CS2100 Computer Organization
2020/21 Semester 2
Mid-term Assessment (Backup)

1. This assessment paper is valid only when instructed to be used by the course coordinators, via the proctors.

2. Answer the questions on a piece of paper. Listen to instructions from the proctors on the ending time of the assessment.

3. When the proctors have told you to stop writing, you have 5 minutes to scan your answers using your mobile phone. You can use software like CamScanner, or simply take a picture. In either case ensure that your writing is legible.

4. Answers that cannot be read will not be marked and you will receive 0 for those questions.

5. When you have finished scanning, email your scans to colin.mod.mails@gmail.com.

6. All restrictions and instructions in the midterm SOP continue to apply.

7. There are FIFTEEN (15) questions on EIGHT (8) printed pages, including this one.

MRQ Questions

1. Given the hexadecimal value 0x3F2C, choose all of the possible interpretations of this value below:

   a. A 16-bit 1's complement negation of -16173
   b. +The decimal value 16172.
   c. A 16-bit 2's representation of 16173.
   d. +The ASCII characters '?' and ','
   e. The C string "?,"

Working:

0x3F2C = 0b0011 1111 0010 1100. This has 1's at bits 2, 3, 5, 8, 9, 10, 11, 12, 13

So integer value is 2^2 + 2^3 + 2^5 + 2^8 + 2^9 + 2^10 + 2^11 + 2^12 + 2^13
= 16172

0x3F is also ASCII '?' and 0x2C is ASCII for ','. This is not a C string as there is no NULL terminator.

2. We wish to implement the pseudo-instruction BLE $s1, $s2, target, which branches to "target" if $s1<=$s2. Choose all of the sequences below that implement this behaviour correctly (Note: We are only looking for correct behaviour. May not necessarily be the shortest possible implementation.)

   a. slt $t1, $s1, $s2
      bne $t1, $zero, target

   b. sub $t1, $s1, $s2
      slt $t2, $t1, $zero
      bne $t2, $zero, target

   c. +slt $t1, $s2, $s1
      beq $t1, $zero, target

   d. +sub $t1, $s2, $s1
      slt $t2, $t1, $zero
      beq $t2, $zero, target

   e. slt $t1, $s2, $s1
      bne $t1, $zero, target

Working: (a) and (b) will not branch if $s1 == $s2. (c) will branch if $s2>=$s1, or $s1<=$s2, hence this is correct. For (d), if $s2 >= $s1, then $s2 - $s1 >= 0, and this will branch. For (e), this will branch if $s2 < $s1, which is not $s1 <= $s2.

3. Aiken has written a program to multiply a number stored in register $s0 by 3 and store the result in register $s1. Sadly he lost the assembly code and only has the following fragments of MIPS machine code in hexadecimal. Help Aiken to find which instructions are from his program. The ordering of the instructions is not important, and we further assume that the register $s2 contains the value 3.

    a. 0x02128818
    b. +0x02308820
    c. 0x02008840
    d. 0x02118824
    e. **+0x00108840**

Working:

Option (a)

mult $s1, $s0, $s2 (Note: Incorrect use of mult – not covered in lecture but tests student ability to understand datasheet)

000000	10000	10010	10001	00000	011000

000000 10000 10010 10001 00000 011000
0000 0010 0001 0010 1000 1000 0001 1000
0x02128818

Countercheck:
0x02128818 = 0000 0010 0001 0010 1000 1000 0001 1000
000000 10000 10010 10001 00000 011000

Op=0=r-type, rs = $16=$s0, rt=$18 = $s2, rd = $17=$s1, shamt=0, funct = 0x18 = mult.

+Option (b)

add $s1, $s1, $s0

000000	**10001**	**10000**	**10001**	**00000**	**100000**

00000010001100001000100000100000
0000 0010 0011 0000 1000 1000 0010 0000
0x02308820

Counter-check:
0x02308820 = 0000 0010 0011 0000 1000 1000 0010 0000
000000 10001 10000 10001 00000 100000
Opcode = 0 = R-Type, rs = $17 = $s1, rt = $16 = $s0, rd = $17 = $s1, shamt = 0, funct = 0x20, so this is add $s1, $s1, $s0.

Option (c)

000000	10000	00000	10001	00001	000000

0000 0010 0000 0000 1000 1000 0100 0000
0x02008840
Incorrect as this does sll $s1, $zero, 1

Option (d)

and $s1, $s0, $s1

000000	10000	10001	10001	00000	100100

0000 0010 0001 0001 1000 1000 0010 0100
0x02118824

Countercheck:
0x02118824 = 0000 0010 0001 0001 1000 1000 0010 0100
000000 10000 10001 10001 00000 100100
Opcode = 0 = r-type, rs = $16 = $s0, rt = $17 = $s1, rd = $17 = $s1, shamt = 0, funct = 0x24 = and

+Option (e)

sll $s1, $s0, 1

000000	**00000**	**10000**	**10001**	**00001**	**000000**

0000 0000 0001 0000 1000 1000 0100 0000
0x00108840

Counter-check:
0x00108840 = 000000 00000 10000 10001 00001 000000
sll: R[rd] = R[rt] << shamt
Opcode = 0 = R-Type, RS does not matter, rt = register 16 = $s0, rd = register 17 = $s1, shamt = 00001 = 1, funct = 000000 = sll.

4. Pick all of the statements below that are TRUE about the MIPS datapath.

   a. A single cycle implementation is better than multicycle implementation since every MIPS instruction takes exactly the same amount of time to go through the datapath.
   b. +A multicycle implementation is better than single-cycle since not all instructions need to pass through every stage of the datapath.
   c. In an N stage datapath where each stage may take a different amount of time to complete, a multicycle implementation will give an N times speedup over a single cycle implementation for instructions that must pass through only one stage.

d. **+In a multicycle implementation where every stage takes the same amount of time to complete, the instruction that passes through all the stages will have the same execution time as in a single-cycle implementation.**
e. **+In a multicycle implementation where every stage may take a different amount of time to complete, the instruction that passes through all the stages will NOT have the same execution time as in a single cycle implementation.**

5. We wish to implement a no-operation (NOP) instruction in the MIPS datapath that, when executed, effectively does nothing except waste time (this sounds strange but is actually very useful in applications that must meet strict minimum timing requirements). Choose all the signal combinations that would correctly implement NOP (note: X = Don't care.). NOP is implemented as an R-format instruction with a function code of 0x3F. (Note: Here we only consider the specifications for the signal values; we do not consider actual implementation yet).

   a. **+RegDst=1, RegWrite=0, AluSrc=0, MemWrite=0, MemRead=0, MemToReg=0, PCSrc=0**
   b. RegDst=1, RegWrite=X, AluSrc=X, MemWrite=0, MemRead=0, MemToReg=X, PCSrc = 0
   c. RegDst=0, RegWrite=0, ALuSrc=1, MemWrite=0, MemRead=0, MemToReg=1, PcSrc=1
   d. RegDst=X, RegWrite=0, AluSrc=X, MemWrite=X, MemRead=0, MemToReg=X, PCSrc=0
   e. **+RegDst=X, RegWrite=0, AluSrc=X, MemWrite=0, MemRead=0, MemToReg=1, PCSrc=0**

   Reasoning: NOP works as long as RegWrite, MemWrite are both 0, since nothing (register nor memory) will be changed.

   Although the function code is 0x3F (and hence the Immed value is 0x3F), this will not add (4 x 0x3F) to PC+4 even if PCSrc = 1, because branch will be 0 since this is not a branch (beq/bne) instruction.

MCQ Questions

Section 1 – Number Systems

1. To find the 10's complement of an N-digit decimal number Y, we do $10^N - Y$. Which ONE of the following statements is TRUE?

    a. In an N-digit 10's complement number system, the leftmost digit has a weight of 10^N.
    b. In an N-digit 10's complement number system, the leftmost digit has a weight of 10^{N-1}**.**
    c. In an N-digit 10's complement number system, the leftmost digit has a weight of -10^N (negative 10^N)
    d. In an N-digit 10's complement number system, the leftmost digit has a weight of -10^{N-1} (negative 10^{N-1}).
    e. +None of the choices in this question (except this one) are true.

Unlike binary numbers, the leftmost digit does not have a weight, but rather the complement X' of an N-digit 10's number X is defined just as the additive inverse of X. I.e. $X + X' = 10^N$.

Taking the 10's complement of 1 (i.e. finding -1), we have $10^4 - 1 = 9999$.
We can see that none of the four choices are correct as the largest digit possible is 9 (not 1 as in binary), and hence here we get, for example, -9 x 10^3 + 9 x 10^2 + 9 x 10^1 + 9 = -8001 which is not equal to -1.

2. In a 4-bit 1's complement number system, doing -3 - 6 (negative 3 minus 6) will give us:

    a. -9
    b. 5
    c. +6
    d. -5
    e. -6

Straightforward. -3 = 0b1100, -6 = 0b1001
$-3 - 6 = 1100_2 + 1001_2 = 10101_2$, add in the end-round carry = $0101_2 + 1_2 = 0110_2 = 6$. This is an overflow but the student is not required to state that, but just to work out the value.

Section 2 – C Programming

1. What does the following C function do? Assume that a long is twice the size of an int.

```
   unsigned long mystery(unsigned int x, unsigned int y) {
      unsigned long a = 0, b = x;

      while(y) {
                  if(y & 0b1) {
                              a = a + b;
                  }

                  b = b << 1;
                  y = y >> 1;
      }

      return a;
}
```

   a. Adds x and y.
   b. Shifts b to the right by 1 bit and y to the left by 1 bit and adds them together.
   c. Produces an even parity for x and y.
   d. +**Multiplies x and y.**
   e. Divides x by y.

This does a long binary multiply of x and y. E.g. 15 x 3 is:

$$
\begin{array}{r}
00101_2 \\
\times\ 00011_2 \\
\hline
00101_2 \\
00101_2 \\
\hline
001111_2
\end{array}
$$

This code works by sampling the rightmost bit of y, and if it is a 1, it adds x to a sum (initially 0). It then left shifts x (accomplishing the shift we see above), then right-shift y to see if the following bit is also 1. Each time it sees a '1' in the LSB after right shifting y, it adds x, then left-shifts x. This process repeats until there are no more '1' bits in y (i.e. y is now 0) and then it exits since it won't add x anymore to the sum.

2. What does the following C function do?

```
unsigned char mystery2(unsigned char x, unsigned char y) {
    unsigned char a = x;
    unsigned char b = y;
    unsigned char c = 1;
    unsigned char d = 0;

    while(c) {
        if((a & 1) != (b & 1)) {
            d = d + c;
        }

        a = a >> 1;
        b = b >> 1;
        c = c << 1;
    }

    return d;
}
```

a. Shifts a and b left by 1 bit and c right by 1 bit, then adds c to d.
b. +Does a bit-wise XOR of x and y.
c. Checks whether both x and y are equal.
d. The function will never exit.
e. Does a bit-wise division of x and y.

The "if" statement checks if the rightmost bits are the same, and if so it adds c to d. Both a and b are then right shifted by 1 to check that bit, and c is left shifted by 1 bit, so that we can add in that bit if the two bits in a and b are different. Effectively does an XOR.

Section 3. Assembly Language

Dueet wrote the following code in MIPS assembly to process some elements of an array whose base address is in $s2.

1		addi $s0, $zero, 0
2		addi $s1, $zero, 0
3		addi $t0, $s2, 0
4		addi $t1, $s2, 40
5	x:	slt $t2, $t0, $t1
6		beq $t2, $zero, z
7		lw $t3, 0($t0)
8		addi $t0, $t0, 4
9		andi $t3, $t3, 1
10		beq $t3, $zero, y
11		addi $s0, $s0, 1
12		j x
13	y:	addi $s1, $s1, 1
14		j x
15	z:	

1. What does this code do?

    a. It counts the number of array elements divisible by 2 and number of items divisible by 3 and stores the results in $s0 and $s1 respectively.
    b. It counts the number of array elements with even parity and odd parity and stores the results in $s0 and $s1 respectively.
    c. +It counts the number of odd and even array elements and stores the results in $s0 and $s1 respectively.
    d. It counts the number of array elements with even parity and odd parity and stores the results in $s1 and $s0 respectively.
    e. It counts the number of array elements that are zero and non-zero and stores the results in $s0 and $s1 respectively.

2. Dueet's software license for her assembler has expired, and she decided to "hand-assemble" the code above by consulting the datasheet and translating the assembly code into machine code. However when she ran her code she found that her code always terminates with $s1=0 regardless of the contents of the array, though $s0 is correct. What is the likeliest problem?

    a. The j statement at line 12 was encoded to jump to line 2 instead of line 5.
    b. **+She calculated the offset for the branch instruction at line 10 based on the instruction at line 10 instead of line 11.**
    c. She calculated the offset for the jump statement at line 14 using the address at line 15 instead of the address at line 13.

d. She encoded the andi instruction at line 9 to do “andi $t3, $t3, 0” instead of “andi $t3, $t3, 1”.
e. She calculated the offset for the branch at line 10 based in bytes rather than words.

Option (a) would have resulted in an infinite loop and the code would not exit. Option (b) would result in Dueet calculating that the beq should jump 3 instructions instead of 2, causing the beq to branch to line 14 instead of 13, causing the addi to $s1 to be skipped, resulting in $s1 remaining 0. Hence this is the correct option.

Option (c) is nonsensical; jump addresses are absolute. Option (d) would result in $s0 being 0 and $s1 = 10 (# of elements in the array), since the the andi will always result in 0 and hence the branch will always be taken, causing the addi $s0, $s0, 1 to be skipped . Option (e) would result in unpredictable behaviour as the offset would be 4x too large, and hence it is unlikely that $s0 would be correct across multiple runs.

Section 4. Fill In The Blanks (Expanding Opcodes)

In this section we assume a processor with a fixed 16-bit instruction length, 16 registers, and the following 3 instruction classes:

Class A: Two registers.
Class B: One register.
Class C: One register, one 8-bit immediate value.

In all cases we assume that the encoding space for opcodes is fully utilized.

WORKING:

Class A:

8 bits	4 bits	4 bits
Op	Reg1	Reg2

Class B:

12 bits	4 bits
Op	Reg1

Class C:

4 bits	4 bits	8 bits
Op	Reg1	Immed

Most restrictive C->A->B
Least restrictive B->A->C

Minimum number:

Maximize class C, but set aside 1111 for class A and B. So class C opcodes = 0000 to 1110 = 15 instructions.

Maximize class A, but set aside 11111111 for class B. So class A opcodes = 11110000 to 11111110 = 15 instructions.

Maximize class B, so class B opcodes = 111111110000 to 111111111111 = 16 instructions.

Total = 46 instructions.

Maximum number:

Start with class B. # of opcodes = 2^12 = 4096.
Set aside 1111 xxxx yyyy for class C. This is 2^8 = 256 opcodes.
Class B ranges from 0000 0000 0000 to 1110 1111 1111.
Set aside 1110 1111 zzzz for class A = 2^4 = 16 opcodes.
Total number of opcodes = 4096 – 256 – 16 + 1 + 1 = 3826 opcodes.

1. The minimum number of opcodes that can be encoded on this machine is **46**.

2. The maximum number of opcodes that can be encoded on this machine is **3826.**

Section 5. Fill In The Blanks (Floating Point Numbers):

We have a 16-bit floating point number system with the following format:

1 sign bit for the mantissa	6 bit exponent in 2's complement. No reserved bit patterns with special meanings.	9 bit normalized mantissa, no hidden bit, in sign and magnitude representation.

The leftmost bit and the rightmost 9 bits form a sign-and-magnitude mantissa. Answer the following questions, filling in the **exact** numeric answer each time.

1. Within the representation range of this number system (i.e. between the most negative and most positive numbers that can be represented) , the number **0** cannot be represented.

2. The most negative number that can be represented in this number system is **-4286578688**.

Working:

1. **The leftmost mantissa bit is always 1 because it is normalized, hence it is not possible to represent 0. Note that a fundamental property of 0 is that regardless how many times it is added to (or subtracted from) any number, that number always remains completely unchanged. An approximation of 1×2^{-32}, while close to 0, does not have this property and hence is not acceptable.**

2. **The largest exponent is $2^5 - 1 = 31$. The most negative number is $-1.11111111 \times 2^{31} = -4286578688$.**