td>8</td>
<td>$a0 contains the address of the buffer to store the string $a1 contains the maximum length of the buffer</td>
<td></td>
</tr>
<tr>
<td>sbrk - request a memory block</td>
<td>9</td>
<td>$a0 contains the number of bytes in the requested block</td>
<td>$v0 returns the starting address of the block of memory</td>
</tr>
</tbody>
</table>
Write the code for main, CalculatePowers, and Power in MIPS assembly language.

<table>
<thead>
<tr>
<th>pow</th>
<th>num</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Using the MIPS register conventions, write registers should be used for the local variables:

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

<table>
<thead>
<tr>
<th>maxPower</th>
<th>maxNum</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>end Power</th>
<th>end for num</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(6) Use a register for any value you will need after you come back from a subprogram/function/procedure call, e.g., call to "Power()"

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

Always read each of the following parameters to CalculatePowers:

end main

...
# 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"

.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
li $v0, 4
syscall

move $a0, $s3
li $v0, 1
syscall

la $a0, str2
li $v0, 4
syscall

move $a0, $s2
move $a1, $s3
jal Power

move $a0, $v0
syscall

li $v0, 1
syscall

la $a0, str3
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)

lw $s0, 8($sp)

lw $s1, 12($sp)

lw $s2, 16($sp)

lw $s3, 20($sp)

addi $sp, $sp, 20

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:
    addi $a1, $a1, -1  # first parameter is still n in $a0
    jal Power  # put second parameter, e-1, in $a1
    mul $v0, $v0, $a0  # returns with value of Power(n, e-1) in $v0
end_if:
    lw $ra, 4($sp)  # restore return addr. to $ra
    addi $sp, $sp, 4  # pop call frame from stack
end_Power:

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

1 raised to 2 power is 1
1 raised to 3 power is 1
1 raised to 4 power is 1
2 raised to 1 power is 2
2 raised to 2 power is 4
2 raised to 3 power is 8
2 raised to 4 power is 16
3 raised to 1 power is 3
3 raised to 2 power is 9
3 raised to 3 power is 27
3 raised to 4 power is 81
### MIPS Logical Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>and $4, $5, $6</td>
<td>$4\leftarrow$ $4$ AND $5$</td>
</tr>
<tr>
<td>andi $4, $5, 0x5f</td>
<td>$4\leftarrow$ $4$ AND $5_{16}$</td>
</tr>
<tr>
<td>or $4, $5, $6</td>
<td>$4\leftarrow$ $4$ OR $5$</td>
</tr>
<tr>
<td>ori $4, $5, 0x5f</td>
<td>$4\leftarrow$ $4$ OR $5_{16}$</td>
</tr>
<tr>
<td>xor $4, $5, $6</td>
<td>$4\leftarrow$ $4$ XOR $5$</td>
</tr>
<tr>
<td>xori $4, $5, 0x5f</td>
<td>$4\leftarrow$ $4$ XOR $5_{16}$</td>
</tr>
<tr>
<td>nor $4, $5, $6</td>
<td>$4\leftarrow$ $4$ NOT $5_{16}$</td>
</tr>
<tr>
<td>not $4, $5</td>
<td>$4\leftarrow$ NOT $5$ #inverts all the bits</td>
</tr>
</tbody>
</table>

### MIPS Shift and Rotate Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sll $4, $5, 3</td>
<td>$4\leftarrow$ shift left $5$ by 3 positions. Shift in zeros (only least significant 5-bits of immediate value are used to shift)</td>
</tr>
<tr>
<td>sllv $4, $5, $6</td>
<td>Similar to sll, but least significant 5-bits of $6$ determine the amount to shift.</td>
</tr>
<tr>
<td>srl $4, $5, 3</td>
<td>$4\leftarrow$ shift right $5$ by 3 positions. Shift in zeros</td>
</tr>
<tr>
<td>srlv $4, $5, $6</td>
<td>Similar to srl, but least significant 5-bits of $6$ determine the amount to shift.</td>
</tr>
<tr>
<td>sra $4, $5, 3</td>
<td>$4\leftarrow$ shift right $5$ by 3 positions. Sign-extend (shift in sign bit)</td>
</tr>
<tr>
<td>srav $4, $5, $6</td>
<td>Similar to sra, but least significant 5-bits of $6$ determine the amount to shift.</td>
</tr>
<tr>
<td>rol $4, $5, 3</td>
<td>$4\leftarrow$ rotate left $5$ by 3 positions</td>
</tr>
<tr>
<td>rol $4, $5, $6</td>
<td>Similar to above, but least significant 5-bits of $6$ determine the amount to rotate.</td>
</tr>
<tr>
<td>ror $4, $5, 3</td>
<td>$4\leftarrow$ rotate right $5$ by 3 positions</td>
</tr>
<tr>
<td>rotr $4, $5, $6</td>
<td>Similar to above, but least significant 5-bits of $6$ determine the amount to rotate.</td>
</tr>
</tbody>
</table>

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 \times 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: 0 0 0 0 0 0 0 0
   Bit position: 25 24 23 22
   Column: Z Y X . . .
<table>
<thead>
<tr>
<th>E</th>
<th>D</th>
<th>C</th>
<th>B</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
   ```

   To determine if a specific ASCII character, say 'C' (67₁₀) 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.