CS/EE 3810	**Name (Print):**	
Fall 2022	**uid:**	
Midterm	**Left person:**	
10/24/2022	**Right person:**	

Time Limit: 11:50am - 1:10pm

- **Don't forget to write your name on this exam.**
- **This is an open book, open notes exam. But no online or in-class chatting.**
- **Ask us if something is confusing in the questions.**
- **Organize your work**, in a reasonably neat and coherent way, in the space provided. Work scattered all over the page without a clear ordering will receive very little credit.
- **Mysterious or unsupported answers will not receive full credit**. A correct answer, unsupported by explanation will receive no credit; an incorrect answer supported by substantially correct explanations might still receive partial credit.
- If you need more space, use the back of the pages; clearly indicate when you have done this.

Problem	Points	Score
1	20	
2	45	
3	10	
4	20	
5	25	
Total:	120	

1. Performance equation

   Compilers can have a profound impact on the performance of an application. Assume that for a program, compiler A results in a dynamic instruction count of 1.0E9 (i.e., counted by the CPU with a hardware performance counter during execution of the program) and has an execution time of 1.1 s, while compiler B results in a dynamic instruction count of 1.2E9 and an execution time of 1.5 s.

   (a) (10 points) Find the average CPI for each program given that the processor has a clock cycle time of 1 ns.

   Answer: We know that,

   $CPUtime = InstructioncountCPIClockcycletime$

   So,

   $CPI = CPUtime/(InstructioncountClockcycletime)$

   Compiler A:

   $CPI = 1.1/(1.0E91.0E-9) = 1.1$

   Compiler B:

   $CPI = 1.5/(1.2E91.0E-9) = 1.25$

   (b) (10 points) Assume the compiled programs run on two different processors. If the execution times on the two processors are the same, how much faster is the clock of the processor running compiler As code versus the clock of the processor running compiler Bs code?

   Answer: We know that, $1/Clockcycletime = InstructioncountCPI/CPUtime$

   So,

   $Clockrate(Frequency) = InstructioncountCPI/CPUtime$

   For the same CPU or execution time on both processors (here we assume that CPI of each program stays the same as in part above),

   $fB/fA = (Instructioncount(B)CPI(B))/(Instructioncount(A)CPI(A)) = (1.2E91.25)/1.0E91.1) = 1.36$

2. MIPS assembly

   (a) (5 points) Which address is accessed by the following `lw` instruction, `$s0` contains 1234?

   ```
   lw $s0, 4000($s0)
   ```

   Answer: $4000 + 1234 = 5234$

   (b) (5 points) The address field of the jump (`j`) instruction contains 1234? What is the value of the `pc` (program counter) register after this jump instruction is executed? The initial value of the `pc` is 2147483648.

   Answer: We will take a simple answer $1234 * 4 = 4936$, but will give extra 5 points for mentioning that `j` uses *pseudodirect* addressing which means that the top 4 bits of the PC are not changed, hence the address is 2147488584.

   (c) (5 points) The address field of the branch when equal (`beq`) instruction contains 1234, and the value of the program couner is 4000. What is the value of the `pc` (program counter) register after this `beq` instruction is executed (assume the condition is true and the branch is taken)?

   Answer: $4000 + 4 + 1234 * 4 = 8940$

(d) (10 points) Below is the source code a simple C program translate it into MIPS assembly (assume `a` is in `t0`, `b` is in `t1`, `c` is in `t2`, `d` is in `t3`, `e` is in `t4`, and `f` should be in `s0`.

```
f = foo(a, b, c, d, e);
```

I.e., you only have to write assembly for calling the function and getting the result back (you don't really care what the function is doing).

However, you need to know the signature of the function, `foo` function takes 5 integers and returns an integer, i.e.:

```
int foo(int a, int b, int c, int d, int e);
```

Answer:

```
mov $a0, $t0      # first 4 arguments are in registers
mov $a1, $t1
mov $a2, $t2
mov $a3, $t3
addi $sp, $sp, -4 # stack grows down
sw $t4, $sp       # 5th arg is on the stack
jal foo
mov $s0, $v0
addi $sp, $sp, 4  # restore the stack
```

(e) (20 points) Below is the source code a simple C program translate it into MIPS assembly

```
int foo(int a) {
        int sum = 0;
         for (int i = 0; i < a; i++)
            sum = sum + i;
        return sum;
}

int bar(int n)
{
    return foo(n);
}
```

Answer:

```
foo:
        # this is a leaf function, no need to save/restore $ra
        move $v0, $zero         # sum is in $v0
        move $t0, $zero         # i is in $t0

  forloop:
        slt $t1, $t0, $a0       # set $t1 to 1 if $t0 is less then $a0 (i < n)
        beq $t1, $zero, exit    # $t1 is 0 (not set) only when i >= n

        add $v0, $v0, $t0       # sum = sum + i

        addi $t0, $t0, 1        # i += 1
        j forloop
  exit:
        jr $ra                  # return, sum is already in $v0

bar:
    addi $sp, $sp, -4 # stack grows down (reserve space for $ra)
    sw $ra, $sp       # save $ra
    jal foo           # call foo
    lw $ra, $sp       # restore $ra
    addi $sp, $sp, 4  # restore stack
    jr $ra            # return
```

3. Linking and loading: relocation

   Alice compiles the following C file.

```
#include<stdio.h>

int add (int a, int b) {
    printf("Numbers are added together\n");
    return a+b;
}

int main() {
    int a,b;
    a = 3;
    b = 4;

    int ret = add(a,b);

    printf("Result: %u\n", ret);
    return 0;
}
```

   (a) (10 points) Which symbols need to be relocated? Note, that C treats string constants as globals allocated in the read-only data section. Explain your answer.

```
#include<stdio.h>

int add (int a, int b) {
    printf("Numbers are added together\n"); # access to string in the data section
                                            # requires relocation,
                                            # i.e., move $a0, <str_addr>
    return a+b;
}

int main() {
    int a,b;                                # a and b are local (on the stack or
                                            # in registers
    a = 3;                                  # no need to relocate
    b = 4;

    int ret = add(a,b);                     # invocation of a function "add" (i.e.,
                                            # jal <some addr>)
                                            # ret is local (on the stack or in
                                            # registers)

    printf("Result: %u\n", ret);            # access to string (move $a0, <str_addr>
    return 0;
}
```

4. Floating point

   (a) (10 points) Provide an example of two IEEE 754 single precision floating point numbers and an operation on them which results in an *overflow* when performed.

   Answer: Overflow occurs when a positive exponent becomes too large to fit in the exponent field.

   Note that we cannot pick exponent to be 255 (all 1 in binary) since it represents NaN, so let's pick to floating points with exponents of 254 and a fraction of all 0. This represents a floating point number that is $1 * 2^{127}$ (here we assume that the sign is 0). If we multiply two such numbers the overflow occurs since the exponent that is 127 + 127 + BIAS = 381 does not fit in the exponent field.

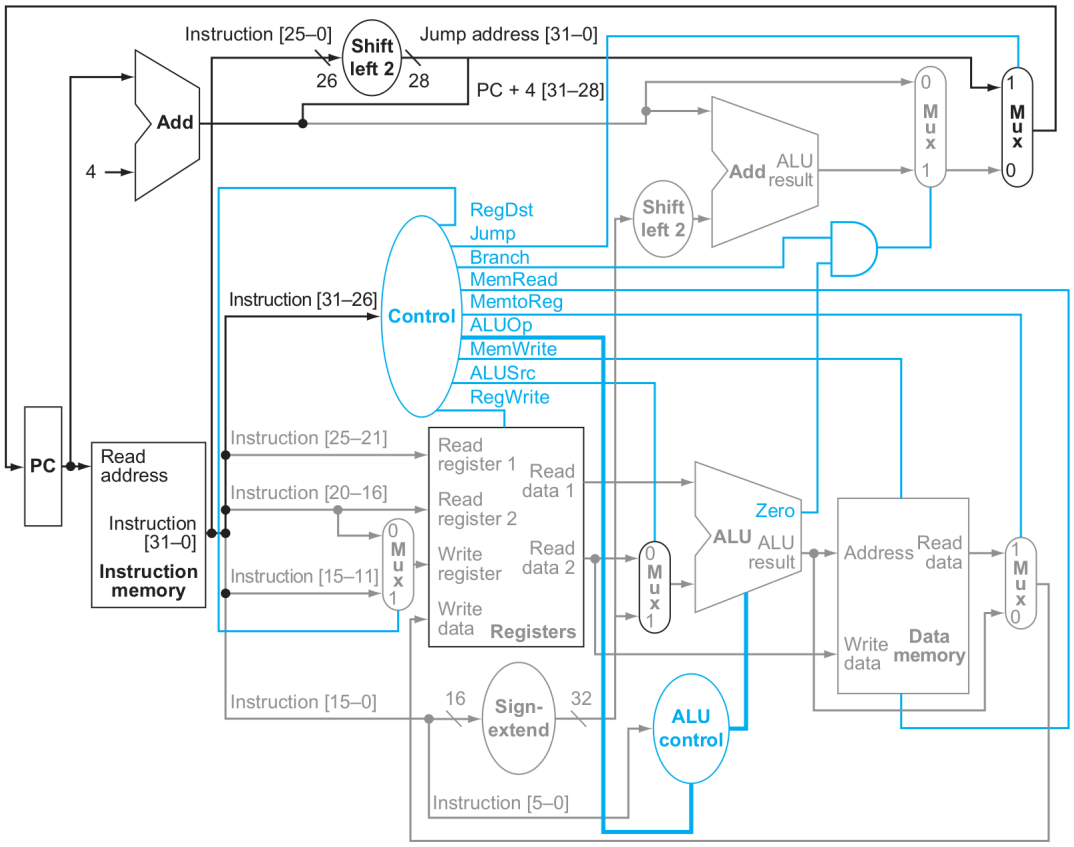
   (b) (10 points) Provide an example of two IEEE 754 single precision floating point numbers and an operation on them which results in *rounding* when performed.

   Answer: Rounding is required when the fraction of the floating point number does not fit into the 23 bits allocated to store it (for single precision).

   Again we should be careful and avoid picking a floating point number with exponent of 0 as it represents a denormalized number. We therefore pick two numbers that have 1 as exponent and all 23 bits as 1 in the fraction field. Adding these two numbers will result in a fraction that is one bit larger than the fraction field, hence ronding is required.

5. Data path

Use the figure below to explain execution of the branch on equal (`beq`) instruction in the MIPS datapath.



(a) (10 points) Explain the general flow of (`beq`) through the datapath.

Answer: The datapath reads the `beq` instruction from the instruction memory at the address contained in the `PC` register. Bits [31-26] are read by the control unit to detect that this is a `beq` instruction. Bits [25-21] and [20-16] select two registers from the register file. The values from these registers are routed to the ALU unit for comparison (note that the multiplexor controlled by the ALUSrc signal selects a register (deasserted). The control unit configures the ALU control unit to perform subtraction operation on the ALU. If register values are equal the Zero signal is asserted.

The AND gate that takes Branch and Zero selects the final value of the `PC`. In any case the `PC` is already incremented by 4. If the Zero is deaserted the branch is not taken and the multiplexor selects this `PC` + 4 value for the next instruction. However if Zero is asserted, the multiplexor selects the value computed by another ALU, which is defined as a sum of `PC` + 4 and by the bits [15-0] of the `beq` instruction that are sign extended to 32 bits and shifted by two.

(b) (5 points) Which control signals are asserted, i.e., true or 1.

Control bits:

ALUSrc - deasserted RegWrite - deasserted RegDst - don't care Jump - deasserted Branch - asserted Zero - asserted (if taken) MemRead - deasserted MemWrite - deasserted MemtoReg - don't care

ALU Control bits are set for Bnegate, CarryIn and to select the addition operation.

(c) (10 points) (and a badge of honor) The datapath has an error (with respect to how operation of the CPU was explained in Chapter 2 of the book). Find and explain the error.

Answer: If you read the book carefully you will notice the following explanation for the `j` (jump) instruction:

> **Elaboration:** Since the PC is 32 bits, 4 bits must come from somewhere else for jumps. The MIPS jump instruction replaces only the lower 28 bits of the PC, leaving the upper 4 bits of the PC unchanged. The loader and linker (Section 2.12) must be careful to avoid placing a program across an address boundary of 256 MB (64 million instructions); otherwise, a jump must be replaced by a jump register instruction preceded by other instructions to load the full 32-bit address into a register.

Specifically, *"leaving the upper 4 bits of the PC unchanged"*. Seemingly this hints that the datapath should concatenate the jump target (bits [25-0] shifted by 2) with the `PC` and not `PC` + 4. And hance a different wire in the datapath should exist.

However, if you look at the MIPS specification `https://www.cs.cornell.edu/courses/cs3410/2008fa/MIPS_Vol2.pdf` (page 115, J instruction) you notice the following explanation:

"The remaining upper bits are the corresponding bits of the address of the instruction in the delay slot (not the branch itself)"

I.e., really it's the upper bits of the next instruction or `PC` + 4. So the datapath seems to be correct with respect to the MIPS specification but incorrect with respect to the earlier explanation in the book.

Finally, note that the MIPS specification seems to have a bug in how it explainst behavior of the jump instruction

Operation:

```
I:
I+1:PC ← PC_GPRLEN..28 || instr_index || 0^2
```

Specifically, it again mentions $PC_{GPRLEN..28}$ instead of $(PC + 4)_{GPRLEN..28}$, which is obscure.