Columbia University  
CSEE 3827 Fundamentals of Computer Systems  
Final Exam  
Prof. Martha A. Kim  
December 17, 2013

Name: ___________________________  
First  
Last (Family)  
UNI (e.g., mak2191) 

- You are allowed 3 hours.
- You may consult your own 8.5” × 11” double-sided sheet of notes, but nothing else (e.g., no text, no other notes, no computers or calculators).
- 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 Assembly</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td></td>
<td>Single Cycle Datapath</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td></td>
<td>Processor Performance</td>
</tr>
<tr>
<td>4</td>
<td>15</td>
<td></td>
<td>Software Performance</td>
</tr>
<tr>
<td>5</td>
<td>15</td>
<td></td>
<td>Pipelined Execution</td>
</tr>
<tr>
<td>6</td>
<td>10</td>
<td></td>
<td>Cache Performance</td>
</tr>
<tr>
<td>7</td>
<td>15</td>
<td></td>
<td>Direct-Mapped Cache</td>
</tr>
<tr>
<td>Total</td>
<td>90</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
1. MIPS Assembly (15 pts.)

(a) The swap function swaps two words in memory. The two addresses are passed as arguments and the function has no return value. Modify the implementation below to adhere to calling conventions.

```
swap: lw $s0, 0($s1)
   lw $s2, 0($s3)
   sw $s0, 0($s3)
   sw $s2, 0($s1)
   jr $ra
```

```
swap: lw $t0, 0($a0)
   lw $t1, 0($a1)
   sw $t0, 0($a1)
   sw $t1, 0($a0)
   jr $ra
```

(b) Correct the functional bugs in my implementation of maximum below. It should find the maximum value in an array of positive integers. The arguments, in order, are a pointer to the array of integers and the number of integers in the array. The function should return the maximum value, or 0 in the case of an empty array.

```
maximum: li $t0, 0
top:    beqz $a1, done
   lw $t0, 0($a0)
   slt $t0, $v0, $t0
   beqz $t0, adv
   mv $v0, $t0
adv:    addi $a1, $a1, -1
       addi $a0, $a0, 1
       j top
done:   jr $ra
```

The array pointer should be incremented by 4: addi $a0, $a0, 4.
Also, slt writes to $t0, overwriting the integer that had been loaded from the array. It should write to a different $t register, which should be tested by the following beqz.
2. Single Cycle Datapath (10 pts.)

Explain why my implementation of move if zero (movz) below is incorrect. The instruction is a conditional move on zero, and should copy register \( rs \) to register \( rd \) if and only if register \( rt \) contains 0. My changes are highlighted with a ⋆.

\[
\text{movz } rd, rs, rt
\]

<table>
<thead>
<tr>
<th></th>
<th>00</th>
<th>00</th>
<th>00</th>
<th>00</th>
</tr>
</thead>
<tbody>
<tr>
<td>rs</td>
<td>31</td>
<td>26</td>
<td>25</td>
<td>21</td>
</tr>
<tr>
<td>rt</td>
<td>20</td>
<td>16</td>
<td>15</td>
<td>11</td>
</tr>
<tr>
<td>rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>movz</td>
<td>00</td>
<td>00</td>
<td>01</td>
<td>10</td>
</tr>
</tbody>
</table>

On a \text{movz}, \( rd \) will always be written, because the \text{RegWrite} input to the or gate is always true for an R-type. Moreover, for instructions that should not write to the register file, this processor will end up writing whenever \( rt = 0 \).
3. Processor Performance (10 pts.)

Consider a 10 billion instruction program, consisting of 20% loads, 20% stores, and 60% other instructions. Will this program run faster on processor A or processor B?

- **Processor A**: A 200 MHz single cycle processor.
- **Processor B**: A 1000 MHz multi-cycle processor on which
  - loads take 10 cycles;
  - stores take 8 cycles; and
  - all other instructions take 4 cycles.

\[
\text{Time}_A = 10B \times 1 \times 200MHz \\
\text{Time}_B = 10B \times CPI_B \times 1000MHz \\
CPI_B = 0.2 \times 10 + 0.2 \times 8 + 0.6 \times 4 = 2.0 + 1.6 + 2.4 = 5.0
\]

A's CPI is one fifth of B's, but so is its clock frequency, so the two terms even out, meaning \(\text{Time}_A = \text{Time}_B\).
4. Software Performance (15 pts.)

Running on our fully bypassed, five-stage pipelined processor (i.e., W-X, M-X, M-D forwarding), which implementation of array \texttt{sum} is faster and why?

This function takes two arguments

- \( \texttt{a0} \): pointer to an array of integers
- \( \texttt{a1} \): number of integers in array

And returns the sum

- \( \texttt{v0} \): sum of all of the integers in the array

Hint: reason about the runtime of \texttt{sum1} and \texttt{sum2} on arrays of length \( N \).

\texttt{sum1}: 
```
add \$v0, \$0, \$0  # init v0 = 0
top1: lw \$t0, 0(\$a0)
    addi \$a1, \$a1, -1
    addi \$a0, \$a0, 4
    add \$v0, \$v0, \$t0
    beq \$a1, \$0, done1 # if done, exit
    beq \$0, \$0, top1 # else, repeat
done1:
```

\texttt{sum2}: 
```
sum1: add \$v0, \$0, \$0  # init v0 = 0
top1: lw \$t0, 0(\$a0)
    addi \$a1, \$a1, -1
    addi \$a0, \$a0, 4
    add \$v0, \$v0, \$t0
    beq \$a1, \$0, done1 # if done, exit
    beq \$0, \$0, top1 # else, repeat
done1:
```

- \texttt{init}: 1 instr
- \texttt{loop (N − 1 iters)}: 6 instrs + 1 mispredict (beq to top1)
- \texttt{loop (N^{th} iter)}: 5 instrs + 1 mispredict (beq to done1)

\textit{cycles} = 1 + 7 \times (N - 1) + 6 = 7N

\texttt{sum2}: 
```
add \$t1, \$a0, \$a1
    add \$t1, \$t1, \$a1
    add \$t1, \$t1, \$a1  # init t1 = end of array
    add \$v0, \$0, \$0  # init v0 = 0
    top2: beq \$a0, \$t1, done  # if done, exit
        lw \$t0, 0(\$a0)  # else, load and add
        addi \$a0, \$a0, 4
        add \$v0, \$v0, \$t0
        beq \$0, \$0, top2  # repeat
done2:
```

- \texttt{init}: 5 instrs
- \texttt{loop (N iters)}: 5 instrs + 1 mispredict (beq to top2)
- \texttt{loop (N^{th} iter)}: 1 instrs + 1 mispredict (beq to done2)

\textit{cycles} = 5 + 6 \times N + 2 = 7 + 6N

So, for sufficiently large \( N \), \texttt{sum2} will be faster.
5. Pipelined Execution (15 pts.)

Examine the datapath below, that has been frozen place with five in flight instructions. Indicate the values on the five wires that are circled in the datapath and listed below.

(a) ForwardAE = from W stage, so 01

(b) ForwardBE = from M stage, so 10

(c) ForwardBD = from regfile, so 0

(d) StallD = 1, because $t0 in beq produced by instr in X

(e) StallF = 1, whenever D stalls, F does too
6. Cache Performance (10 pts.)

Which of the following three cache hierarchies can be expected to offer the best performance? For each level, the access time (in cycles) and miss rates are indicated.

<table>
<thead>
<tr>
<th>Level</th>
<th>Hierarchy A</th>
<th>Hierarchy B</th>
<th>Hierarchy C</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>1 cycle; miss rate 25%</td>
<td>2 cycles; miss rate 10%</td>
<td>1 cycle; miss rate 25%</td>
</tr>
<tr>
<td>L2</td>
<td>30 cycles; miss rate 10%</td>
<td>20 cycles; miss rate 20%</td>
<td>20 cycles; miss rate 20%</td>
</tr>
<tr>
<td>Memory</td>
<td>100 cycles; miss rate 0%</td>
<td>100 cycles; miss rate 0%</td>
<td>100 cycles; miss rate 0%</td>
</tr>
</tbody>
</table>

Hierarchy A: Expected AccessTime = \(1 + 0.25 \times (30 + 0.1 \times 100) = 1 + 0.25 \times 40 = 11 \text{ cycles}\)

Hierarchy B: Expected AccessTime = \(2 + 0.10 \times (20 + 0.2 \times 100) = 2 + 0.10 \times 40 = 6 \text{ cycles}\)

Hierarchy C: Expected AccessTime = \(1 + 0.25 \times (20 + 0.2 \times 100) = 2 + 0.25 \times 40 = 12 \text{ cycles}\)
7. Direct-Mapped Cache (15 pts.)

For the 16B direct-mapped cache depicted below, fill in the cache's state after each lb operation shown below. Assume the cache starts empty, and to save time you may leave don't care values blank.

At 16B total over 4 lines, there are 4B per line. Thus, 2 byte offset bits, 2 set index bits (4 sets), and the remaining 4 bits are tag.

The memory contents are:

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>H a v e</td>
</tr>
<tr>
<td>0x08</td>
<td>o o d v a c a</td>
</tr>
<tr>
<td>0x10</td>
<td>t i o n</td>
</tr>
</tbody>
</table>

**lb 0x01 (Miss. Write block to set 0.)**

<table>
<thead>
<tr>
<th>v</th>
<th>tag</th>
<th>byte 0</th>
<th>byte 1</th>
<th>byte 2</th>
<th>byte 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**lb 0x08 (Miss. Write block to set 2.)**

<table>
<thead>
<tr>
<th>v</th>
<th>tag</th>
<th>byte 0</th>
<th>byte 1</th>
<th>byte 2</th>
<th>byte 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**lb 0x0A (Hit. No change.)**

<table>
<thead>
<tr>
<th>v</th>
<th>tag</th>
<th>byte 0</th>
<th>byte 1</th>
<th>byte 2</th>
<th>byte 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**lb 0x13 (Miss. Tag mismatch. Overwrite block to set 0.)**

<table>
<thead>
<tr>
<th>v</th>
<th>tag</th>
<th>byte 0</th>
<th>byte 1</th>
<th>byte 2</th>
<th>byte 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Default C Calling Convention (O32)

### Stack Management
- The stack grows down.
- Subtract from $sp$ to allocate local storage space.
- Restore $sp$ by adding the same amount at function exit.
- The stack must be 8-byte aligned.
- Modify $sp$ only in multiples of eight.

### Function Parameters
- Every parameter smaller than 32 bits is promoted to 32 bits.
- First four parameters are passed in registers $v0$–$v3$.
- 64-bit parameters are passed in register pairs:
  - Little-endian mode: $a1$–$a0$ or $a3$–$a2$.
  - Big-endian mode: $a0$–$a1$ or $a2$–$a3$.
- Every subsequent parameter is passed through the stack.
- First 16 bytes on the stack are not used.
- Assuming $sp$ was not modified at function entry:
  - The 1st stack parameter is located at 16($sp$).
  - The 2nd stack parameter is located at 20($sp$), etc.
- 64-bit parameters are 8-byte aligned.

### Return Values
- 32-bit and smaller values are returned in register $v0$.
- 64-bit values are returned in registers $v0$ and $v1$:
  - Little-endian mode: $v1$–$v0$.
  - Big-endian mode: $v0$–$v1$.

### Assembler-Language Function Example

```c
#define __attribute__((packed)) unaligned;

typedef struct
{
    int u;
} __attribute__((packed)) unaligned;

int unaligned_load(void *ptr)
{
    unaligned *uptr = (unaligned *)ptr;
    return uptr->u;
}
```

### C/Assembly-Language Function Interface

```c
#include <stdio.h>

int asm_max(int a, int b);

int main()
{
    int x = asm_max(10, 100);
    int y = asm_max(200, 20);
    printf("%d %d\n", x, y);
}
```

### Invoking MULT and ADD Instructions From C

```c
int dp(int a[], int b[], int n)
{
    int i;
    long long acc = (long long) a[0] * b[0];
    for (i = 1; i < n; i++)
        acc += (long long) a[i] * b[i];
    return (acc >> 31);
}
```

### MIPS SDE-GCC Compiler Defines

```c
#define __mips MIPSEL
#define __mips_dsp __mips
#define __mips_isa_rev _MIPS_ARCH_
#define __mips_tune_cpu _MIPS_TUNE_CPU
#define _mips _MIPS_SDE_GCC_C
#define _mips_isa_rev _MIPS_SDE_GCC_C
#define _mips_dsp _MIPS_SDE_GCC_C
#define _MIPS_ARCH_CPU _MIPS_SDE_GCC_C
#define _MIPS_TUNE_CPU _MIPS_SDE_GCC_C
```

### Notes
- Many assembler pseudo-instructions and some rarely used machine instructions are omitted.
- The C calling convention is simplified. Additional rules apply when passing complex data structures as function parameters.
- The examples illustrate syntax used by GCC compilers.
- Most MIPS processors increment the cycle counter every other cycle. Please check your processor documentation.