Name:___________________________

CSE 30321 – Computer Architecture I – Fall 2011
Midterm Exam
October 14, 2011

Test Guidelines:

1. Place your name – or at least your initials! – on ***EACH*** page of the test in the space provided. **Be sure to do this on p. 1 and 2!**
2. Answer every question in the space provided. If separate sheets are needed, make sure to include your name and clearly identify the problem being solved.
3. Read each question carefully. Ask questions if anything needs to be clarified.
4. The exam is open book and open notes.
5. All other points of the ND Honor Code apply. By writing your name on the exam, you agree to abide by the ND Honor Code.
6. Upon completion, please turn in the test and any scratch paper that you used.

Suggestion:

- Whenever possible, show your work and your thought process. This will make it easier for us to give you partial credit.

Name:_______________________________

Score Sheet

Question	Possible Points	Your Points
1	20	
2	15	
3	10	
4	15	
5	20	
6	20	
Total	100	

Name:___________________________

Problem 1: (20 points)

Below, snippets of assembly code are shown for different, nested procedures. Based on the code, answer the questions below. **Your answers should reflect the MIPS procedure call conventions.**

main

```
add   $t1, $t2, $t3
lw    $s1, 0($t1)
lw    $s2, 4($t1)
add   $a0, $s1, $0
add   $a1, $s2, $0
jal   Function_1
…
sw    $s1, 0($v0)
sw    $s2, 4($v0)
```

Function_1

```
…
…
sub   $t1, $a0, $a1
lw    $t2, 0($t1)
…
add   $a0, $t1, $0
jal   Function_2
…
jr    < to main >
```

Function_2

```
…
…
…
…
…
…
…
…
jr    < to Function_1 >
```

Question A: (3 points)
Based on the code provided, what registers must be saved to the stack by Function_1 to preserve program correctness?

Answer: $ra (to main), $s1 $s2 (1 point each) (if answer includes references to other parts, OK)

Question B: (4 points)
Write the MIPS assembly language needed to implement your answer to Question A.

Answer:

```
addi  $sp, $sp, -12     (If subi is used, 0 points were subtracted.)
sw    $s1, 0($sp)
sw    $s2, 4($sp)
sw    $ra, 8($sp)       (any order of storing is fine…)
```

Question C: (3 points)
Presumably, how many arguments does Function_2 require?

Answer: 1. Only one argument is copied into an argument register – and it is $a0 (the first $a register)

Question D: (3 points)
If Function_1 uses the data in $a0 and $a1 (passed from main) *after* the call to Function_2, how are the values of $a0 and $a1 saved? What function preserves them? (No MIPS code is needed, just explain.)

Answer: Function_1 must save $a0 and $a1 to the **stack**. (Both must be mentioned, otherwise, -2)

Question E: (3 points)
If Function_2 calls the library routine X, what data do you think X would save to the stack at entry?

Answer: It would callee save the $ra, $sp, and $s registers. (Its OK to mention other pointer registers.)

Question F: (4 points)
Based on the code provided, what must Function_1 do before the jr instruction back to main?

Answer: (1) Write data to $v0 (1 bonus point) and **(2)** restore stack data ($s regs and $ra) (4 points)

Name:______________________________

Problem 2: (15 points)

Question: (15 points)
Translate the HLL code below to MIPS assembly. Every instruction should have a comment!

HLL Code

```
if (i==j) {
      a[i] = b[j];
}
else {
      a[i] = b[j+1];
}
```

Assume the following

The starting address of array `a[]` is in $20
The starting address of array `b[]` is in $21
Index `i` maps to $5
Index `j` maps to $6

If any temporary registers are used, please start with $10

Answer (long)

```
        bneq    $5, $6, else
if:     sll     $10, $5, 2
        sll     $11, $6, 2
        add     $10, $10, $20
        add     $11, $11, $21
        lw      $12, 0($11)
        sw      $12, 0($10)
        jump    end
else:   sll     $10, $5, 2
        sll     $11, $6, 2
        add     $10, $10, $20
        add     $11, $11, $21
        lw      $12, 4($11)
        sw      $12, 0($10)
end:
```

Total: 14 instructions

Answer (short)

```
        sll     $10, $5, 2          # multiply i*4
        sll     $11, $6, 2          # multiply j*4
        add     $10, $10, $20       # address of a[i]
        add     $11, $11, $21       # address of b[i]
        bneq    $5, $6, else        # does i == j?
if:     lw      $12, 0($11)         # load b[i]
        jump    end                 # skip else case
else:   lw      $12, 4($11)         # load b[j+1]
end:    sw      $12, 0($10)         # write a[i]
```

Total: 9 instructions

(If students use an addi instruction to handle j+1 – and then use lw, sw instructions with base 0, that's OK.)

For a **3 point** bonus, complete the question with 10 instructions or less.

Name:____________________________

Problem 3: (10 points)

Benchmark B is comprised of ALU, Load, Store, and Branch/Jump instructions. The number of instructions required to execute benchmark B are broken down by class-type in the table below. The number of clock cycles required to execute each instruction is also given in the table below.

Instruction Class	Clock Cycles per Instruction	Number of Instructions
ALU	6	1,000,000,000 (i.e. 1.0×10^9)
Load	8	200,000,000 (i.e. 2.0×10^8)
Store	7	180,000,000 (i.e. 1.8×10^8)
Branch / Jump	5	140,000,000 (i.e. 1.4×10^8)

To improve the performance of this benchmark, 2 changes will be made.

First, a new instruction will be added – *load++* – which will (a) copy a piece of data from memory into a register, and (b) update the register value that is currently used to calculate a memory address (such that a separate instruction is not required to calculate the address of the next data element).

The new *load++* instruction:

- Requires 8 CCs to execute – just like the original load.
- Will lead to the elimination of 70,000,000 (or 7×10^7) ALU instructions.

Second, a change will be made to the existing datapath to reduce the number of clock cycles associated with branch / jump instructions from 5 to 4.

Assuming a constant clock rate of 2 GHz, what speedup is obtained if both changes are implemented?

Answer:

Two CPU times must be calculated – a baseline CPU time and a new CPU time.

CPU_{base}:

$= [(1 \times 10^9)(6) + (2 \times 10^8)(8) + (1.8 \times 10^8)(7) + (1.4 \times 10^8)(5)]$ cycles $\times (0.5 \times 10^{-9}$ s / cycle)
$= [(6.0 \times 10^9) + (1.6 \times 10^9) + (1.26 \times 10^9) + (7.0 \times 10^8)]$ cycles $\times (0.5 \times 10^{-9}$ s / cycle)
$= (9.56 \times 10^9$ cycles) $\times (0.5 \times 10^{-9}$ s / cycle)
= 4.78 seconds

CPU_{new}:

- To calculate a new CPI, we first need to calculate a new number of ALU instructions.
  - $(1 \times 10^9) - (7 \times 10^7) = 9.3 \times 10^8$ ALU instructions.
- We can then perform a similar calculation to that given above
  - (Of course, the CPI for branch / jump instructions must be reduced by 1.)

$= [(9.3 \times 10^8)(6) + (2 \times 10^8)(8) + (1.8 \times 10^8)(7) + (1.4 \times 10^8)(4)]$ cycles $\times (0.5 \times 10^{-9}$ s / cycle)
$= [(5.58 \times 10^9) + (1.6 \times 10^9) + (1.26 \times 10^9) + (5.6 \times 10^8)]$ cycles $\times (0.5 \times 10^{-9}$ s / cycle)
$= (9.0 \times 10^9$ cycles) $\times (0.5 \times 10^{-9}$ s / cycle)
= 4.5 seconds

Thus, the new version is: 9.56 / 9.0 – or 6.2% faster.

Name:____________________________

Problem 4: (15 points)
Statistics regarding the number of instructions associated with a large MP3 encode and decode are shown in the table below. For this problem you should assume that these benchmarks are executed on a multi-cycle datapath. The number of clock cycles required to execute each instruction is also provided in the table below.

(Note that in this question, the number of clock cycles required for each instruction type is the same as for the MIPS multi-cycle datapath discussed in class.)

Statistics for large MP3 encode				Statistics for large MP3 decode		
Instruction type	Instruction count	Clock cycles per instruction		Instruction type	Instruction count	Clock cycles per instruction
Loads	4.20×10^8	5		Loads	5.10×10^8	5
Stores	2.14×10^8	4		Stores	1.77×10^8	4
Branch/Jump	4.70×10^7	3		Branch/Jump	8.70×10^7	3
ALU	4.69×10^8	4		ALU	1.01×10^9	4

Assume that we want the performance of the slower benchmark to match the original performance of the faster benchmark. For this problem, the only way that you can do this is to redesign your datapath hardware to improve CPI values (instruction counts and clock rate will remain the same).

What must the new, average CPI of the slower benchmark be in order to match the original performance of the faster benchmark?

Answer:

$$CPI_{lame} = \frac{(4.20\times10^8)(5) + (2.14\times10^8)(4) + (4.70\times10^7)(3) + (4.69\times10^8)(4)}{(4.20\times10^8) + (2.14\times10^8) + (4.70\times10^8) + (4.69\times10^8)} = \frac{4.973\times10^9}{1.15\times10^9} = 4.324$$

$$CPI_{mad} = \frac{(5.10\times10^8)(5) + (1.77\times10^8)(4) + (8.70\times10^7)(3) + (1.01\times10^9)(4)}{(5.10\times10^8) + (1.77\times10^8) + (8.70\times10^7) + (1.01\times10^9)} = \frac{7.543\times10^9}{1.78\times10^9} = 4.238$$

Given that the CPIs are similar, and the instruction count of *mad* is much larger than *lame*, *mad* will be the slower benchmark.

We now need to set the execution time of *lame* to that of *mad* – and see by what percentage the *mad* CPI must be reduced…

$$(1.15\times10^9)(4.324)(clock\ period) = (1.78\times10^9)(4.238)(N)(clock\ period)$$

Solving for N, we find that N = 0.659.

As such, the CPI of *mad* must be 66% of its original value – or **2.793** – to match the original performance of *lame*.

Name:____________________________

Problem 5: (20 points)

- Assume that we want to implement a simple, 16-instruction processor.
- The processor will have 64, user programmable registers.
- All 16 instructions – as well as the resulting register transfer language for different instruction types – are summarized in the table below:

Instruction Class	Instructions	Example Syntax	Comment
ALU (all data in registers)	ADD, SUB, AND, OR, NAND, NOR	ADD Rx, Ry, Rz Rx ← Ry + Rz	
ALU (1 operand immediate value)	ADDI, SUBI, SLL, SRL	ADDI Rx, Ry, N Rx ← Ry + Constant N	For this instruction, N is a 12-bit signed constant.
MULT/DIV (all data in registers)	MULT, DIV	MULT Rx, Ry, Rz Rx ← Ry * Rz	
Load	LOAD	LOAD Rx, Ry Rx ← Memory(Ry)	Note that the base address in register Ry is the final address sent to memory. No constant value is added.
Store	STORE	STORE Rx, Ry Memory(Ry) ← Rx	Note that the base address in register Ry is the final address sent to memory. No constant value is added.
Branch	BEQ, BNEQ	BEQ Rx, Ry, N if Rx == Ry, PC ← PC + N else, PC ← PC + 4	For this instruction, N is a 12-bit signed constant. (Thus, the PC value can increase OR decrease.)

Question A: (4 points)
How many bits must an instruction encoding be in order to support all of the instruction types described above?

Answer:

ALU:
- 4 bits of opcode
- 6 bits for destination register
- 6 bits for source register 1
- 6 bits tor source register 2
- 22 bits total

ALU (imm):
- 4 bits of opcode
- 6 bits for destination register
- 6 bits for source register 1
- 12 bits for constant
- 28 bits total

Ld: or St
- 4 bits of opcode
- 6 bits for destination / source register
- 6 bits for address register
- 16 bits total

Branch:
- 4 bits of opcode
- 6 bits for source register 1
- 6 bits for source register 2
- 12 bits for constant
- 28 bits total

28 bits are needed to support all instruction encodings.

Name:____________________________

Question B: (4 points)
Two sequences of instructions that would execute the core of the loop code shown below are provided.

You can / should assume the following:
- Register R0 contains the number 0.
- Register R1 maps to the variable `i` and is initially 0
- Register R2 maps to the variable `N`; N is greater than 0
- Register R10 contains the starting address of `a[]`
- Register R7 contains the constant 5
- To fetch the next data word in our array, we update the address by 4 – just like in MIPS

C-Code

```
for(i=0; i<N; i++) {
    a[i] = a[i] * 5;
}
```

Assembly Version 1

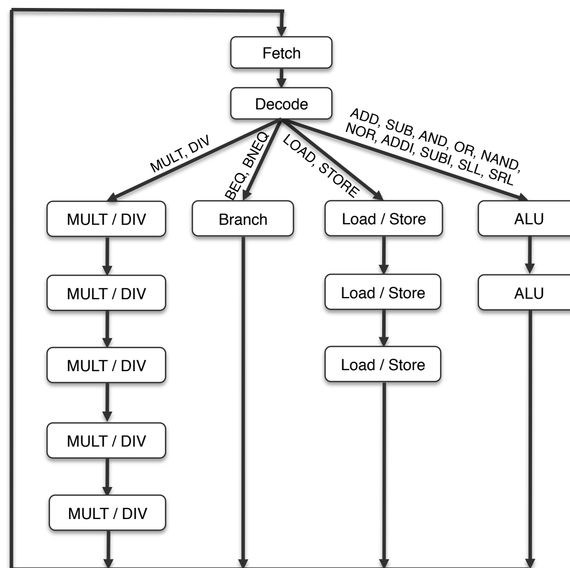
```
        ...
start:  BEQ    R1, R2, end
        SLL    R3, R1, 2
        ADD    R4, R3, R10
        LOAD   R5, R4
        ADD    R5, R5, R5
        ADD    R5, R5, R5
        ADD    R5, R5, R0
        Store  R5, R4
**      Addi   R1, R1, 1
        BEQ    R0, R0, start
end:
```

Assembly Version 2

```
        ...
start:  BEQ    R1, R2, end
        SLL    R3, R1, 4
        ADD    R4, R3, R10
        LOAD   R5, R4
        MULT   R5, R5, R7
        Store  R5, R4
**      Addi   R1, R1, 1
        BEQ    R0, R0, start
end:
```

Based on the finite state machine for this particular ISA / datapath shown above, how many clock cycles does **1 iteration** of each loop take?

State Machine



Answer:
Version 1 requires: 3 + 4 + 4 + 5 + 4 + 4 + 4 + 5 + 4 + 3 = 40 CCs
Version 2 requires: 3 + 4 + 4 + 5 + 7 + 5 + 4 + 3 = 35 CCs

Name:___________________________

<u>Question</u> <u>C</u>: (4 points)
In what clock cycles would the new value of R1 – written by the **'ed addi instructions – be available for reading with the updated value?

Answer:

- Clock cycle #38 for Sequence #1 and clock cycle #33 for Sequence #2

<u>Question</u> <u>D</u>: (4 points)
What if the C-Code in Question B instead read:

```
for(i=0; i<N; i++) {
    a[i] = a[i] * 8;
}
```

How might you change the assembly code shown above to make it more efficient? How much faster would each iteration of the loop be?

Answer:
The multiply instruction could be changed to a SLL instruction – i.e. SLL R5, R5, 3.
This would shave 4 more clock cycles off of the 34 clock cycle total – for a total of 31 CCs per iteration.
This would result in an ~10% speedup.

<u>Question</u> <u>E</u>: (4 points)
Write a sequence of assembly instructions with this ISA that will swap the data in address 1000 with the data in address 2000. Each line MUST have a comment associated with it.

Answer:

```
ADDI   R1, R0, 1000        # Put address 1000 into R1
ADDI   R2, R0, 2000        # Put address 2000 into R2
Load   R3, R1              # Load data at address 1000
Load   R4, R2              # Load data at address 2000
Store  R4, R1              # Store data that was in address 2000 into address 1000
Store  R3, R2              # Store data that was in address 1000 into address 2000
```

Name:___________________________

Problem 6: (TBD points)

We want to add a new instruction to the MIPS multi-cycle datapath that we have discussed in class.

For your reference, (i) a description of what happens during each cycle of execution, (ii) the finite state machine diagram, and (iii) a copy of the datapath are all attached to the end of this document. If it's helpful, feel free to remove this from the packet. *Nothing* has changed from the class and/or homework versions.

The instruction that will be added will copy a value from 1 register to another. The instruction syntax, register transfer language, and encoding are all shown below. Note that the encoding is quite similar to a MIPS addi instruction – which is also included for the sake of comparison.

Syntax: mov Rx, Ry
RTL: Rx $\leftarrow$ Ry

Encoding:

	Bits 31-26	**Bits 25-21**	**Bits 20-16**	**Bits 15-0**
Mov	opcode	source register	destination register	Unused / don't care
Addi	opcode	source register	destination register	Immediate value

Example:

- Assume Rx initially holds the number 10 and Ry initially holds the number 20.
- After this instruction finishes, both Rx and Ry will hold the number 20.
- The encoding for mov $1, $7 would look like this:

	Bits 31-26	**Bits 25-21**	**Bits 20-16**	**Bits 15-0**
Mov	opcode	00111	00001	xxxx xxxx xxxx xxxx

Based on this information answer the questions below.

Name:___________________________

<u>Question A</u>: (5 points)
It is possible to accomplish this *mov* instruction in just 3 clock cycles without using the ALU in clock cycle number 3 – although a small hardware change will be required. Moreover, no modifications to the current / existing *fetch* and *decode* steps are required. Explain why no changes are needed to the *fetch* and *decode* states.

Answer:

- We load data into temporary registers A and B in cycle 2.
- We can use the data in the **A register** (need to mention or refer to) in cycle 3; send it back to the register file

<u>Question B</u>: (5 points)
As mentioned above, we can finish this instruction in the 3rd cycle of execution without using the ALU. However, a very small hardware change is needed. What is it?

Answer:
We need to add a path from the A register to the multiplexor that precedes the register file data port.

<u>Question C</u>: (5 points)
Do we need to augment the finite state machine diagram – i.e. by adding or changing a state in cycle 3? If so, why? Note that you do not need to do this. Just explain ***if*** you need to do it and why.

Answer:
Yes. We want to write the register file with data from the temporary A register. Unique sets of control signals must be generated.

<u>Question D</u>: (5 points)
In Problem 1, the following code snippets were given for Function_1:

```
Function_1

        …
        …
        sub     $t1, $a0, $a1
        lw      $t2, 0($t1)
        …
        add     $a0, $t1, $0
        jal     Function_2
        …
        jr      <to main>
```

Will the *mov* instruction discussed in this problem improve the performance of this code?

Answer:
Yes. The add instruction can be replaced. We can save 1 CC.