CS2100 2021/22 Semester I

Backup Midterm Paper

1. This paper is only to be used in the event of a serious LumiNUS failure. Otherwise this paper should be completed online in the LumiNUS Quiz Tool.

2. This is an open-internet paper. You can use any tool (compilers, code converters, online forums, etc.), as long as you **DO NOT COMMUNICATE WITH ANYONE NOR RECEIVE COMMUNICATIONS FROM ANYONE. IF YOU DO SO YOU WILL BE DEEMED TO HAVE CHEATED AND SERIOUS DISCIPLINARY ACTION WILL BE TAKEN AGAINST YOU.**

3. You may ask questions at this link: https://forms.gle/1vqjA8huxLfkh3MQA. Your questions must be phrased in a way that can be answered with a "YES" or a "NO", with no further elaboration. If your question is not appropriately phrased, you may be asked to rephrase, or you may be ignored.

4. This paper consists of FIFTEEN (15) questions on SEVEN (7) printed pages including. Marks are indicated, and total 40 marks, forming 20% of your final course grade.

5. You may complete this paper electronically on your tablet, print it out and write down your answers, or simply write your answers on a blank piece of paper. If you use a blank piece of paper, ensure that your answers are properly labeled so that we can identify which question (or which part of a question) your answer belongs to.

6. This paper is 60 minutes long.

7. **ENSURE THAT YOUR NAME, STUDENT NUMBER AND TUTORIAL GROUP NUMBER ARE CLEARLY WRITTEN ON YOUR ANSWER SHEET. IF YOU FAIL TO WRITE ANY OF THESE YOUR PAPER MAY NOT BE GRADED AND YOU WILL RECEIVE ZERO FOR THIS ASSESSMENT!**

8. At the end of the assessment, submit your answers as a PDF named TYY_Axxxxxx.pdf, where TYY is your tutorial group number, and Axxxxxx is your student number. **IF YOU SUBMIT USING AN INCORRECT NAMING, YOUR SCRIPT MAY NOT BE GRADED AND YOU WILL RECEIVE ZERO FOR THIS ASSESSMENT!**

9. IF LUMINUS IS AVAILABLE, Submit to your personal folder in the MIDTERMS->Txx folder, where Txx is your tutorial group number. If LumiNUS is not available, email your script, PROPERLY NAMED TYY-Axxxxxx.pdf, to cs2100.papers@gmail.com. Here TYY is your tutorial group number and Axxxxx is your student number. **REMINDER: IF YOU SUBMIT USING AN INCORRECT NAMING, YOUR SCRIPT MAY NOT BE GRADED AND YOU WILL RECEIVE ZERO FOR THIS ASSESSMENT!**

10. All rules in the Standard Operating Procedures are to be observed. Failure to do so may result in disciplinary action being taken against you.

NAME: _____ STUDENT NUMBER: _____ TUTORIAL GROUP: _____

1. We are given the following addition in an unknown base x (2 marks):

   1231_x
   $+ 3231_x$

   10012_x

   The unknown base x is base **5.**

Check:

$1231_5 = 191$

$3231_5 = 441$

$191 + 441 = 632$

$10012_5 = 632$

2. We are given the following subtraction in an unknown base y (2 marks):

   3231_y
   $- 1114_y$

   2113_y

   The unknown base y is base **6**.

Check:

$3231_6 = 739$

$1114_6 = 262$

$739 – 262 = 477$

$2113_6 = 477$

3. Fill in the unknown base w in the following conversion (2 marks):

   $3237_w = 1695_{10}$

   The unknown base w is base **8.**

Check: $3237_8 = 3 \times 8^3 + 2 \times 8^2 + 3 \times 8^1 + 7 = 1536 + 128 + 24 + 7 = 1695$

4. We have the following conversion from binary to an unknown base v (2 marks):

   $10101110111_2 = 2567_v$

   The unknown base v is base **8.**

Check:

$10101110111_2 = 10\ 101\ 110\ 111_2 = 2\ 5\ 6\ 7_8$

5. We consider a 24-bit signed number system in excess-1048576. In this number system, there are **1048576** negative numbers and **15728640** non-negative numbers (2 marks).

Check: Excess-1048576 means the most negative number is -1048576. From -1048576 to -1 there are 1048576 numbers. With 24 bits we have 2^24 = 16777216 numbers, of which 1048576 are negative, so 16777216 – 1048576 = 15728640 numbers are non-negative.

6. Find the 4-digit 3's complement representation in base 4, of the following decimal numbers. 3's complement in base 4 is the diminished radix complement for base 4 (4 marks).

   112_{10} = **1300** $_{3s}$

   -114_{10} = **2031** $_{3s}$

Check:

$112 = 1300_4 = 1300_{3s}$
$114 = 1302_4$; Therefore, $-114 = ((4^4 - 1) - 1302)_{3s} = (3333 - 1302)_{3s} = (2031)_{3s}$

7. Fill in the blank below in 3's complement (2 marks):

   $112_{10} - 114_{10}$ = **3331** $_{3s}$

$112 - 114 = -2 = ((4^4 - 1) - 2)_{3s} = (3333 - 2)_{3s} = 3331_{3s}$

Note: Complement number systems are always defined over a fixed number of digits, e.g. 4-bit 2's complement, 3 digit 6's complement, etc.

The number of digits must be sufficient to cover all numbers in the equation. Here the minimum number of digits to cover 112 and 114 is 4 digits. Hence any answer with fewer than 4 digits is incorrect.

Aiken has discovered that, for signed integers, the ">>" operator in C actually does an "arithmetic shift right" rather than a "logical shift right" like the srl operator in MIPS. What this means is that rather than filling the left bits with 0, the ">>" operator fills the vacated left bits with the sign bit.

On a 32-bit system, let's take:

```
int x = 0x80000000;
printf("%x\n", x); // Prints out 0x80000000, i.e. 0b1000 0000 .... 0000
printf("%x\n", x >> 1); // Prints out 0xc0000000, i.e. 0b1100 0000 ... 0000
printf("%x\n", x >> 2); // Prints out 0xe0000000, i.e. ob1110 0000 ... 0000
etc.
```

Aiken wants to write a function called "shift_right" which will do a logical right shift of signed integers instead of an arithmetic right shift. His best friend Dueet says that the best way to do this is to use "type-casting".

However, Aiken didn't learn "type-casting" in CS2100, so he wants to use bitwise logical operators like & and |. This is his attempt:

```
int right_shift(int val, int shift) {
// Returns the logical right shift of "val" by "shift" bits
    int mask1 = 0x80000000;
    int mask2 = *Mystery Instruction 1*

    return *Mystery Instruction 2*
}
```

General strategy; form a mask that zeros the leftmost shift bits. So, we must form a mask that has 0's in the leftmost shift bits, and 1's elsewhere.

8. What is "Mystery Instruction 1"? (2 marks):

   Mystery instruction 1 = ~(mask1 >> (shift - 1));

   Take shift = 3. Initially mask1 = 0x1000 0000 0000 0000 0000 0000 0000 0000

   mask1 >> (shift - 1) = mask1 >> 2 = 0x1110 0000 0000 0000 0000 0000 0000 0000

   ~(mask1 >> 2) = 0x0001 1111 1111 1111 1111 1111 1111 1111

   a. ~(mask1 >> shift);
   b. ~(mask1 << shift);
   c. (mask1 >> (shift-1));
   d. ~(mask1 >> (shift-1));
   e. ~(mask1 << (shift -1));

9. What is “Mystery Instruction 2”? (2 marks):

   Now we shift val itself and AND against our mask:

   Mystery instruction 2 = ((val >> shift) & mask2);

   E.g. let val = 0xBF000000 = 0b1011 1111 0000 0000 0000 0000 0000 0000

   We right shift by 3 positions: 0b1111 0111 1110 0000 0000 0000 0000 0000

   & with mask2 = (0b1111 0111 1100 0000 0000 0000 0000 0000) & (0b0001 1111 1111 1111 1111 1111 1111 1111) = 0b0001 0111 1100 0000 0000 0000 0000 0000

   a. (val & mask2);
   b. ((val >> shift) & mask2);
   c. ((val >> shift) && mask2);
   d. (val | mask2)
   e. ((val >> shift) | mask2)

We consider the MIPS assembly language program below, which processes two arrays, Array 1 and Array 2, both with the same number of elements. The outer loop reads from Array 1, the inner loop reads from Array 2.

```
        addi $5, $zero, 0
        addi $6, $2, 0
        addi $7, $3, 0
        sll $8, $4, 2
        add $9, $2, $8
        add $10, $3, $8
a:      slt $11, $6, $9
        beq $11, $zero, e
        lw $11, 0($6)
b:      slt $12, $7, $10
        beq $12, $zero, d
        lw $12, 0($7)
        bne $11, $12, c
        addi $5, $5, 1
c:      addi $7, $7, 4
        j b
d:      addi $6, $6, 4
        addi $7, $3, 0
        j a
e:
```

Within each array, numbers are never repeated. So Array 1 might hold 10, 15, 12, 92, 5, while Array 2 might hold 92, 3, 101, 10, 78. However you will not have the case where, for example, Array 1 holds 10, 15, 12, 10, 5, because here 10 is repeated.

10. Choose ALL the statements that are correct about this program (2 marks).

We can see the following:

$2 and $3 are base addresses of the two arrays. The number of elements is in $4, $5 is a counter, $6 and $7 are addresses of elements currently being accessed, $9 and $10 are terminating addresses for the arrays.

Since there's only one register that contains the # of elements, we assume that both arrays contain the same number of elements.

a. Register $2 and $3 contain the base addresses of the two arrays.
b. Register $8 contains the number of elements in Array 1.
c. Register $6 contains the address of the current element being processed from one of the arrays.
d. The inner and outer loops run the same number of times.
e. Register $9 contains the address of the final element of Array 1 to be loaded.

11. What is the MINIMUM number of instructions executed, if this program processes all the elements in two 10-element arrays? Fill your answer here: **688** (2 marks).

Inner loop:

```
b:      slt $12, $7, $10
        beq $12, $zero, d
        lw $12, 0($7)
        bne $11, $12, c
        addi $5, $5, 1
c:      addi $7, $7, 4
        j b
```

In the MINIMUM number of instructions, the bne is always taken so the addi $5, $5, 1 is never executed. On each iteration we therefore execute 6 instructions (slt, beq, lw, bne, addi $7, j). With 10 elements there are 10 iterations giving us 6 x 10 = 60 instructions. On the 11^{th} iteration only the slt and beq are executed, giving us 60 + 2 = 62 instructions total.

Now including the outer loop:

```
a:      slt $11, $6, $9
        beq $11, $zero, e
        lw $11, 0($6)
b:      slt $12, $7, $10
        beq $12, $zero, d
        lw $12, 0($7)
        bne $11, $12, c
        addi $5, $5, 1
c:      addi $7, $7, 4
        j b
d:      addi $6, $6, 4
        addi $7, $3, 0
        j a
```

On the outer loop, we will execute slt, beq, lw, the inner loop, addi $6, addi $7, j for 10 times. This is (6 + (# of inner loop insturctions)) x 10 = (6 + 62) x 10 = 680 instructions. Again on the 11th iteration only the slt and beq at b: are executed. This gives us 682 instructions executed.

Finally, we have 6 instructions before the outer loop giving us 682 + 6 = 688 instructions.

What is the MAXIMUM number of instructions executed, if this program processes all the elements in two 10-element arrays? Fill your answer here: **698** (2 marks).

For the MAXIMUM number of instructions executed, the bne $11, $12, c is not taken each time there is a match, so the addi $5, $5, 1 will executed, adding one extra instruction per match. Since numbers are never repeated, there will be only one match per complete execution of the inner loop, for each value read in the outer loop. Since 10 values are read in the outer loop, there will be 10 matches in total in the inner loop. Hence the addi $5, $5, 1 instruction will be executed 10 times.

So we have 688 + 10 = 698.

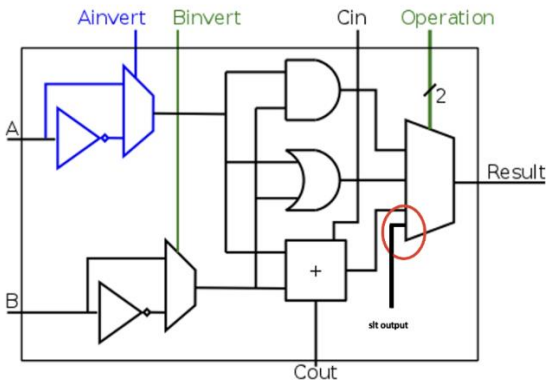
12. What does $5 contain at the end of the execution of this program (2 marks)?

a. $5 contains the number of elements in Array 1 that are also in Array 2.
b. $5 contains the number of elements that are in Array 1 but not in Array 2.
c. $5 contains the number of elements that are in Array 2 but not in Array 1.
d. $5 contains the differences between elements in Array 1 and Array 2.
e. $5 returns the sum of the elements in Array 1 and Array 2.

13. The MIPS slt instruction takes the form slt $rd, $rs, $rt, where [$rd] = 1 if [$rs] < [$rt], and 0 otherwise.

    We recall that the full 32-bit ALU is made up of single-bit ALU slices, with the carry-out (Cout) of bit 0 connected to the carry-in (Cin) of 1, the Cout of bit 1 connected to Cin of bit 2, etc.

    We will now implement slt using the last input (input 0b11) of the output multiplexer of each ALU slice. For brevity we will call this input "Input ob11 of the bitX ALU slice", where X is from 0 to 31. Input 0b11 for one slice is circled below.

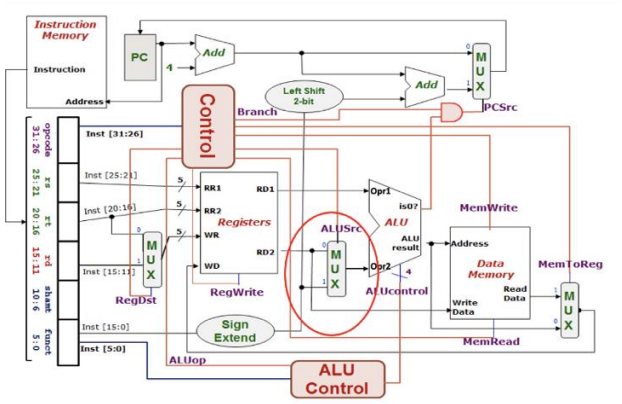


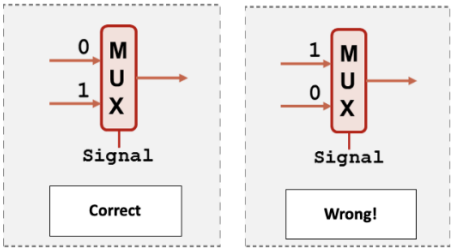
Choose ALL of the options needed to implement the slt instruction (4 marks):

General strategy: If A<B, then A-B is negative and the output of the adder for the bit31 slice will be 1. Hence we set Ainvert to 0, Binvert to 1, Operation to 0b11 (since this is where we are connecting the SLT output to), giving ALUControl = 0b0111. We then connect the output of the adder for bit31 ALU slice to the 0b11 input of the ALU slice for bit 0, and connect 0's to all the other 0b11 inputs. This gives us an ALU output of 1 if A<B, and 0 if A>=B, which is exactly what we want for slt.

a. Set ALUControl to 0b0011.
b. Set the 0b11 input of the ALU slice for bit31 to the output of the adder of the bit 0 ALU slice.
c. Set ALUControl to 0b0110.
d. Set the 0b11 input of the ALU slice for bit 0 to the output of the adder of the bit31 ALU slice, and the 0b11 inputs of the remaining ALU slices to 0.
e. Set ALUControl to 0b1011
f. Set ALUControl to 0b0111
g. Set the 0b11 input of the ALU slice for bit 0 to 1, and the 0b11 inputs of the remaining ALU slices to 0.
h. Set the 0b11 inputs of all ALU slices to the output of the adder of the bit 31 ALU slice.

14. In one of our tutorial questions, Mr. De Blunder accidentally swapped the inputs of the RegDst multiplexer. That problem has since been fixed. Unfortunately, now Mr. De Blunder accidentally swapped the inputs of the ALUSrc multiplexer!





Each register has been initialized to contain its own register number. Example: register $1 contains 1, register $2 contains 2, register $3 contains 3, etc.

Find the results, in decimal, of the following instructions (2 marks each, total 8 marks):

a. addi $3, $2, 8192

The result **5** will be written to register $3.

b. sub $4, $5, $6

The result **-8221** will be written to register $4.

c. andi $7, $8, 0b1111

The result **0** will be written to register $7.

d. add $9, $10, $11

The result **18474**will be written to register $9.

Working:

The effect of swapping the ALUSrc inputs is that the OPR2 input of the ALU will receive bits 0-15 of an R-type instruction, and the contents of RT for I-type instructions. Thus for R-type instructions we need to work out the binary encoding, then take the rightmost 16 bits as the second input to the ALU.

a. addi $3, $2, 8192

   Here [$3] will be sent to the ALU instead of 8192, resulting in the result 3 + 2 = 5 written to $3.

b. sub $4, $5, $6

   Binary encoding:

   rs = $5 = 00101, rt = $6 = 00110, rd = $4 = 00100, opcode = 000000, func = 0x22 = 100010

   000000 00101 00110 | 00100 00000 100010

   Last 16 bits = 0010 0000 0010 0010 = 0x2022 = 8226

   So we have $4 = $5 – 8226 = 5 – 8226 = -8221

c. andi $7, $8, 0b1111

   Here we will be doing [$8] AND [$7] (i.e. rs AND rt) instead of [$8] AND 0b1111. Thus we have 0b1000 & 0b0111 = 0.

d. add $9, $10, $11

   rs = $10 = 0b01010, rt = $11 = 0b1011, rd = $9 = 0b01001, opcode = 000000, funct=0x20 = 100000

   Thus binary encoding = 000000 01010 01011 | 01001 00000 1000000

   Last 16 bits = 0b0100 1000 0010 0000 = 0x4820 = 18464

   We add 10 = 18464 and get 18474.