Supplement for MIPS (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 noticable, 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 ACcummulator, 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:

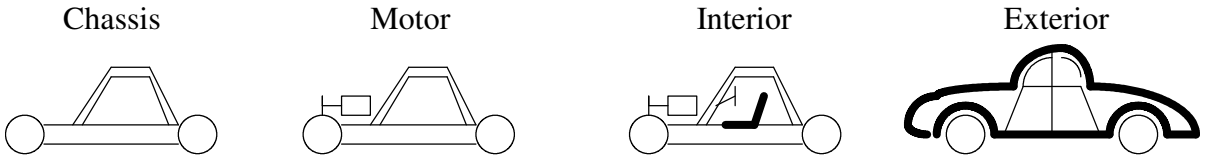
- reduce compiler complexity and aid assembly language programming. Compilers were not too good during the 50's to 70's, (e.g., they made poor use of general purpose registers so code was inefficient) so some programs were written in assembly language.
- reduce the program size. More powerful/complex 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.

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:

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

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 that 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

opcode	dest reg	operand 1 reg	operand 2 reg	unused

Unconditional Branch/"jump": j someLabel

opcode	large offset from PC or absolute address

Arithmetic with immediate: addi R1, R2, 8

opcode	operand 1 reg	operand 2 reg	immediate value

Conditional Branch: beq R1, R2, end_if

opcode	operand 1 reg	operand 2 reg	PC-relative offset to label

Load/Store: lw R1, 16(R2)

opcode	operand reg	base reg	offset from base reg

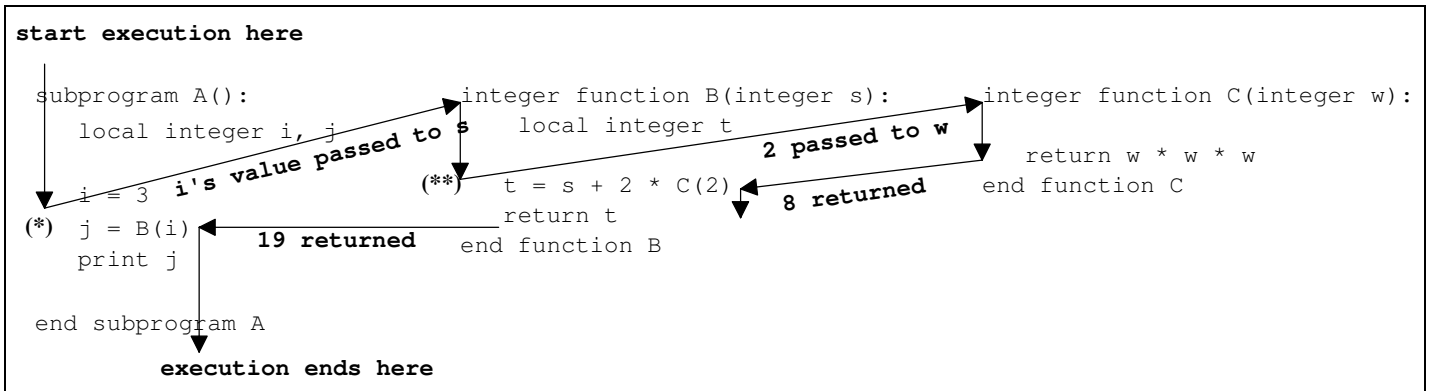
- hardwired control unit. The simple instructions can be performed using hardwired control unit that allows for a fast clock cycle

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

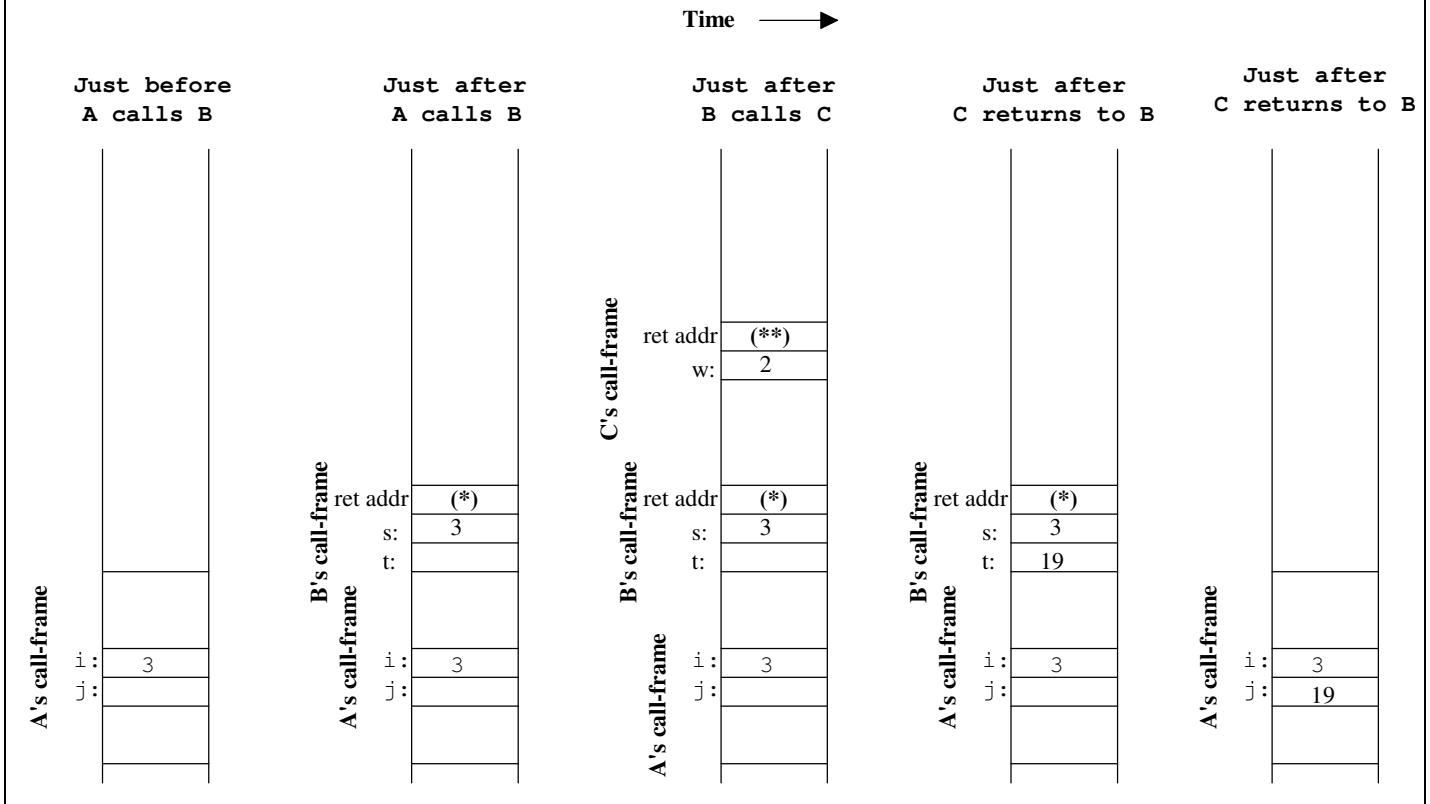
the subprogram, *formal parameters* are used to describe the task. When a subprogram is called, the actual parameter values are passed to the formal parameters. This is called parameter passing.
To help manage memory for subprograms, a run-time stack is used to provide memory space for a subprogram when 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:



Supplement for MIPS (Section 4.14 of the textbook)

Consider the more realistic program that calculates the values:

1^1 1^2 1^3 1^4
2^1 2^2 2^3 2^4
3^1 3^2 3^3 3^4

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
```

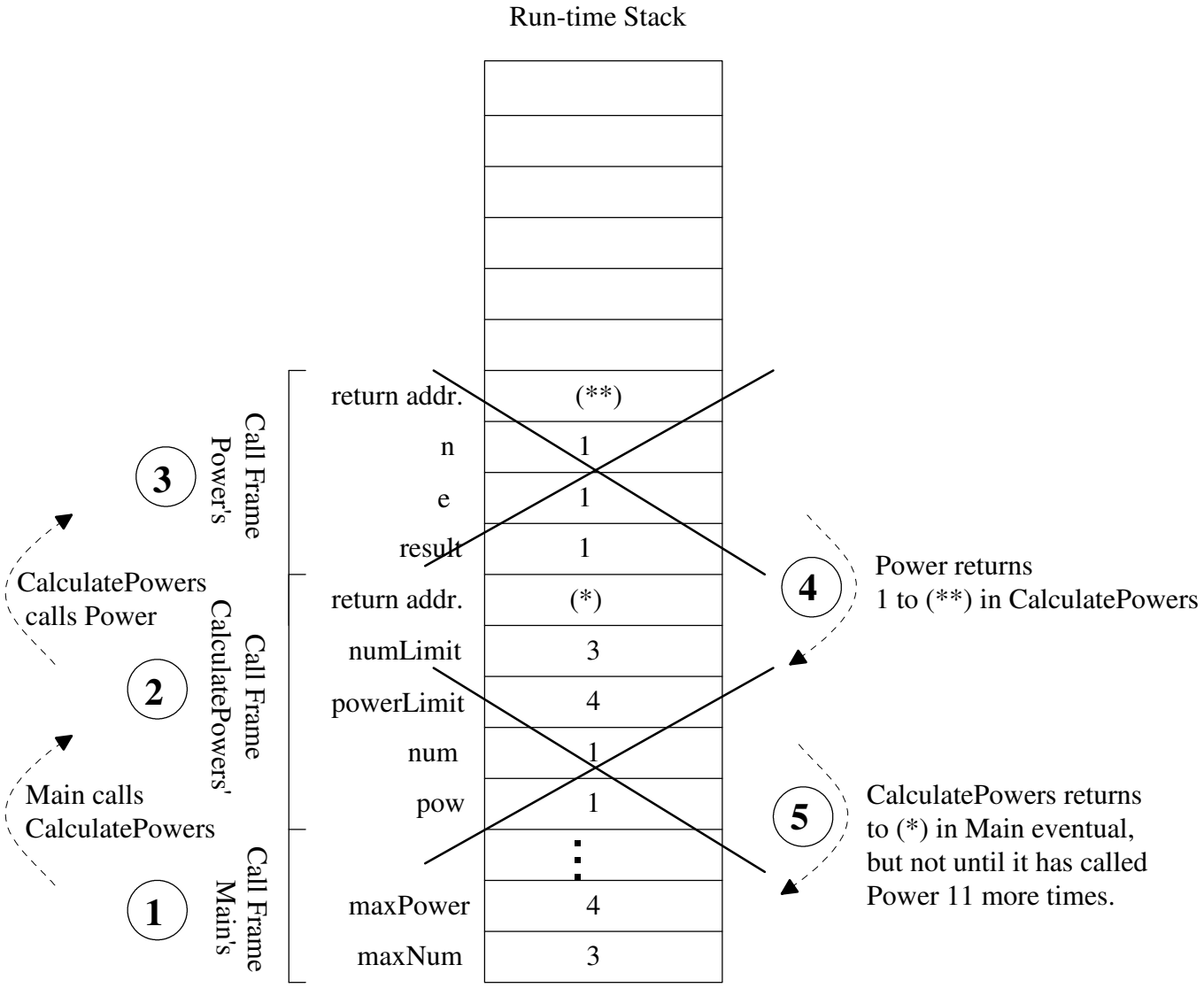
end CalculatePowers

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

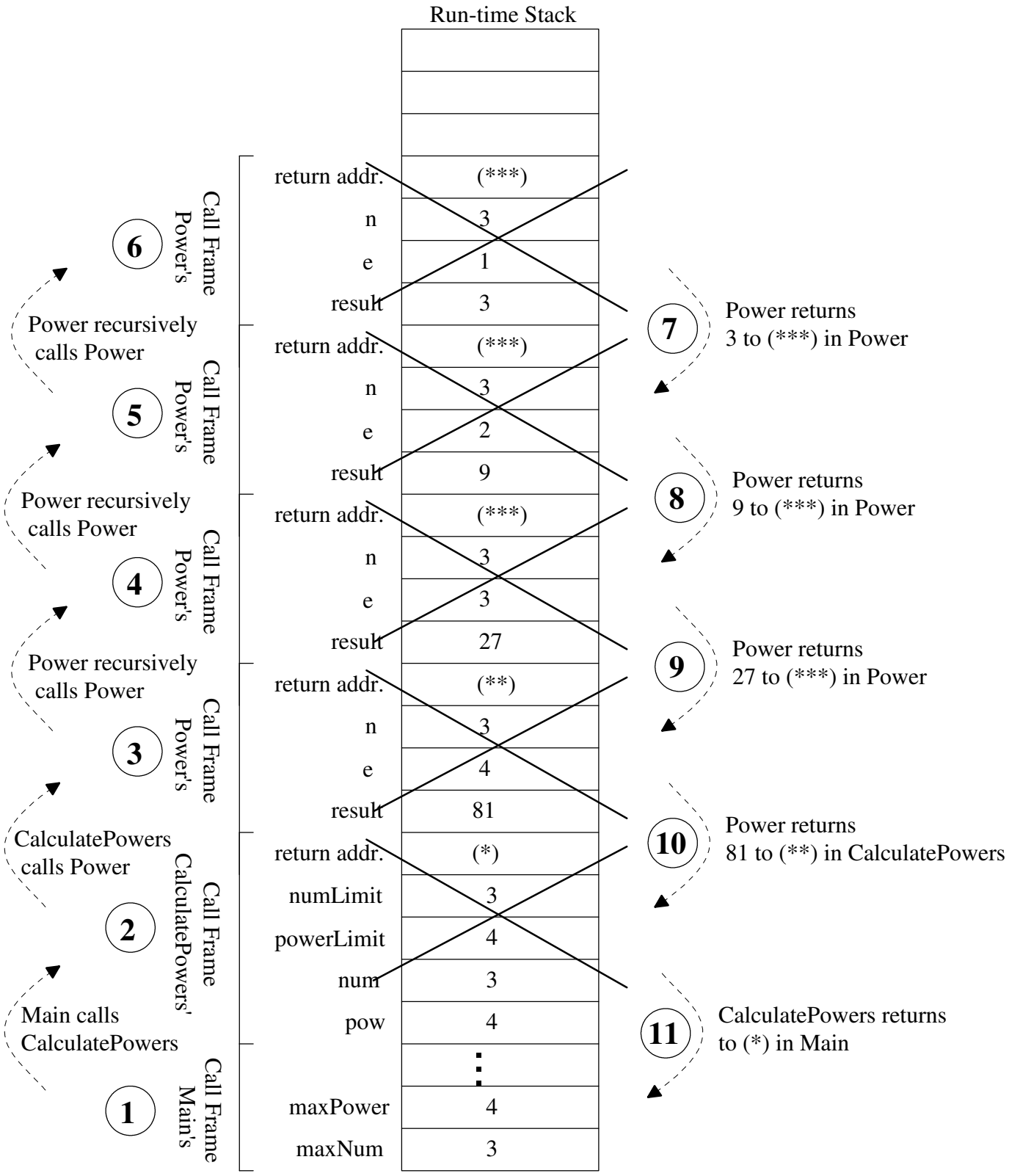
```
   integer result
   if e = 0 then
      result = 1
   else if e = 1 then
      result = n
   else
     result = Power(n, e - 1) * n        (***)
   end if
   return result
```
end Power

Supplement for MIPS (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.


Run-time Stack
return addr.
(***)
n
3
e
1
result
3
return addr.
(***)
n
3
e
2
result
9
return addr.
(***)
n
3
e
3
result
27
return addr.
(**)
n
3
e
4
result
81
return addr.
(*)
numLimit
3
powerLimit
4
num
3
pow
4
maxPower
4
maxNum
3
Call Frame Power's
Call Frame Power's
Call Frame Power's
Call Frame Power's
Call Frame CalculatePowers'
Call Frame Main's
6
5
4
3
2
1
Power recursively calls Power
Power recursively calls Power
Power recursively calls Power
CalculatePowers calls Power
Main calls CalculatePowers
7
Power returns 3 to (***) in Power
8
Power returns 9 to (***) in Power
9
Power returns 27 to (***) in Power
10
Power returns 81 to (**) in CalculatePowers
11
CalculatePowers returns to (*) in Main

Supplement for MIPS (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 there program
- to decrease the amount of memory used to store the program
- to gain access to low-level features of the machine that are difficulty 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.

Supplement for MIPS (Section 4.14 of the textbook)

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

Lets look at a simple, complete MIPS program to calculate: result = (x + y) * (10 - z), where x is 1, y is 2, and z is 3.

```
# Simple program to calculate:  result = (x + y) * (10 - z).
     .data                  # Data segment used to global variables
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                  # Text segment used to store the program
     .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, ":".

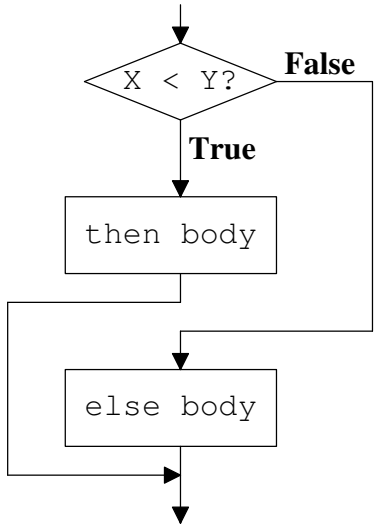
Supplement for MIPS (Section 4.14 of the textbook)

Lets 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

```
if X < Y then
     ...
else
     ...
end if
```

Flow chart



Assembly Language

```
     lw $8, X
     lw $9, Y
     bge $8,$9,ELSE
THEN:

     J END_IF
ELSE:

END_IF:
```

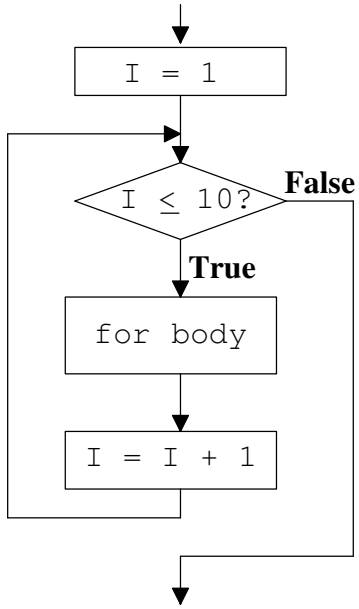
Since we want to conditionally jump over the THEN part when X < Y is False, the branch condition we check is the opposite of less-than, i.e., **g**reater-than-or-**e**qual (bge). If the THEN part is executed, then we jump to the END_IF.

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

HLL statement

```
for I = 1 to 10 do
     ...
end for
```

Flow chart



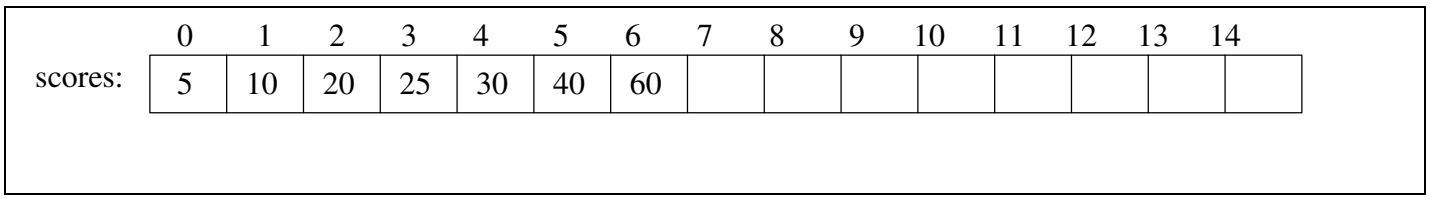
Assembly Language

```
FOR_INIT: li $5,1
FOR_COND: bgt $5,10,END_FOR
FOR_BODY:

          addi $5,$5,1
          j FOR_COND
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 ≤ 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.

Supplement for MIPS (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.



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:

```
            .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
```

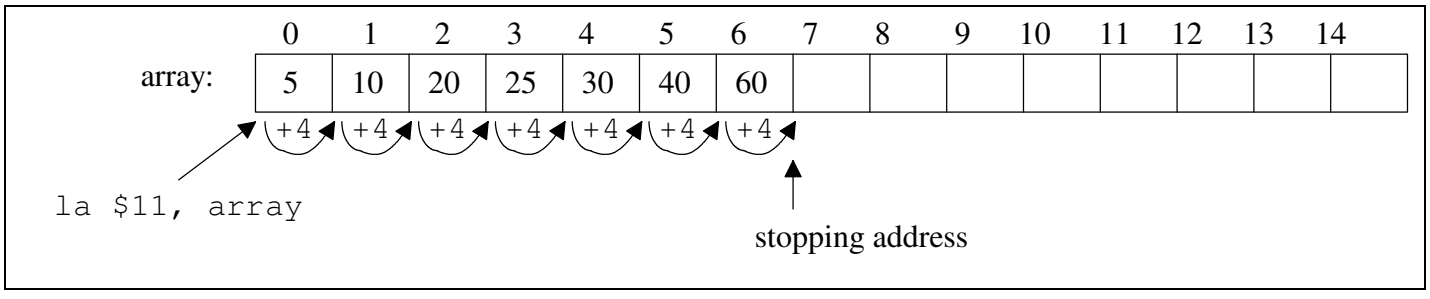
The above load instruction "`lw $8, 0($7)`" loads register $8 with the address specified by `0($7)` where 0 is a displacement added to the address in $7. Since we calculated the exact address of array[i] in $7, adding 0 is what we want to do. A displacement is useful in an array if you are accessing 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)`.

Supplement for MIPS (Section 4.14 of the textbook)

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

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



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.

```
      .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
```

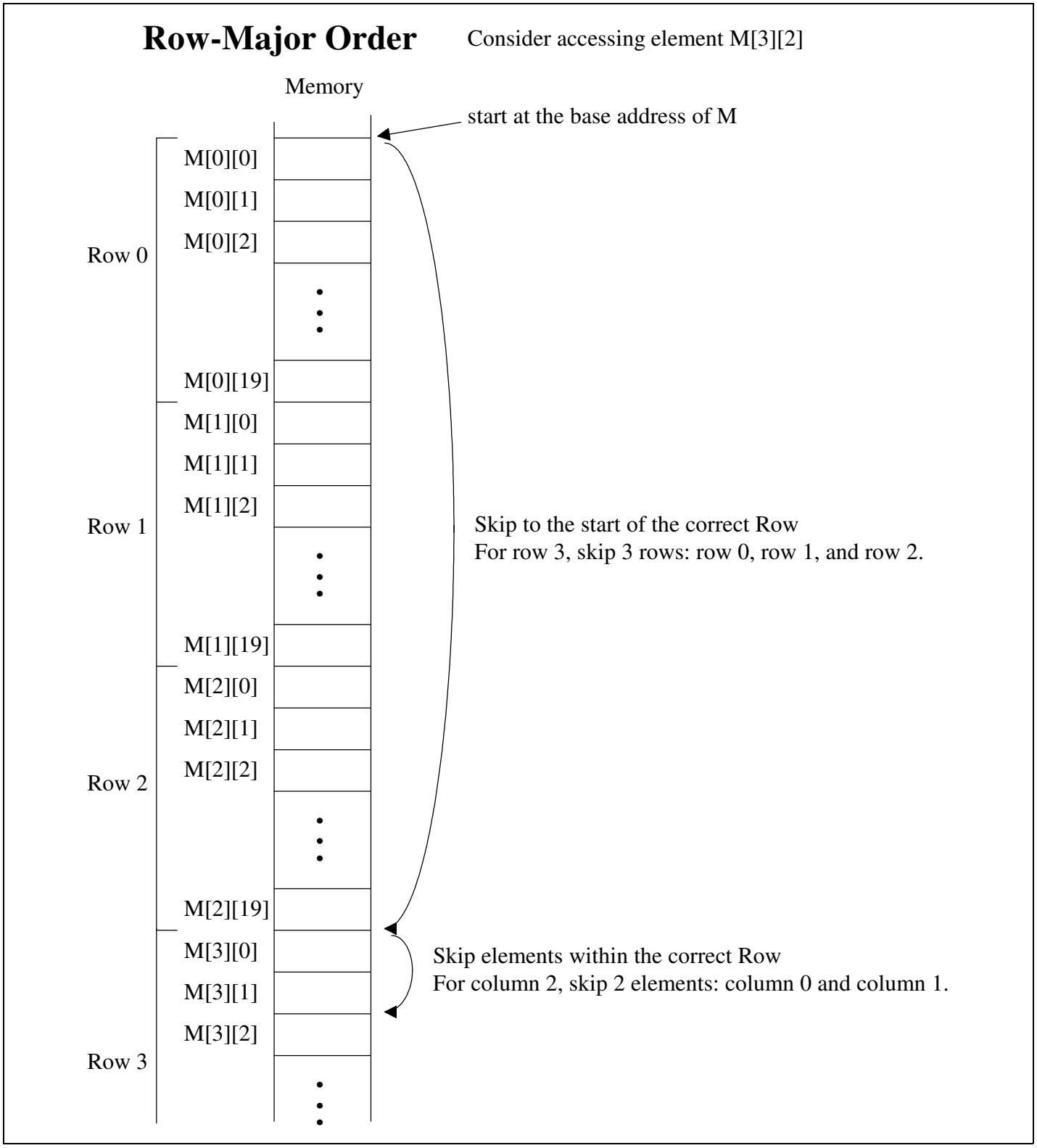
Multi-dimensional Arrays:

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.

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

Lets 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.



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

Supplement for MIPS (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 $5  and c's is in $6
      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.