1. [15 Points] **x86 Programming**

Write the 64-bit x86 code for the following function, following all standard x64 calling conventions:

```c
int my_func(unsigned char *input_array) {
    int output = 0;
    while (*input_array) {
        if (*input_array < 127)
            output += *input_array;
        ++input_array;
    }
    return output;
}
```

**Answer:**

```assembly
my_func:
    pushq %rbp
    movq %rsp, %rbp
    xor %eax, %eax

loop:
    cmpb (%rdi), $0
    je exit_loop
    cmpb (%rdi), $127
    jge increment
    xor %ecx, %ecx
    movb (%rdi), %ecx
    add %ecx, %eax

increment:
    inc %rdi
    jmp loop

exit_loop:
    leave
    ret
```
2. [15 Points] **MIPS Programming**

Write the MIPS code for the following function, following all standard MIPS32 calling conventions:

```c
int fib(int x) {
    if (x <= 1)
        return 1;
    return fib(x-1) + fib(x-2);
}
```

**Answer:**

```
fib:
    addiu $sp, $sp, -12
    sw $ra, 8($sp)
    addi $v0, $0, 1
    sub $a0, $a0, $v0
    blez $a0, base_case
    sw $a0, 4($sp)
    jal fib
    sw $v0, 0($sp)
    lw $a0, 4($sp)
    addi $a0, $a0, -1
    jal fib
    lw $t0, 0($sp)
    add $v0, $v0, $t0
base_case:
    lw $ra, 8($sp)
    addiu $sp, $sp, 12
    jr $ra
```
3. **[15 points] Datapath**

As you may recall, in x86 the load effective address instruction computes and stores an effective memory address in a register. For example, “`leal (%eax, %edx, 4), %eax`” assigns the value of `%eax + %edx * 4` to the `eax` register. Suppose we want to add a similar instruction to the 5-stage pipelined MIPS processor you implemented in lab. The syntax of this new I-format instruction is `lea rs, rt, scale` and it stores the value of `R[rt] * scale + R[rs]` into `R[rt]`.

(a) The diagram below is a simple version of the 5-stage pipelined MIPS processor from class. The controller has a new output wire named `LEA`, which is high when the controller decodes an `lea` instruction and low otherwise. Add the necessary logic to support this new instruction to the datapath below.

**Answer:**

(b) Complete the following table of control signals for the newly-supported `lea` instruction, specifying the value of each signal as 0, 1, or X (don’t care). Add columns for any control signals your added logic requires. Writing a 0 or 1 when an X is more accurate is not correct.

<table>
<thead>
<tr>
<th>Opcode</th>
<th>LEA</th>
<th>RegWrite</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>ALUOp</th>
<th>MemWrite</th>
<th>MemRead</th>
<th>MemToReg</th>
<th>PCSrc</th>
</tr>
</thead>
<tbody>
<tr>
<td>lea</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Answer:**

<table>
<thead>
<tr>
<th>Opcode</th>
<th>LEA</th>
<th>RegWrite</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>ALUOp</th>
<th>MemWrite</th>
<th>MemRead</th>
<th>MemToReg</th>
<th>PCSrc</th>
</tr>
</thead>
<tbody>
<tr>
<td>lea</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>add</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
4. [15 points] **Pipelining**

Imagine a pipelined processor with the following pipe stages:

Fetch1 → Fetch2 → Decode/Reg → Execute → Memory1 → Memory2 → Writeback

That is, accessing memory (for both instructions and data) requires two pipeline stages. Because of the increased delay in fetching instructions, this machine has 2 branch delay slots. Further, there is no partial result forwarding (e.g., there exist no forwarding paths from Mem1 → EX).

Fill in the following pipeline stage diagram for this processor when it executes the following code:

```plaintext
LOOP:
  LW $4, 0($5)
  BEQ $4, $4, LOOP
  ADDI $5, $5, 8
  SUBI $5, $5, -4
```

**Answer:**

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Fetch1</th>
<th>Fetch2</th>
<th>Decode/Reg</th>
<th>Ex</th>
<th>Mem1</th>
<th>Mem2</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LW</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>BEQ</td>
<td>LW</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>ADDI</td>
<td>BEQ</td>
<td>LW</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>SUBI</td>
<td>ADDI</td>
<td>BEQ</td>
<td>LW</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>SUBI</td>
<td>ADDI</td>
<td>BEQ</td>
<td>NOP</td>
<td>LW</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>SUBI</td>
<td>ADDI</td>
<td>BEQ</td>
<td>NOP</td>
<td>NOP</td>
<td>LW</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>SUBI</td>
<td>ADDI</td>
<td>BEQ</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
<td>LW</td>
</tr>
<tr>
<td>7</td>
<td>LW</td>
<td>SUBI</td>
<td>ADDI</td>
<td>BEQ</td>
<td>NOP</td>
<td>NOP</td>
<td>NOP</td>
</tr>
<tr>
<td>8</td>
<td>BEQ</td>
<td>LW</td>
<td>SUBI</td>
<td>ADDI</td>
<td>BEQ</td>
<td>NOP</td>
<td>NOP</td>
</tr>
<tr>
<td>9</td>
<td>ADDI</td>
<td>BEQ</td>
<td>LW</td>
<td>SUBI</td>
<td>ADDI</td>
<td>BEQ</td>
<td>NOP</td>
</tr>
<tr>
<td>10</td>
<td>SUBI</td>
<td>ADDI</td>
<td>BEQ</td>
<td>LW</td>
<td>SUBI</td>
<td>ADDI</td>
<td>BEQ</td>
</tr>
<tr>
<td>11</td>
<td>SUBI</td>
<td>ADDI</td>
<td>BEQ</td>
<td>NOP</td>
<td>LW</td>
<td>SUBI</td>
<td>ADDI</td>
</tr>
<tr>
<td>12</td>
<td>SUBI</td>
<td>ADDI</td>
<td>BEQ</td>
<td>NOP</td>
<td>NOP</td>
<td>LW</td>
<td>SUBI</td>
</tr>
</tbody>
</table>
5. [15 Points] Caching

Suppose your processor has a data cache of the following geometry:

- Total data size of 128 B
- Cache block size of 16 B
- 2-way set associativity with LRU replacement
- Writeback coherence policy
- Allocate-on-write for store misses

Hit returns in 1 cycle Miss penalty 4 cycles

Assuming a miss penalty of 4 cycles and that a cache hit returns in 1 cycle, what would be the hit rate, miss rate, number of writebacks, and average memory access time (in cycles) for the following address stream?

L 0x00000001
S 0x00000002
L 0x00000010
L 0x00000011
L 0x00000001
L 0x00000200
L 0x00000300
L 0x00000400
S 0x00000201
L 0x00000401
L 0x00000301

Answer:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Hit Rate</td>
<td>4/11</td>
</tr>
<tr>
<td>Miss Rate</td>
<td>7/11</td>
</tr>
<tr>
<td># Writebacks</td>
<td>2</td>
</tr>
<tr>
<td>AMAT</td>
<td>32/11</td>
</tr>
</tbody>
</table>
Imagine you’re building an iPhone application for processing images. As part of your application you need to extend the standard math functions described in math.h to compute the average value of a matrix of normalized pixel values. As a smart programmer, you wish to avoid “reinventing the wheel” so you query Google for implementations. Your search returns two promising results, both conveniently implemented as functions in the C programming language.

Result A:

```c
double average_matrix_by_row(double* data[], int numRows, int numCols) {
    double sum = 0.0;
    int r, c;
    for(r = 0; r < numRows; r++) {
        for(c = 0; c < numCols; c++) {
            sum += data[r][c];
        }
    }
    return sum / ((double) numRows * numCols);
}
```

Result B:

```c
double average_matrix_by_col(double* data[], int numRows, int numCols) {
    double sum = 0.0;
    int c, r;
    for(c = 0; c < numCols; c++) {
        for(r = 0; r < numRows; r++) {
            sum += data[r][c];
        }
    }
    return sum / ((double) numRows * numCols);
}
```

Considering that the iPhone 4’s A4 processor has a 32 KB, 4-way set associative cache with 16-word (64 B) blocks, which implementation would you choose for your program? You may assume that all local variables (e.g., sum, r, c) are kept in registers in the compiled version of the code. Justify your choice with quantitative reasons.

**Answer:** Result A takes much better advantage of spatial locality than Result B. Result A leverages the row major representation of two-dimensional arrays and in doing so enjoys a low miss rate on the order of 1 miss per every 8 accesses for a total miss rate of \((r \times c) / 8\) whereas Result B misses on each access for matrices with 9 or more columns for a worst-case miss rate of 1 and a best-case miss rate no lower than 75%.
7. [15 Points] True / False

Circle True or False for each of the following questions.

(a) True / False: Paging is the only way to provide protection between processes on x86 processors
(b) True / False: In x64 all arguments to functions are passed via the stack
(c) True / False: MIPS is an accumulator based architecture
(d) True / False: Programs always run faster on systems that have caches
(e) True / False: Adding a pipeline stage to a processor can decrease performance
(f) True / False: Memory mapped I/O devices can use memory that is cached by the main processor
(g) True / False: When the x86 processor receives an external interrupt while in user mode, the processor vectors to the given user mode interrupt handler
(h) True / False: I enjoyed this class

Answer:

(a) (2 pts) False
(b) (2 pts) False
(c) (2 pts) False
(d) (2 pts) False
(e) (2 pts) True
(f) (2 pts) True
(g) (2 pts) False
(h) (1 pt) True or False

8. [1 Point] Extra Credit

Steven (the TA) shares his initials with the opcode for what MIPS32 instruction?

Answer: Shift Right Logical (srl)
This page intentionally left blank for extra answer space, scratch work, caricatures of the course staff, doodles of your winter break plans or anything else you desire which fits here.