Columbia University
CSEE 3827 Fundamentals of Computer Systems
Midterm #2
Prof. Martha Kim
December 10th, 2015

SOLUTION KEY

<table>
<thead>
<tr>
<th>First</th>
<th>Last (Family)</th>
<th>UNI</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- You are allowed 75 minutes.
- You may consult your own 8.5" × 11" double-sided sheet of notes, but nothing else (e.g., no textbooks, no other notes, calculators, etc.).
- Perform all work on the test itself. Use the backs of the pages if necessary. Do not use scratch paper.
- Explain your answers.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Value</th>
<th>Score</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15</td>
<td></td>
<td>MIPS ISA</td>
</tr>
<tr>
<td>2</td>
<td>15</td>
<td></td>
<td>Calling Conventions</td>
</tr>
<tr>
<td>3</td>
<td>26</td>
<td></td>
<td>Single-Cycle Datapath</td>
</tr>
<tr>
<td>4</td>
<td>30</td>
<td></td>
<td>Pipelined Execution</td>
</tr>
<tr>
<td>5</td>
<td>14</td>
<td></td>
<td>Caches</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>100</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
1. (15 pts.) MIPS ISA. Here is a buggy implementation of a substring search function. It takes two arguments:

- $a0: pointer to string to be searched (aka, text)
- $a1: pointer to string to search for (aka, substring)

It searches for instances of the substring in the text. If a match is found, it returns:

- $v0: pointer to the start of the substring match in the text
- $v1: length of the string match

If the substring is not found, it returns:

- $v0: 0
- $v1: 0

This implementation has two nested loops. The outer loop advances through the text string, and, at each position, the inner loop compares the characters in the substring.

It also has a couple of bugs. Find them, explain the problem, and correct them so that the function works correctly.

```assembly
find:
outertop:  move $t0, $a0
            move $t1, $a1

innertop:  lbu $t2, 0($t0)
            lbu $t3, 0($t1)
            beqz $t2, exitnomatch
            beqz $t3, exitmatch
            beq $t2, $t3, advanceinner

advanceouter: addi $t0, $t0, 1
              j outertop

advanceinner: addi $t0, $t0, 1
              addi $t1, $t1, 1
              j innertop

exitmatch:   move $v0, $a0
             subu $v1, $a1, $t1
             jr $ra

exitnomatch: li $v0, 0
             li $v1, 0
             jr $ra
```
2. (15 pts.) Calling Conventions. Insert the necessary instructions to make this implementation of fibonacci adhere to MIPS calling conventions. You may not remove or change any of the instructions provided. (FYI: \( fib(0) = 0; fib(1) = 1; fib(n) = fib(n-1) + fib(n-2) \))

\[
\text{fib:} \\
\text{li } $t1, 2 \\
\text{slt } $t1, $a0, $t1 \\
\text{beq } $t1, $zero, recurse \\
\text{move } $v0, $a0 \\
\text{jr } $ra \\
\]

\[
\text{recurse:} \\
\text{addi } $sp, $sp, -12 \\
\text{sw } $s0, 0($sp) \\
\text{sw } $s1, 4($sp) \\
\text{sw } $ra, 8($sp) \\
\text{move } $s1, $a0 \\
\text{addi } $a0, $a0, -1 \\
\text{jal } fib \\
\text{move } $s0, $v0 \\
\text{move } $a0, $s1 \\
\text{addi } $a0, $a0, -2 \\
\text{jal } fib \\
\text{add } $v0, $v0, $a0 \\
\text{lw } $s0, 0($sp) \\
\text{lw } $s1, 4($sp) \\
\text{lw } $ra, 8($sp) \\
\text{addi } $sp, $sp, -12 \\
\text{jr } $ra
\]
3. (26 pts.) Single-cycle Datapath.

(a) (13 pts.) Extend the processor below to support the `lbu` instruction (opcode=0x100100).

![Diagram of a Datapath model with extended instruction `lbu` and its corresponding opcode.]

<table>
<thead>
<tr>
<th>Inst.</th>
<th>OP</th>
<th>RegWrite</th>
<th>RegDest</th>
<th>ALUSrc</th>
<th>Branch</th>
<th>MemWrite</th>
<th>MemToReg</th>
<th>ALUOp</th>
<th>lbu Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>000000</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td>100011</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>sw</td>
<td>101011</td>
<td>0</td>
<td>-</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>beq</td>
<td>000100</td>
<td>0</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>0</td>
<td>01</td>
</tr>
</tbody>
</table>

`lbu 100100 1 0 1 0 0 0 0 0 1`
(b) (15 pts.) Starting again from the original single cycle design, extend it to support the lui instruction (opcode=001111).

```
<table>
<thead>
<tr>
<th>Inst.</th>
<th>OP</th>
<th>RegWrite</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>Branch</th>
<th>MemWrite</th>
<th>MemToReg</th>
<th>ALUOp</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>00000</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>lw</td>
<td>100011</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>101011</td>
<td>0</td>
<td>-</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td>000100</td>
<td>0</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>-</td>
<td>01</td>
</tr>
</tbody>
</table>
```

```
4. (30 pts.) Pipelined Execution. Examine this sum function – that sums the integers in a zero-terminated array – as it executes on a five stage pipeline. The pipeline has forwarding and stall logic and branches are resolved in the decode stage. While \texttt{j} is not supported by our pipeline, assume that its target is calculated in the fetch stage, with the jump target fetched immediately in the following cycle.

\begin{verbatim}
add $v0, $0, $0
top: lw $t0, 0($a0)
beq $t0, $0, done
add $v0, $v0, $t0
addi $a0, $a0, 4
j top
done:
\end{verbatim}

(a) (15 pts.) Assume that sum is called with $a0 pointing to an array with one non-zero integer followed by a zero. Trace sum’s execution on a the pipeline using the grid on the next page.

(b) (15 pts.) Give an expression for the CPI of sum when called on an array with one billion entries.

\begin{itemize}
  \item CPI of non-zero cell iter:
    \begin{align*}
      & \text{5 instrs, 2 bubbles} \\
      & \rightarrow \frac{7}{5}
    \end{align*}
  \\
  \item CPI of zero cell iter:
    \begin{align*}
      & \text{2 instrs, 2 bubbles} \\
      & \rightarrow \frac{4}{2}
    \end{align*}
  \\
  \item CPI of initial add:
    \begin{align*}
      & \text{1 instr, 0 bubbles} \\
      & \rightarrow 1
    \end{align*}
\end{itemize}

\[
\text{CPI} = \frac{10^9 \cdot 5 \cdot \frac{7}{5} + 2 \cdot \frac{4}{2} + 1 \cdot 1}{5 \cdot 10^9 + 2 + 1}
\]
5. (14 pts.) Caches. Consider a computer a cache containing 32, 64-byte blocks, backed by $2^{32}$ bytes of main memory.

(a) (7 pts.) Assume that the cache is **direct mapped** and indicate whether each load will hit (H) or miss (M) in the cache.

```
lw $t1, 0x00C4($0)    M
lw $t2, 0x08C4($0)    M
lw $t3, 0x08C8($0)    H
lw $t1, 0x00C4($0)    M
lw $t2, 0x08C4($0)    M
lw $t3, 0x08C8($0)    H
lw $t1, 0x00C4($0)    M
lw $t2, 0x08C4($0)    M
lw $t3, 0x08C8($0)    H
```

\[
\begin{align*}
0x000000C4 &= \ldots 0000 1100 0100 \\
0x000008C4 &= \ldots 1000 1100 0100 \\
0x000008C8 &= \ldots 1000 1100 1000 \\
\end{align*}
\]
(b) (7 pts.) Now assume that the cache is 4-way set associative, and indicate whether the same references hit or miss in the cache.

```
lw $t1, 0x00C4($0)  M
lw $t2, 0x08C4($0)  M
lw $t3, 0x08C8($0)  H
lw $t1, 0x00C4($0)  H
lw $t2, 0x08C4($0)  H
lw $t3, 0x08C8($0)  H
lw $t1, 0x00C4($0)  H
lw $t2, 0x08C4($0)  H
lw $t3, 0x08C8($0)  H
```
### Arithmetic Operation

<table>
<thead>
<tr>
<th>Operation</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td><code>+</code></td>
</tr>
<tr>
<td>Subtract</td>
<td><code>-</code></td>
</tr>
<tr>
<td>Multiply</td>
<td><code>*</code></td>
</tr>
<tr>
<td>Divide</td>
<td><code>/</code></td>
</tr>
</tbody>
</table>

### Logical Operation

<table>
<thead>
<tr>
<th>Operation</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td><code>&amp;</code></td>
</tr>
<tr>
<td>OR</td>
<td>`</td>
</tr>
<tr>
<td>NOT</td>
<td><code>!</code></td>
</tr>
</tbody>
</table>

### Bitwise Operation

<table>
<thead>
<tr>
<th>Operation</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bitwise AND</td>
<td><code>&amp;</code></td>
</tr>
<tr>
<td>Bitwise OR</td>
<td>`</td>
</tr>
<tr>
<td>Bitwise NOT</td>
<td><code>!</code></td>
</tr>
</tbody>
</table>

### Jump and Branch Operation

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLTZ</td>
<td><code>BLTZ</code></td>
</tr>
<tr>
<td>BEQ</td>
<td><code>BEQ</code></td>
</tr>
<tr>
<td>BNE</td>
<td><code>BNE</code></td>
</tr>
<tr>
<td>BLT</td>
<td><code>BLT</code></td>
</tr>
<tr>
<td>BLTU</td>
<td><code>BLTU</code></td>
</tr>
</tbody>
</table>

### Conditional Jump and Control Operation

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAL</td>
<td><code>JAL</code></td>
</tr>
<tr>
<td>JALR</td>
<td><code>JALR</code></td>
</tr>
<tr>
<td>JALF</td>
<td><code>JALF</code></td>
</tr>
</tbody>
</table>

### System Instruction

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDI</td>
<td><code>ADDI</code></td>
</tr>
<tr>
<td>SUBI</td>
<td><code>SUBI</code></td>
</tr>
<tr>
<td>SLLI</td>
<td><code>SLLI</code></td>
</tr>
<tr>
<td>SRLI</td>
<td><code>SRLI</code></td>
</tr>
<tr>
<td>SLTI</td>
<td><code>SLTI</code></td>
</tr>
<tr>
<td>SLTIU</td>
<td><code>SLTIU</code></td>
</tr>
</tbody>
</table>

### Quick Reference

- MIPS32 Instruction Set
- System Instructions
-

### Arithmetic Instructions

- Add: `add
- Subtract: `sub
- Multiply: `mul
- Divide: `div

### Logical Instructions

- AND: `and
- OR: `or
- NOT: `not

### Bitwise Instructions

- Bitwise AND: `&
- Bitwise OR: `|
- Bitwise NOT: `!

### Jump and Branch Instructions

- BLTZ: `bltz
- BEQ: `beq
- BNE: `bne
- BLT: `blt
- BLTU: `bti

### Conditional Jump and Control Instructions

- JAL: `jal
- JALR: `jalr
- JALF: `jalf

### System Instructions

- ADDI: `addi
- SUBI: `subi
- SLLI: `slli
- SRLI: `srli
- SLTI: `slt
- SLTIU: `sltiu

---

The above instructions are for the complete instruction set of the MIPS processor.

Please refer to the detailed MIPS instruction set documentation for full details.

---

- **Code**
  - Instruction Set Architecture
  - MIPS Register File
  - Control and Status Register
  - Instruction Execution
  - Immediate Addressing
  - Register-Based Addressing
  - Instruction Format
  - MIPS Assembler Syntax
  - MIPS Linkage Editor
  - Source Code Extensions
  - Other

- **Quick Reference**
  - MIPS32 Instruction Set
  - System Instructions