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, result

for num := 1 to numLimit do
    for pow := 1 to powerLimit do
        Power(num, pow, result) (**)
        print num “ raised to “ pow “ power is “ result
    end for pow
end for num
end CalculatePowers

Power(In: integer n, integer e, Out: result)
if e = 0 then
    result = 1
else if e = 1 then
    result = n
else
    Power(n, e - 1, result) (***)
    result = result * n
end if
end Power

1) Trace the execution of the recursive function Power by showing the run-time stack.

2) What is the most number of call frames on the stack for the whole program?
3) Trace the hypothetical assembly language program and indicate the resulting value of the registers Reg1, Reg2, Reg3, and Reg4.

```
.data
X: .WORD 2 ; variable X initialized at assembly time to 2
Y: .WORD 3 ; variable Y initialized at assembly time to 3
Z: .WORD 0 ; variable Z initialized at assembly time to 0

.program
Begin:
  LOAD Reg1, X ; loads X’s value into register Reg1
  LOAD Reg2, Y
  ZERO Reg3 ; sets Reg3’s value to 0
  MOVE Reg4, Reg2 ; Reg4 := Reg2
Loop:
  ADD Reg3, Reg3, Reg1 ; Reg3 := Reg3 + Reg1
  SUB_IMMEDIATE Reg4, Reg4, #1 ; Reg4 := Reg4 - 1
  BRANCH_GREATER_THAN_ZERO Reg4, Loop ; if Reg4 > 0 then goto Loop label
  STORE Reg3, Z ; store Reg3’s value into variable Z
End:
```

<table>
<thead>
<tr>
<th>Resulting register values</th>
<th>Reg1</th>
<th>Reg2</th>
<th>Reg3</th>
<th>Reg4</th>
</tr>
</thead>
</table>

a) What is the resulting value in Z?

b) What calculation does this code perform?

4) During the execution of the above assembly language code: (Assuming no cache)
a) How many memory reads were performed? (state any assumptions)
   # data reads =

   # instruction reads (assume one read per instruction-fetch) =

b) How many memory writes were performed? (state any assumptions)

5) List (in decreasing order of importance) why somebody would write assembly language code.
   (top reason) a)
   b)
   c)
6) “Moore’s law” (Gordon Moore - cofounder of Intel) - predicts that the number of transistors that could be put on a single chip would double every year (later changed to 18 months). What kind of curve (# transistors vs. time) does Moore’s law predict? (linear, quadratic, exponential, etc.)

7) What impact does Moore’s law have on Computer Architectures?

8) In the "Machine-Instruction Weighted" columns in Table 13.2 (Stallings) why does a subroutine call require many assembly-language instructions?

9) In the "Memory-Reference Weighted" columns in Table 13.2 (Stallings) why does the subroutine call take even longer when memory accesses are considered?
Instruction/Machine Cycle of stored-program computer - repeat all day

1. Fetch Instruction - read instruction pointed at by the program counter (PC) from memory into Instr. Reg. (IR)

2. Decode Instruction - figure out what kind of instruction was read

3. Fetch Operands - get operand values from the memory or registers

4. Execute Instruction - do some operation with the operands to get some result

5. Write Result - put the result into a register or in a memory location

(Note: Sometime during the above steps, the PC is updated to point to the next instruction.)
<table>
<thead>
<tr>
<th>Type of Instruction</th>
<th>MIPS Assembly Language</th>
<th>Register Transfer Language Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Access</td>
<td><strong>lw</strong> $4$, Mem</td>
<td>$4\leftarrow \text{Mem} \quad \text{(Load and Store)}$</td>
</tr>
<tr>
<td></td>
<td><strong>sw</strong> $4$, Mem</td>
<td>Mem$\leftarrow 4$</td>
</tr>
<tr>
<td></td>
<td><strong>lw</strong> $4$, $16(3)$</td>
<td>$4\leftarrow [\text{Mem at address in } 3 + 16] \quad \text{(Load and Store)}$</td>
</tr>
<tr>
<td></td>
<td><strong>sw</strong> $4$, Mem</td>
<td>$4\leftarrow \text{Mem at address in } 3 + 16 \leftarrow 4 \quad \text{(Load and Store)}$</td>
</tr>
<tr>
<td>Move</td>
<td>move $4$, $2$</td>
<td>$4\leftarrow 2 \quad \text{(Move)}$</td>
</tr>
<tr>
<td></td>
<td>li $4$, 100</td>
<td>$4\leftarrow 100 \quad \text{(Load IMMEDIATE)}$</td>
</tr>
<tr>
<td>Load Address</td>
<td>la $5$, mem</td>
<td>$4\leftarrow \text{Load address of mem}$</td>
</tr>
<tr>
<td>Arithmetic Instruction (reg. operands only)</td>
<td><strong>add</strong> $4$, $2$, $3$</td>
<td>$4\leftarrow 2 + 3$</td>
</tr>
<tr>
<td></td>
<td>mul $10$, $12$, $8$</td>
<td>$10\leftarrow 12 \times 8 \quad \text{(32-bit product)}$</td>
</tr>
<tr>
<td></td>
<td><strong>sub</strong> $4$, $2$, $3$</td>
<td>$4\leftarrow 2 - 3 \quad \text{(Subtract)}$</td>
</tr>
<tr>
<td>Arithmetic with Immediates (last operand must be an integer)</td>
<td><strong>addi</strong> $4$, $2$, 100</td>
<td>$4\leftarrow 2 + 100 \quad \text{(Add IMMEDIATE)}$</td>
</tr>
<tr>
<td></td>
<td>mul $4$, $2$, 100</td>
<td>$4\leftarrow 2 \times 100 \quad \text{(32-bit product)}$</td>
</tr>
<tr>
<td>Conditional Branch</td>
<td>bgt $4$, $2$, LABEL</td>
<td>Branch to LABEL if $4 &gt; 2 \quad \text{(Conditional Branch)}$</td>
</tr>
<tr>
<td></td>
<td>(bge, blt, ble, beq, bne)</td>
<td></td>
</tr>
<tr>
<td>Unconditional Branch</td>
<td>j LABEL</td>
<td>Always Branch to LABEL \quad \text{Always Branch to LABEL)}</td>
</tr>
</tbody>
</table>

Fibonacci Sequence: \[0 \, 1 \, 1 \, 2 \, 3 \, 5 \, 8 \, 13 \, 21 \]
Position in Sequence: \[0 \, 1 \, 2 \, 3 \, 4 \, 5 \, 6 \, 7 \, 8 \]

A high-level language program to calculate the \(n\)th fibonacci number would be:

```plaintext
temp2 = 0
temp3 = 1
for i = 2 to n do
    temp4 = temp2 + temp3
    temp2 = temp3
    temp3 = temp4
end for
result = temp4
```

A complete assembly language MIPS program to calculate the \(n\)th fibonacci number.

```plaintext
#data
n: .word 8         # variable in memory
result: .word 0    # variable in memory

.text
.globl main
main: li $2, 0
li $3, 1
lw $5, n # load "n" into $5
li $6,
for_loop: bgt $6, $5, end_for
    add $4, $2, $3
    move $2, $3
    move $3, $4
    addi $6, $6, 1
    j for_loop
end_for:
    sw $4, result
    li $v0, 10 # system code for exit
    syscall
```

5