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

   if $X < Y$ then
       min = $X$
   else
       min = $Y$
   end if

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

   a) for $S4 = 0$ to $100$ by steps of size $10$ do
       if ($S3 < S4$) AND ($S2 >= 50$) then
           $S2 = S2 + S3$
       end if
   end for

   b) while ($S8 > 20$) do
       if ($S8 < 100$) OR ($S8 > 200$) then
           $S7 = S8$
           $S8 = S8 - 10$
       else
           $S8 = S8 - S7$
       end if
       $S7 = S6 + 4$
   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
```
Lecture 16: MIPS Introduction

RISC Arch. (CISC - Intel x86)

Reg. File 32 reg. 32-bit

Load/Store machine

\[ \text{add } \#2, \#3, \#4 \Rightarrow \#2 \leftarrow \#3 + \#4 \]

\[ \text{lw } \#2, \#3, \#4 \]

\[ \text{sw } \#2, \text{result} \]

4GB (2^{32}) 32-bit addrs.

X

Y

Result

\[ 2^{32} \]

2-1

Don't Use

\$2 - \$25

Use

\$2 - \$25
LLL: if $X < Y$ then
    "then body"
else
    "else body"
end if

for I = 1 to 10 do
    I = 1
    I <= 10? False
    True
    body
    I = I + 1
end for

LLL:
lookup $2, X$
lookup $3, Y$
bge $2, $3, else
  "then body"
else
  "else body"
end if
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 pseudo-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:

```assembly
.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 (use $8 for sum)
#   sum := sum + array[i] (use $10 for length-1)
# end for (use $11 for base addr. of array)

.text
.globl main

main:
   li $8, 0  # load immediate 0 in reg. $8 (sum)
   la $11, array  # load base addr. of array into $11
   for:
      lw $10, length  # load length in reg. $10
      addi $10, $10, -1  # $10 = length - 1
      li $9, 0  # initialize i in $9 to 0
      for_compare:
         bgt $9, $10, end_for  # drop out of loop when i > (length-1)
         mul $12, $9, 4  # mult. i by 4 to get offset within array
         add $12, $11, $12  # add base addr. of array to $12 to get addr. of array[i]
         lw $12, 0($12)  # load value of array[i] from memory into $12
         add $8, $8, $12  # update sum
         addi $9, $9, 1  # increment i
         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>and $4, $5, $6</td>
<td>$4←$5 (bit-wise AND) $6</td>
</tr>
<tr>
<td>andi $4, $5, 0x5f</td>
<td>$4←$5 (bit-wise AND) 5f₁₆</td>
</tr>
<tr>
<td>or $4, $5, $6</td>
<td>$4←$5 (bit-wise OR) $6</td>
</tr>
<tr>
<td>ori $4, $5, 0x5f</td>
<td>$4←$5 (bit-wise OR) 5f₁₆</td>
</tr>
<tr>
<td>xor $4, $5, $6</td>
<td>$4←$5 (bit-wise Exclusive-OR) $6</td>
</tr>
<tr>
<td>xori $4, $5, 0x5f</td>
<td>$4←$5 (bit-wise Exclusive-OR) 5f₁₆</td>
</tr>
<tr>
<td>nor $4, $5, $6</td>
<td>$4←$5 (bit-wise NOR) $6</td>
</tr>
<tr>
<td>not $4, $5</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>sll $4, $5, 3</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>slvv $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←shift right $5 by 3 positions. Shift in zeros</td>
</tr>
<tr>
<td>slrv $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←shift right $5 by 3 positions. Sign-extend (shift in sign bit)</td>
</tr>
<tr>
<td>sraw $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←rotate left $5 by 3 positions</td>
</tr>
<tr>
<td>ro $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←rotate right $5 by 3 positions</td>
</tr>
<tr>
<td>ror $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 * 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 # 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:

   ![Bit-string representation](image)

   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.
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:

```
HLL statement | Flow chart | Assembly Language
if X < Y then  |
               | X < Y?    |
               | False    |
               | True     |
               | then body|
else          |           |
               |           |
               | else body|
end if        |           |
```

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:

```
HLL statement | Flow chart | Assembly Language
for I = 1 to 10 do |
               | I = 1     |
               | I ≤ 10?   |
               | False    |
               | True     |
               | for body  |
               | I = I + 1 |
end for        |           |
```

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 $I \leq 10$ is False, then we want to drop out of the loop. Since $I \leq 10$ 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.
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 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 $a1 contains the maximum length of the buffer</td>
<td></td>
</tr>
<tr>
<td>sbrk - request a memory block</td>
<td>9</td>
<td>$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>

MIPS program to input an integer, square it, and print the result

```
data
.prompt: .asciiz "Enter a number: 
.label: .asciiz "\nThe number squared is: 

.text
.globl main
main: la $a0, prompt  #print prompt
li $v0, 4
syscall
li $v0, 5  # number read into $v0
syscall
mul $t0, $v0, $v0  # squared value in $t0
la $a0, label  # print label
li $v0, 4
syscall
move $a0, $t0
li $v0, 1
syscall
li $v0, 10
syscall  # exit
```

![Console](Console.png)

Enter a number: 3
The number squared is: 9
1) Consider the following array scores:

<table>
<thead>
<tr>
<th>scores:</th>
<th>10</th>
<th>30</th>
<th>45</th>
<th>20</th>
<th>80</th>
<th>20</th>
<th>70</th>
<th>30</th>
<th>50</th>
</tr>
</thead>
<tbody>
<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 $S5$ contained the index $i$'s value, what MIPS instructions would calculate the address of scores[$i$] into register $S13$?

c) What MIPS instruction would load the value of scores[$i$] into register $S14$

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.

```mips
# 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: 5 10 20 30 40 60

ULL:

x = array[3]

array[3] = 23

RAM access O(1) "constant time"

"n" items in array

addr. of array[i] = (base addr. of array) + i * (size)

1a $11, array

hi is $9

mul $12, $9, 4

add $12, $11, $12

lw $14, 0($12)

 Lecture 17 ☞
**Bubble Sort** ascending sort smaller to larger

<table>
<thead>
<tr>
<th>numbers</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0</td>
<td>20</td>
<td>10</td>
<td>30</td>
<td>50</td>
<td>90</td>
<td>60</td>
<td>60</td>
</tr>
</tbody>
</table>

**Simple Sort**

```
outer loop - track the dividing line between the sorted and unsorted part of the array
inner loop - extend the sorted by one item
```

ILLI:

```
11. 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
```

Lecture 17-3
for_init_1: li $19, length
    add $8, $9, -1

for_compare_1: blt $8, $1, end-for_1

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

for_compare_2: bgt $10, $11, end-for_2

ifi:
    mul $13, $10, 4
    add $13, $12, $13 # addr, numbers[for]?
    lw $14, 0($13) # value numbers[for]?
    lw $15, 4($13) # value numbers[for]?
    ble $14, $15, end-if
    sw $14, 4($13)
    sw $15, 0($13)
end-if:
    add $10, $10, 1
    j for_compare_2
end_for_2:
    add $8, $8, -1
    j for_compare_1
end_for_1:
# 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
    add $13, $12, $13  # addr. of numbers[test]
    lw $14, 0($13)     # value of numbers[test]
    lw $15, 4($13)     # value of numbers[test+1]
bli $14, $15, end_if
    sw $14, 4($13)
    sw $15, 0($13)
end_if:
    addi $10, $10, 1

Page 1
j for_compare_2
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 Array 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:

\[
\text{address of } \text{array}[i] = \text{base address} + (i \times \text{element size in bytes})
\]

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:

```mips
.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).
MIPS Array Supplement (Section 4.14 of the textbook)

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

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

.text
.globl main

main:
    li  $8, 0   # load immediate 0 in reg. $8 (sum)
    la  $11, array # load base addr. of array into $11
for:
    lw  $10, length     # load length in reg. $10
    addi $10, $10, -1  # $10 = length - 1
    li  $9, 0  # initialize i in $9 to 0
for_compare:
    bgt $9, $10, end_for # drop out of loop when i > (length-1)
    mul $12, $9, 4       # mult. i by 4 to get offset within array
    add $12, $11, $12    # add base addr. of array to $12 to get addr. of array[i]
    lw  $12, 0($12)      # load value of array[i] from memory into $12
    add $8, $8, $12      # update sum
    addi $9, $9, 1       # increment i
    j for_compare
end_for:
    sw  $8, sum
    li  $v0, 10  # system code for exit
    syscall
```

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 of array access](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 Array Supplement (Section 4.14 of the textbook)

.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, 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

start
here

10 rows

20 columns

Row-major order

start here

10 rows

20 columns
```

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]

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

- address of M[r][c] = base address + r * size in a row + c * size of an element
- address of M[r][c] = base address + r * # of columns * size of an element + c * size of an element
- address of M[r][c] = base address + (r * # of columns + c) * size of an element
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 \times \# \text{ of columns}
    add $7, $7, $6      # calculates $r \times \# \text{ of columns} + c$
    mul $7, $7, 4       # calculates $(r \times \# \text{ of columns} + c) \times \text{size of an element}$
    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.
1) Using the idea of "walking pointers" modify 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.

```
# 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:
  lw $8, targetValue
  lw $6, length
  addi $6, $6, -1
  li $7, -1
  la $12, scores
  for_init:
    li $5, 0
  for_compare:
    bgt $5, $6, end_for
    if:
      mul $13, $5, 4
      add $13, $12, $13
      lw $14, 0($13)
      bne $14, $8, end_if
      move $7, $5
      j end_for
    end_if:
    addi $5, $5, 1
    j for_compare
  end_for:
  sw $7, foundIndex
  li $v0, 10
  syscall

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 +

b) Suppose M has 10 rows and 15 columns of 4-byte words. How many bytes would need to be added to "walk a pointer" down the diagonal (i.e., M[0][0], M[1][1], M[2][2], etc.)?

Column-major order

start here

rows


columns
3. Modify the MIPS Assembly Language insertion sort program below to use walking pointers.

```assembly
# Insertion sort High-Level Algorithm:
# 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

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

.text
.globl main
main:
    for_init:
        li    $7, 1
        lw    $10, length # calculate (length-1) in $10
        addi  $10, $10, -1
        la    $11, numbers
    for_compare:
        bgt   $7, $10, end_for
        add   $8, $7, -1
        mul   $12, $7, 4
        add   $12, $11, $12
        lw    $9, 0($12)
    while:
        blt   $8, 0, end_while
        mul   $12, $8, 4
        add   $12, $11, $12
        lw    $13, 0($12)
        ble   $13, $9, end_while
        sw    $13, 4($12)
        addi  $8, $8, -1
        j     while
    end_while:
        mul   $12, $8, 4
        add   $12, $11, $12
        sw    $9, 4($12)
        addi  $7, $7, 1
        j     for_compare
    end_for:
        li    $v0, 10 # exit system call
        syscall
```

Register usage:
- $7 = firstUnsortedIndex
- $8 = testIndex
- $9 = elementToInsert
- $10 = value of length-1
- $11 = base address of numbers
- $12 = addr. of numbers[testIndex]
- $13 = value of numbers[testIndex]
The MIPS code to access $M[r][c]$ where $M$ has 10 rows and 20 columns and is stored in row-major order:

```
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.
Comp. Org

Lecture 19

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?
main:

<table>
<thead>
<tr>
<th>CalculatePowers(\textbf{In}: integer numLimit, integer powerLimit)</th>
<th>integer \textbf{Power}( \textbf{In}: integer n, integer e)</th>
</tr>
</thead>
<tbody>
<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>CalculatePowers(maxNum, maxPower)</td>
<td>result = 1</td>
</tr>
</tbody>
</table>
| (*
| for num := 1 to numLimit do |
| for pow := 1 to powerLimit do |
| ... | else |
| end main | result = n |
| else |
| print num " raised to " pow " power is " |
| Power(num, pow) | end if |
| end for pow |
| end for num | return result |
| end CalculatePowers |
| \begin{align*} 
\textit{end Power} 
\end{align*} |

\begin{itemize}
\item[a)] \text{Using the MIPS register conventions, what registers would be used to pass each of the following parameters to CalculatePowers:}
\begin{itemize}
\item maxNum
\item maxPower
\end{itemize}
\item[b)] \text{Using the MIPS register conventions, which of these parameters ("numLimit", "powerLimit", or both of them) should be moved into s-registers? (\textbf{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")}
\item[c)] \text{Using the MIPS register conventions, what registers should be used for each of the local variables:}
\begin{itemize}
\item num
\item pow
\end{itemize}
\item[d)] \text{Using the MIPS register conventions, what registers would be used to pass each of the following parameters to Power:}
\begin{itemize}
\item num
\item pow
\end{itemize}
\item[e)] \text{Using the MIPS register conventions, which of these parameters ("n", "e", or both of them) should be moved into s-registers?}
\item[f)] \text{Using the MIPS register conventions, what register should be used for the local variable:}
\begin{itemize}
\item result
\end{itemize}
\item[g)] \text{Write the code for main, CalculatePowers, and Power in MIPS assembly language.}
\end{itemize}
Lecture 19

Run-time stack in HLL

Memory

"heap"

Subtask A

Subtask B

Task

Run-time stack

Task 1

Task 2

Task 3

"Call-frames"

Global data

. data

ML

. pgm

(much)

Pgm

Task 1

Task 2

Task 3

Subtask A

Subtask B
Most high-level programming languages (C, C++, Ada, Java, Python, etc.) enable programs to be written in small reusable sections of code call 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.

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:

<table>
<thead>
<tr>
<th>Just before A calls B</th>
<th>Just after A calls B</th>
<th>Just after B calls C</th>
<th>Just after C returns to B</th>
<th>Just after C returns to B</th>
</tr>
</thead>
<tbody>
<tr>
<td>A's call-frame</td>
<td>B's call-frame</td>
<td>C's call-frame</td>
<td>ret addr <strong>(*)</strong></td>
<td>ret addr <strong>(*)</strong></td>
</tr>
<tr>
<td>i: 3</td>
<td>s: 3</td>
<td>w: 2</td>
<td>A's call-frame</td>
<td>A's call-frame</td>
</tr>
<tr>
<td>j:</td>
<td>t:</td>
<td></td>
<td>i: 3</td>
<td>i: 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>j: 3</td>
<td>j: 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>19 returned</td>
<td>19 returned</td>
</tr>
</tbody>
</table>

start execution here

subprogram A():
local integer i, j
i = 3

integer function B(integer s):
local integer t
(*)
return t
19 returned
end function B

integer function C(integer w):
2 passed to w
return w * w * w
end function C

j = B(i)
print j

end subprogram A

execution ends here
MIPS Functions Supplement (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 $a0 for an "argument"/parameter register. The caller of the subprogram would place the parameter value in $a0 and call the subprogram. The subprogram has access to the parameter value since it is in a 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 psuedoinstructions</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 an subprogram it calls not to change them (so a subprogram wishing to use these registers must save them on entry and restore them before it exits)</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>
The general steps for using the MIPS register conventions are listed below.

<table>
<thead>
<tr>
<th>Caller Code (caller of the subprogram)</th>
<th>Callee Code (the subprogram itself)</th>
</tr>
</thead>
<tbody>
<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>. . . code for the callee</td>
</tr>
<tr>
<td>4) for functions, place result to be returned in $v0-$v1</td>
<td>5) restore any callee-saved registers ($s0 - $s7) from step (2) above</td>
</tr>
<tr>
<td>5) restore any callee-saved registers ($s0 - $s7) from step (2) above</td>
<td>6) restore $ra if it was saved on the stack in step (3)</td>
</tr>
<tr>
<td>6) restore $ra if it was saved on the stack in step (3)</td>
<td>7) pop stack frame by adding frame size to $sp</td>
</tr>
<tr>
<td>7) pop stack frame by adding frame size to $sp</td>
<td>8) return to caller by &quot;jr $ra&quot; instruction</td>
</tr>
</tbody>
</table>

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:**

\[
\begin{align*}
\text{maxNum} &= 3 \\
\text{maxPower} &= 4 \\
\text{CalculatePowers}(&\text{maxNum, maxPower}) \\
(*&) \\
\text{end main}
\end{align*}
\]

**CalculatePowers** (integer numLimit, integer powerLimit)

\[
\begin{align*}
\text{integer num, pow} \\
\text{for num := 1 to numLimit do} \\
\text{for pow := 1 to powerLimit do} \\
\text{print num } \text{" raised to } \text{" pow } \text{" power is } \text{"} \\
\text{Power(num, pow)} \\
\text{end for pow} \\
\text{end for num} \\
\text{end CalculatePowers}
\end{align*}
\]

**integer function Power** (integer n, integer e)

\[
\begin{align*}
\text{integer result} \\
\text{if e = 0 then} \\
\text{result = 1} \\
\text{else if e = 1 then} \\
\text{result = n} \\
\text{else} \\
\text{result = Power(n, e - 1) * n} \\
\text{end if} \\
\text{return result} \\
\text{end Power}
\end{align*}
\]
MIPS Functions 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.
MIPS Functions Supplement (Section 4.14 of the textbook)

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

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, 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>

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
MIPS Functions Supplement (Section 4.14 of the textbook)

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
  addi $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
MIPS Functions Supplement (Section 4.14 of the textbook)

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
```

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
    addi $sp, $sp, -4       # make room for the call-frame
    sw $ra, 4($sp)          # save $ra on stack

    ... (Body of Power function)
    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 below:

```mips
.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 powLimit
    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
```

MIPS Functions Supplement - Page 9
MIPS Functions Supplement (Section 4.14 of the textbook)

    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:

    # 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  # 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:

```
Console:
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
```
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 conventions

\[ n \to 0, 1, 2, 3, 4, 5, 6 \]

Fibonacci: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

Main:

\[
\text{result} = \text{fib}(5) \quad \text{jal fib}
\]

\[
\text{def fib}(n) = \begin{cases}  \text{0} & \text{if } n = 0 \\  \text{1} & \text{if } n = 1 \\  \text{fib}(n-1) + \text{fib}(n-2) & \text{if } n \geq 2 \end{cases}
\]

if \( n = 0 \) then

\[
\text{return } 0 \quad \text{jr $ra0}
\]

else if \( n = 1 \) then

\[
\text{return } 1 \quad \text{jr $ra0}
\]

else

\[
\text{return fib}(n-1) + \text{fib}(n-2) \quad \text{jr $ra0}
\]
.data
result: .word 0

.text
.globl main

main:
    li $a0, 5  # n is 5
    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:
else:
    addi $sp, $sp, -12
    sw  $ra, 4($sp)
    sw  $so, 8($sp)
    sw  $sl, 12($sp)
    move $so, $a0
    addi $a0, $so, -1
    jal  fib
    move $sl, $v0
    addi $a0, $sl, -2
    jal  fib
    add  $v0, $sl, $v0
    lw  $ra, 4($sp)
    lw  $so, 8($sp)
    lw  $sl, 12($sp)
    addi $sp, $sp, 12
    jr  $ra
# Recursive fibonacci example
# Fibonacci seq.: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...
.data
result: .word 0

.text
.globl main
main:
    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 $S5$ contains $0xAF00A5D$ and $S6$ contained $0x6$, what hexadecimal value would be in $S4$ after each of the following?
   a) slt $S4$, $S5$, 3
   b) sllv $S4$, $S5$, $S6$
   c) sra $S4$, $S5$, 3
   d) ror $S4$, $S5$, $S6$

2. If $S5$ contains $0xA5D$ and $S6$ contained $0x63C$, what hexadecimal value would be in $S4$ after each of the following?
   a) and $S4$, $S5$, $S6$
   b) ori $S4$, $S5$, $0xBF$
   c) xor $S4$, $S5$, $S6$
   d) nor $S4$, $S5$, $S6$
   e) not $S4$, $S5$

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:

$$
\begin{align*}
\text{bit position:} & \quad 25 \quad 24 \quad 23 \\
\{ \text{‘A’, ‘B’, ‘D’, ‘Y’} \} \text{ is} & \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 1 \quad 0 \\
\text{in } S5 & \quad 0 \quad 1 \quad 0 \quad 1 \quad 1
\end{align*}
$$

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 $S3$?
   b) If a bit-string set of letters is in register $S5$, then what instruction(s) can be used to check if the character ‘C’ (using the mask in $S3$) is in the set contained in $S5$?
   c) If a bit-string set of letters is in register $S5$ and another bit-string set of letters is in $S6$, then what instruction(s) can be used to calculate in $S7$ the union of sets in $S5$ and $S6$? (i.e., all elements in either set)
   d) If a bit-string set of letters is in register $S5$ and another bit-string set of letters is in $S6$, then what instruction(s) can be used to calculate in $S7$ the intersection of sets in $S5$ and $S6$? (i.e., all elements in both sets)
   e) If a bit-string set of letters is in register $S5$ and another bit-string set of letters is in $S6$, then what instruction(s) can be used to calculate in $S7$ the set difference: set in $S5$ - set in $S6$? (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.

\[
\begin{array}{c|c|c|c}
\text{unused} & 'Z' & 'Y' & 'X' \\
\hline
\{ 'A', 'B', 'D', 'Y' \} & 0 & 0 & 0 \\
& 0 & 1 & 0 \\
\hline
\text{bit position:} & 25 & 24 & 23 \\
\end{array}
\]

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 .ASCII 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 .ASCII 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

1. $4, 0xB9 = #40000000 01011101
2. $5, 0x35 = #50000000 00011010

and $6, $4, $5 = 60000000 00011010

or $7, $4, $5

logical-fill with 0's

shift left

$8, $4, 2

rol $8, $4, 2

ror $9, $4, 2

0x 4000 0002 E16
### 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><code>$4$ &amp; $5$ (bit-wise AND) $6$</code></td>
</tr>
<tr>
<td><code>andi $4, $5, 0x5f</code></td>
<td><code>$4$ &amp; $5$ (bit-wise AND) $5f_{16}$</code></td>
</tr>
<tr>
<td><code>or $4, $5, $6</code></td>
<td><code>$4$ &amp; $5$ (bit-wise OR) $6$</code></td>
</tr>
<tr>
<td><code>ori $4, $5, 0x5f</code></td>
<td><code>$4$ &amp; $5$ (bit-wise OR) $5f_{16}$</code></td>
</tr>
<tr>
<td><code>xor $4, $5, $6</code></td>
<td><code>$4$ &amp; $5$ (bit-wise Exclusive-OR) $6$</code></td>
</tr>
<tr>
<td><code>xorl $4, $5, 0x5f</code></td>
<td><code>$4$ &amp; $5$ (bit-wise Exclusive-OR) $5f_{16}$</code></td>
</tr>
<tr>
<td><code>nor $4, $5, $6</code></td>
<td><code>$4$ &amp; $5$ (bit-wise NOR) $6$</code></td>
</tr>
<tr>
<td><code>not $4, $5</code></td>
<td><code>$4$ NOT $5$ #inverts all the bits</code></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><code>$4$-shift left $5$ by 3 positions. Shift in zeros (only least significant 5-bits of immediate value are used to shift)</code></td>
</tr>
<tr>
<td><code>sllv $4, $5, $6</code></td>
<td>Similar to <code>sll</code>, but least significant 5-bits of $6$ determine the amount to shift.</td>
</tr>
<tr>
<td><code>srl $4, $5, 3</code></td>
<td><code>$4$-shift right $5$ by 3 positions. Shift in zeros</code></td>
</tr>
<tr>
<td><code>srvl $4, $5, $6</code></td>
<td>Similar to <code>srl</code>, but least significant 5-bits of $6$ determine the amount to shift.</td>
</tr>
<tr>
<td><code>sra $4, $5, 3</code></td>
<td><code>$4$-shift right $5$ by 3 positions. Sign-extend (shift in sign bit)</code></td>
</tr>
<tr>
<td><code>srav $4, $5, $6</code></td>
<td>Similar to <code>sra</code>, but least significant 5-bits of $6$ determine the amount to shift.</td>
</tr>
<tr>
<td><code>rol $4, $5, 3</code></td>
<td><code>$4$-rotate left $5$ by 3 positions</code></td>
</tr>
<tr>
<td><code>rol $4, $5, $6</code></td>
<td>Similar to <code>rol</code>, but least significant 5-bits of $6$ determine the amount to rotate.</td>
</tr>
<tr>
<td><code>ror $4, $5, 3</code></td>
<td><code>$4$-rotate right $5$ by 3 positions</code></td>
</tr>
<tr>
<td><code>ror $4, $5, $6</code></td>
<td>Similar to <code>ror</code>, 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  # 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:

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