Name:____________________________

CSE 30321 – Computer Architecture I – Fall 2010
Midterm Exam
October 14, 2010

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.

Grading will be done on Friday!

Name:____________________________

Score Sheet

Question	**Possible Points**	**Your Points**
1	15	
2	15	
3	20	
4	10	
5	15	
6	10	
7	15	
Total	100	

Name:_____________________________

Problem 1: (15 points, Course Goal #4)

This question deals with the 6-instruction ISA that was discussed in Lecture 02 and Lecture 03. As you saw in Lecture 02 and Lecture 03, the instruction encodings for the 6-instruction processor are as shown in the table below:

Instruction	Opcode	16-bit encoding				Function
Mov Ra, d	0000	Opcode **(4 bits)**	Destination Register **(4 bits)**	Address **(8 bits)**		RF[a]←M[d]
Mov d, Ra	0001	Opcode **(4 bits)**	Source Register **(4 bits)**	Address **(8 bits)**		M[d]←RF[a]
Add Ra,Rb,Rc	0010	Opcode **(4 bits)**	Destination Register **(4 bits)**	Source Register **(4 bits)**	Source Register **(4 bits)**	RF[a]←RF[b] + RF[c]
Mov Ra, #C	0011	Opcode **(4 bits)**	Destination Register **(4 bits)**	Constant **(8 bits)**		RF[a] ← c
Sub Ra,Rb,Rc	0100	Opcode **(4 bits)**	Destination Register **(4 bits)**	Source Register **(4 bits)**	Source Register **(4 bits)**	RF[a]←RF[b] - RF[c]
Jumpz Ra, X	0101	Opcode **(4 bits)**	Source Register **(4 bits)**	Offset **(8 bits)**		If RF[a] == 0, PC←PC+offset

Question A: (5 points)

Assume that you want to augment this ISA to support **20** additional and unique instructions (e.g. Mult, And, Or, etc.), while still keeping the instruction encoding as 16 bits. How will the execution and encoding of the Add instruction be affected? (Other instructions could be affected too, but you just need to comment on how the Add instruction will be impacted.)

Answer:

The number of bits dedicated to the opcode will need to increase from 4 to 5. As such, there will be one less bit to encode the number of the destination register or the number of one of the source registers. Given this change:

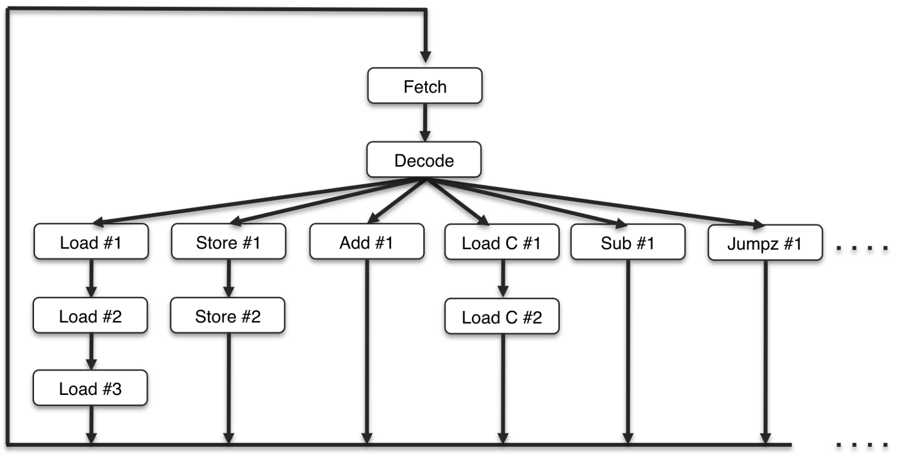
- An extra bit will be needed to specify the opcode.
- The result of the Add must be written to registers 0-through-7 *OR …*
- … one of the source registers will only be allowed to be register 0-through-7.

Name:______________________________

Question B: **(5 points)**
Now, assume that the 20 new instructions have all been added. Their addition has resulted in changes to the state diagram discussed in lecture. A portion of the new finite state diagram is shown below. *Given this new state diagram*, how many clock cycles will the code (that is also shown below) take to run?

Code:

```
        Mov       R1, #1
        Mov       R2, #2
        Mov       R4, #4
        Sub       R5, R4, R2
        Sub       R5, R5, R2
        Jumpz     R5, X
        Add       R1, R1, R1
X:      Add       R1, R2, R4
        Mov       10, R1
```



Answer:

	Mov	R1, #1	# R1 ← 1;	4 CCs
	Mov	R2, #2	# R2 ← 2;	4 CCs
	Mov	R4, #4	# R4 ← 4;	4 CCs
	Sub	R5, R4, R2	# R5 ← R4 – R2 = 2	3 CCs
	Sub	R5, R5, R2	# R5 ← R5 – R2 = 0	3 CCs
	Jumpz	R5, X	# Jump to X	3 CCs
	Add	R1, R1, R1	# SKIP	
X:	Add	R1, R2, R4	# R1 ← R2 + R4 = 6	3 CCs
	Mov	10, R1	# M(10) ← R1	4 CCs
Total:				28 CCs

Question C: **(5 points)**
In what clock cycles does R1 change state? (i.e. when is new data put into R1 available?)

Answer:
R1 changes state after the instruction "Add R1, R2, R4" is executed. This is in cycle 24 – which means that R1 data is available at the beginning of clock cycle 25.

It also changes state in CC #5.

Comments:
The most common mistakes I observed had to do with Parts B and C. If a mistake was made on Part B, it generally had to do with the fact that you either assumed that both Add instructions were executed, or miscounted the cycles from the state diagram. If a mistake was made on Part C, you generally assumed that data was available in the last cycle of execution – not the next cycle. (This is why the comment "when is data put into R1 available?" was included.)

Name:____________________________

Problem 2: (15 points, Course Goal #4)

Question A: (4 points)
Consider a hypothetical branch-if-equal instruction that is 32 bits long:

- 6 bits are used to encode the opcode
- 6 bits are used to encode one register number
- 6 bits are used to encode another register number
- 14 bits are used to encode an offset that will be added to the program counter (PC) if the branch ends up being taken, and a new instruction address is required.
    - (The number is not in 2s complement form, and all 14 bits can encode a constant.)

Thus, the instruction syntax might be: BEQ R12, R11, X

- If R12 == R11, the PC will be set to PC + X instead of PC + 4.

Given this instruction, is the code shown in the table below valid? Why or why not?

Address	Instruction
5000	…
5004	BEQ R12, R11, X
5008	Add R1, R2, R3
…	…
X: 21256	Sub R1, R2, R3

Answer:
Yes, this code is valid. The 14 bit offset allows you to encode a number that is as large as $2^{14} - 1$ (=16383). Thus, even if the PC has not yet been incremented, you can reach address 21384 (5004 + 16383), which is beyond address 21256.

Comments:
Most students got this question right.

Question B: (4 points)
Assuming that the PC has already been incremented by 4 when the comparison for the BEQ instruction at address 5004 is made, how many *instructions* away from the BEQ instruction could we reach?

Answer:
Floor($(2^{14}-1) / (2^2)$) = 4095 + 1 = 4096.

- The floor function is needed because the address must be a multiple of 4.

Comments:
Some assumed that the PC was not incremented and based their answer accordingly. This received full credit as long as your answer was explained correctly.

Name:____________________________

Question C: **(7 points)**
Assume that you have 24-bit instructions. A hypothetical "R-type" / ALU instruction (i.e. add, subtract, multiply, etc.) might be encoded as follows:

Opcode	Destination *and* Source Register	Source Register	Function Code
6 bits	6 bits	6 bits	6 bits

Thus, 1 register serves as both a source *and* a destination:
Add R5, R7 # R5 ← R5 + R7

Given this type of encoding – i.e. where one register is always both a source and a destination – can the code shown below be translated into assembly with just these types of ALU instructions? If yes, write your code. If no, explain what functionality is missing.

```
…
y = y + x;
z = z * q;
q = y + z;
z = z * y;
q = q + z;
…
```

Answer:
Yes. It is possible to come up with a sequence of instructions given the specific code shown below such that the operation is in fact possible. For example:

y = y + x
z = z * q
q = q * 0 (assume R0 = 0)
q = q + y
q = q + z
z = z * y
q = q + z

No. If the original value of q would be needed later at all, the register restrictions make it more difficult and some type of copy operation would be needed. Simply adding a number to R0 would not help in this case – as R0 always maps to 0 and cannot be overwritten.

Comments:
This question was intentionally designed to be somewhat open-ended. I was really looking to evaluate your reasoning more than your specific answer (which of course still needed to be correct!) For the particular sequence of instructions given, it is possible to use instructions where a register is both a source and a destination to evaluate the above code (see example above). Some looked at the code more "generically" and answered no. Provided a suitable explanation was given (like that shown above), students received full credit. The majority of students answered yes.

Name:____________________________

Problem 3: (20 points, Course Goal #2)

Assume that to spell check a large file, 820,000,000 instructions are needed. The instructions in the program are broken down into 4 different classes, and each class requires N clock cycles to execute. Specific information is given in the table below. (Here, N is the same as in the MIPS multi-cycle datapath discussed in class.)

Instruction Class	Clock Cycles per Instruction	Number of Instructions
Branch	3	150,000,000
Store	4	185,000,000
Load	5	260,000,000
ALU / R-type	4	225,000,000

Question A: (5 points)

If the total execution time for this program is found to be 1.57 seconds, what is the clock cycle time of the computer on which it was run?

Answer:

We can apply the CPU time formula: CPU time = IC x CPI$_{avg}$ x CC Time, and solve for CC Time.

$$820{,}000{,}000 \times \left[3\left(\frac{150{,}000{,}000}{820{,}000{,}000}\right) + 4\left(\frac{185{,}000{,}000}{820{,}000{,}000}\right) + 5\left(\frac{260{,}000{,}000}{820{,}000{,}000}\right) + 4\left(\frac{225{,}000{,}000}{820{,}000{,}000}\right)\right] \times N = 1.57s$$

Thus, the clock cycle time is 4.63 x 10^{-10} s. This is a 2.16 GHz machine.

Comments:

Many of you lost a point on this question because you provided a clock *rate* (i.e. Hz, GHz, etc.) instead of a clock cycle *time*. The units should be in seconds. Expect this topic to come up again on the final exam!

Name:___________________________

Question B: (10 points)
Assume that as part of the 820,000,000 instruction spell check, 25% of all load instructions are immediately followed by an ALU / R-type instruction that uses the data that was just loaded. To speed up this program, we are contemplating adding a new type of instruction – an ALU instruction where one of the source operands is a value from memory.

- This new instruction will replace the previous, 2 instruction sequence.
- It will take 7 clock cycles.

Will this change offer any speedup over the original design? If so, how much?

You may assume that the clock rate does not change and your answer to this question does not depend on your answer to Question A.

Answer:
We will need to apply the CPU time formula again, but first need to calculate the new number of load, ALU / R-type, and "new type" instructions:

- The # of branches remains constant:			150,000,000
- The # of stores remains constant:			185,000,000
- The new # of loads is:	(260,000,000 x 0.75)	=	195,000,000
- The new # of ALUs is:	(225,000,000 – 65,000,000)	=	160,000,000
- The number of new instructions is:	(260,000,000 x 0.25)	=	65,000,000
		Total:	**755,000,000**

Thus, the new CPU time is:

$$755{,}000{,}000 \times \left[3\left(\frac{150{,}000{,}000}{755{,}000{,}000}\right)+4\left(\frac{185{,}000{,}000}{755{,}000{,}000}\right)+5\left(\frac{195{,}000{,}000}{755{,}000{,}000}\right)+4\left(\frac{160{,}000{,}000}{755{,}000{,}000}\right)+7\left(\frac{65{,}000{,}000}{755{,}000{,}000}\right)\right]\times N$$

…which can be expressed as: 3,260,000,000N

This can be compared to a similar expression from Part A: 3,390,000,000N.

Thus, the speedup with the new design is ~4%.

Comments:
A few tried to apply Amdhal's Law (which did not work). Most realized that you needed to determine a new number of ALU, load, and new instructions and re-calculate CPU time. Even if other mistakes were made, I was really looking for you to show this thought process. If you did, you received a majority of points for this problem.

Name:____________________________

Question C: (5 points)
Qualitatively, if you see a speedup, where does it come from? If you do not, why not?

Answer:
The fetch/decode overhead is reduced for the extra instructions (i.e. 9 CCs vs. 7 CCs)

Comments:
Most got this question right. If you did Part B incorrectly, and your answer to Part C was reasonable, I gave you credit.

Name:____________________________

Problem 4: (10 points, Course Goal #3)
The number of instructions (of a given type) needed to encrypt and decrypt a message are as shown in the table below:

Instruction Class	Cycles Per Instruction	Encrypt	Decrypt
Branch	3	4,000,000	4,000,000
Store	4	10,000,000	9,000,000
Load	5	28,000,000	25,000,000
ALU / R-type	4	23,000,000	22,000,000
Totals:		**65,000,000**	**60,000,000**

You are considering changing the datapath that these benchmarks are run on so that a load instruction completes in 4 CCs instead of 5. Because the load instruction has to now do more work in a given clock cycle, that clock cycle will need to get longer. What clock cycle slow down is tolerable such that the performance of the encrypt and decrypt benchmarks is not degraded? You may assume that for every message decrypted, a message is also encrypted (and thus, there is equal use).

Answer:
We need to set the execution time for the base case equal to the execution time for the new case (with the slower clock rate) and find out what degradation is tolerable.

- We can simply add the number of instructions for encryption to the number of instructions for decryption.
- Instruction counts will cancel (as seen in the previous problems) and are thus not shown / used.

[(3)(8,000,000) + (4)(19,000,000) + (5)(53,000,000) + (4)(45,000,000)](X) =
[(3)(8,000,000) + (4)(19,000,000) + (4)(53,000,000) + (4)(45,000,000)](X * N)

(545,000,000 * X) = (473,000,000 * X * N)

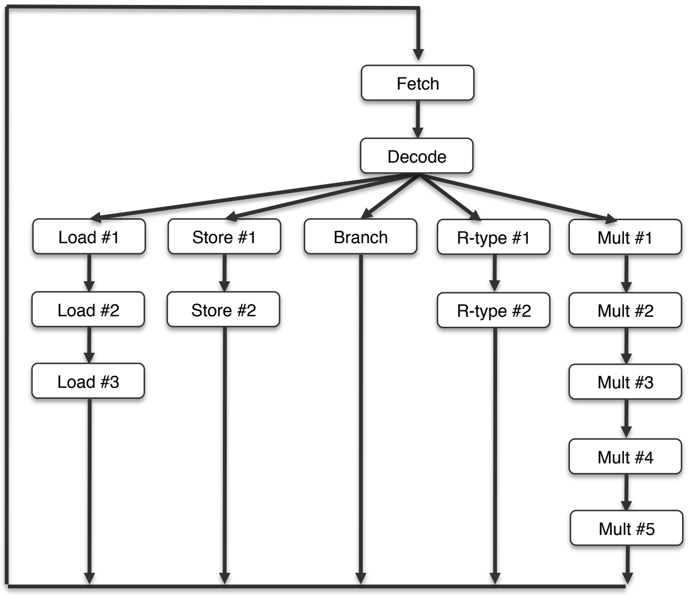
N = (545,000,000 * X) / (473,000,000 * X)
N = 1.15

Therefore, a slowdown of ~15% is tolerable.

Comments:
As seen from the stats, most got this question right. (In fact, the *median* was a 10.)

Name:______________________________

Problem 5: (15 points, Course Goal #5)
Below is a state diagram for a hypothetical processor.



Consider the following for loop:

```
for (i=0; i<N; i++) {
        x(i) = x(i) * 3;
}
```

Question A: (10 points)
Which MIPS translation – Version 1 or Version 2 – do you think is the most efficient? Why?

Version 1:

```
        addi    $1, $0, 1
        addi    $2, $0, N
X:      sll     $3, $1, 2
        add     $4, $3, $5
        lw      $5, 0($4)
        add     $5, $5, $5
        add     $5, $5, $5
        sw      $5, 0($4)
        addi    $1, $1, 1
        bneq    $1, $2, X
```

Version 2:

```
        addi    $1, $0, 1
        addi    $2, $0, N
        addi    $10, $0, 3
X:      sll     $3, $1, 2
        add     $4, $3, $5
        lw      $5, 0($4)
        mult    $5, $5, $10
        sw      $5, 0($4)
        addi    $1, $1, 1
        bneq    $1, $2, X
```

Answer:
It depends on N. If N is >= 5, Version 2 will always be most efficient.

Question B: (5 points)
Given your answer, how could you make the above MIPS code even more efficient?

Answer: Make the loop iterate in chunks of 4, not 1.

Name:____________________________

Comments:
For Part A, most mentioned that as N grew larger, Version 2 would be more efficient. You did not need to explicitly write that N should be greater than 5. A few of you assumed N was small and that Version 1 was more efficient. This received full credit. For Part B, many of you mentioned loop unrolling. I did not initially anticipate this answer, but it is a good one and received full credit. Generally, people did quite well on this problem. Again, the *median* was a 15.

Name:______________________________

Problem 6: (10 points, Course Goal #4)

Assume the following:

- Data for 4 arrays – A, B, C, and X – is stored in memory.
- You may assume that:
    - The data elements for array A are stored in sequential memory addresses.
    - The data elements for array B are stored in sequential memory addresses.
    - The data elements for array C are stored in sequential memory addresses.
    - The data elements for array X are stored in sequential memory addresses.
    - However, arrays A, B, and C **are not** stored sequentially in memory.
- The starting address of array A is contained in $1.
- The starting address of array B is contained in $2.
- The starting address of array C is contained in $3.
- The starting address of array X is contained in $4.

Write the MIPS assembly instruction for the following statement:

```
X[i] = A[B[i]] + C[B[i+4]];
```

To receive full credit, your answer should contain no more than 10 instructions. You may assume i maps to $5.

Please comment your code!

Answer:

1st, calculate the address of B[i]

```
0.      sll     $6, $5, 2               # multiply i*4
1.      add     $7, $2, $6              # add i*4 to the base address of B.
```

2nd, load B[i]

```
2.      lw      $8, 0($7)               # load B[i] into a register
```

3rd, load B[i+4]

```
3.      lw      $9, 16($7)              # load B[i+4] into a register; use start of i previously calculated
```

4th, load A[B[i]]

```
4.      add     $8, $8, $1
5.      lw      $10, 0($8)              # data in B[i] is in $8 and is address needed for A
```

5th, load C[B[i+4]]

```
6.      add     $9, $9, $3
7.      lw      $11, 0($9)              # data in B[i+4] is in $9 and is address needed for C
```

6th, do the add

```
8.      add     $12, $10, $11           # add A[B[i]] + C[B[i+4]]
```

7th, calxulate the address of X[i]

Name:______________________________

```
9.      add     $13, $4, $6         # we still have the old value of i*4 in $6

        8th, do the store
10.     sw      $12, 0($13)         # write data to memory address that maps to X[i]
```

Comments:
My solution here reflects a slightly different solution than that which is hinted at by the problem (i.e. create a 10 instruction solution). (The class last year could make an assumption about ix4.) The most common reasons for lost points usually involved improperly formatted lw and sw instructions. A very common mistake was related to instruction 3 in my solutions – i.e. to get i+4, you needed to use 16($x) rather than 4($x).

Name:___________________________

Problem 7: (15 points, Course Goal #4)
Consider the following C-code:

```
int main(void) {
     int i=0;                         # i maps to $s0
     int j=1;                         # j maps to $s1
     int k=2;                         # k maps to $s2
     int l, m, n, o, p, q;            # see below
     int x, y, z;                     # see below

     l = i+j+k;                       # l maps to $s3
     m = i*j*k;                       # m maps to $s4
     n = l-m;                         # n maps to $s5
     o = i+j;                         # o maps to $s6
     p = m-n;                         # p maps to $s7
     q = n+o-p;                       # q maps to $t1

     x = function_call_1(o,p,q);

     m = l+x+j;

     y = function_call_2(m,l,x);

     z = x+y;
}


int function_call_1(x,y,z) {
     int a;                           # a maps to $s1
     int b;

     a = x+y+z;
     b = function_call_3(a);

     return b;
}


int function_call_2(x,y,z) {
     int a;                           # a maps to $s1

     a = x-y-z;

     return a;
}


int function_call_3(x) {
     return x + x;
}
```

Name:______________________________

Question A: (3 points)
Write the MIPS assembly for the line "`return x+x;`" in function_call_3.

Answer:

```
add     $v0, $a0, $a0;
jr      $31
```

Question B: (3 points)
What stack operations – if any – take place inside of function_call_1? You can assume that function_call_1 is not a system call.

Answer:
- Save the $ra because of a nested function call.
- Save $s1 because it is used in main.

Question C: (3 points)
What MIPS instruction(s) would you expect to see before function_call_1 in main() – given the MIPS calling convention?

Answer:
- We would copy the data in $s6, $s7, and $t1 into argument registers – $a0, $a1, and $a2.
- We would call "jal function_call_1"
- We DO NOT need to save $t1 to the stack because we never use it again

Question D: (3 points)
What register would the variable `x` map to in the line "`x = function_call_1(o,p,q)`" in main()?

Answer:
- $v0

Question E: (3 points)
What stack operations – if any – must main() perform?

Answer:
- Data in variables x would have needed to be saved to the stack.
- Therefore, we would need to pop this value off to do "z=x+y"