CMSC 411 - Section 101 TEST #1 26 October 1994
Computer Architecture

Please return your copy of the test along with your answers.

As an incentive for admitting you have no idea how to answer a question, you'll get 20% credit if you say you don't know the answer (or leave it blank). You can still get partial credit, of course, so please *do* answer questions you think you know even if you don't know the *whole* answer.

Please show all your work – I can't give you partial credit for a wrong answer unless I see how you arrived at the answer. Remember, "the journey is as important as the destination."

1. [10] Represent -73 and +24 in each of the following number systems: sign and (7-bit) magnitude, 8-bit 1's complement, 8-bit 2's complement, and excess-3 binary-coded decimal.

	-73	24
sign & magnitude	11001001	00011000
one's complement	10110110	00011000
two's complement	10110111	00011000
excess-3 BCD	1 1010 0110	0101 0111

2. [15] Add $15.5_{10} + 1.625_{10}$ using IEEE 754 32-bit floating point format. Convert both numbers to floating point format and show the steps necessary to perform the addition. The result should be given in IEEE format (I know you can add the two numbers and come up with 17.125_{10}).

First, we convert the two numbers into IEEE format. $15.5_{10} = 1111.1_2$, which normalizes to 1.1111×2^3. This packs into the IEEE 754 number 0|10000010|11110000000000000000000. Similarly, $1.625 = 1.101 \times 2^0$, which is already normalized. It packs into the IEEE representation 0|01111111|10100000000000000000000. To add the numbers, we must first shift them so both have the same exponent. This is usually done by leaving the larger one in normalized form (to guarantee that there'll be at most one bit of overflow to the left of the decimal point) and shifting the smaller one. In our case, we'll add together 1.1111E3 + 0.001101E3, giving 10.001001E3. We then renormalize by shifting right once and increasing the exponent, leaving the normalized result 1.0001001×2^4. This converts to the IEEE number 0|10000011|00010010000000000000000.

3. [15] Use Robertson's algorithm to multiply $-12_{10} \times -10_{10}$.The multiplication will take two 5-bit numbers (represent in 2's complement notation) as input and produce a single 10-bit number (also represented in 2's complement).

For this solution, we'll assume the multiplier is -12, and the multiplicand is -10. $-12 = 10100_2$ and $-10 = 10110_2$. In the last (correction) step, we'll subtract -10, which is the same as adding 10. Since we selected -12 for the multiplier, the M register = -10.

	A	Q
Initialize	00000	10100
Shift	00000	01010
Add	10110	00101
Shift	11011	00010
Shift	11101	10001
Correct	+01010	10001
	000111	**1000**1

The solution would be similar if the multiplier and multiplicand were switched.

4. [5] Would Booth's algorithm require more or fewer ALU operations (additions & subtractions) than Robertson's algorithm to multiply the multiplier 0101111000011110_2 × the multiplicand 0010110010110110_2? Explain your answer.

Booth's algorithm would require fewer adds & subtracts. Booth's algorithm requires one addition each time a string of zeros ends, and one subtraction each time a string of ones ends. For the multiplier in the problem, this is 3 adds and 3 subtracts. Robertson's algorithm requires one addition for each occurrence of a one, requiring 9 additions.

5. [15] Given the following microinstructions and the signals they activate, answer parts a – c.

Microinstruction	Control Signals
I_1	a b e
I_2	b d f
I_3	a b f
I_4	a c

a. Determine the set of maximal compatibility classes (MCC's).

ad, bc, ~~cd~~, ~~ce~~, ~~cf~~, ~~de~~
cde, cef
The set of MCCs is {ad, bc, cde, cef}

b. Determine all minimal MCC covers.

ad, bc, and cef are all essential (they are the only MCCs covering a, b, and f respectively). They also cover all of the signals, so they must represent the only minimal MCC cover. We could certainly add the other MCC to the set, but we *must* have those three, making them the only minimal cover.

c. Select a cover with the minimum cost such that each control signal is included exactly one. How many bits are needed to encode the minimum cost solution?

We have one redundant signal – c. We can use either of the following encodings: {ad, bc, ef} or {ad, b, cef}. The first requires {2, 2, 2} = 6 bits and the second uses {2, 1, 2} = 5 bits. The second encoding is the best one since it uses only 5 bits.

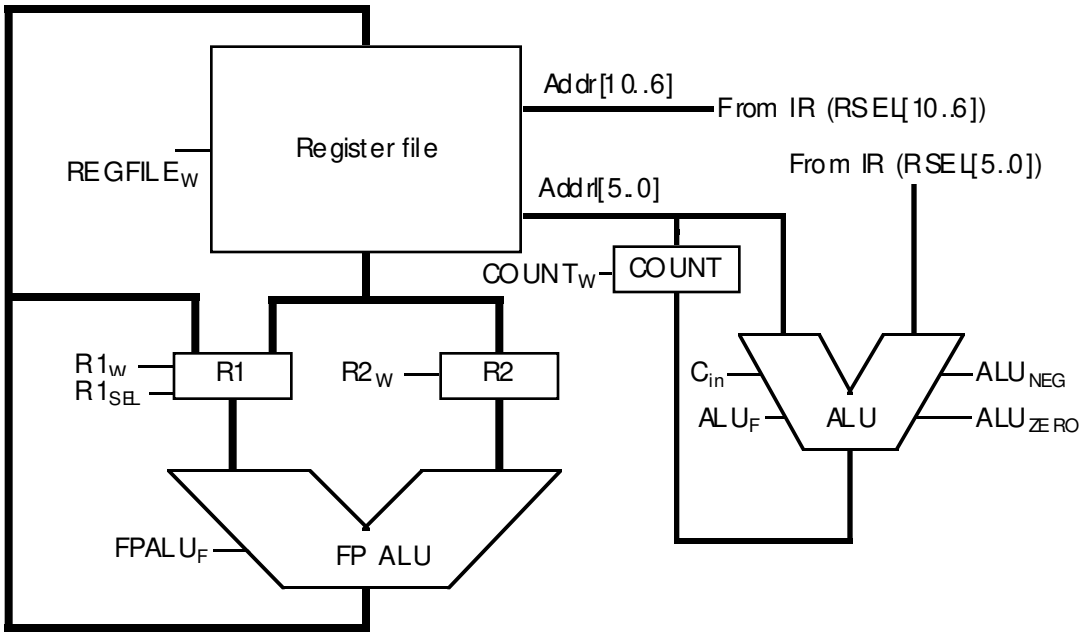
6. [5] Why would a processor designer want to use a combinational multiplier? Are there any reasons to use an iterative multiplier instead?

A combinational multiplier is faster and easier to design than an iterative multiplier, since it has a regular arrangement of parts and does not need "looping." However, a combinational multiplier for long numbers (32 bits, for example), requires a tremendous amount of hardware. If speed is not critical, or if implementation size *is* a factor, an iterative multiplier will do better.

7. [10] Give two advantages and one disadvantage for each of RISC and CISC.

There are lots of possibilities for this question. RISC's advantages are fixed-length instructions, shorter cycle time due to lower complexity, simpler design, and greater number of registers. RISC's disadvantages are longer code length and the large semantic gap between RISC assembly language and high-level languages.
CISC's advantages are smaller programs, a smaller semantic gap, and a more flexible design often due to the use of microcode. CISC's disadvantages are generally slower cycle time and difficulty of design.

8. [25] You're given a monophase microcontrolled floating-point vector arithmetic unit which includes 32 registers that can each hold 64 32-bit IEEE 754 floating point numbers. A typical instruction might be V1 + V2 -> V3. This adds each element of V1 to the corresponding element in V2, storing the result in the corresponding element of V3. So, (1 2 3 4) + (5 6 7 8) -> (6 8 10 12). This is just an example – the vector registers have 64 elements each . The datapath for the microcontroller is shown below; assume that the microcontroller itself uses the AMD 2909 in the standard design we went over in class.



The floating point ALU has three input signals, which work according to the following table. There are no other control signals either from or to the FP ALU.

Value	Output
000	R1
001	R2
100	R1 + R2
101	R1 - R2
110	R1 × R2
111	R1 / R2

The integer ALU attached to the COUNT register supports the same 6 functions, with COUNT corresponding to R1 and the bits from the microinstruction register corresponding to R2. In addition, the C_{in} bit comes from the μIR. The ALU asserts ALU_{NEG} if the result is negative, and ALU_{NEG} if the result is 0.

Since this is the cheap version of the vector unit, the register file can only read or write one element of one vector at any time. The vector number is selected by the μIR, and the element within the vector is selected by the output from the integer ALU. The register file uses the values at the start of the cycle as an index; thus, it is not affected by any changes to the output of the ALU during the cycle.

Each register has a single control line – write enable. If this line is 1, the register will be written. No read enable is necessary, as each register always puts its current value onto its outputs. The same is true for the register file. The control lines for the registers are $R1_W$, $R2_W$, $REGFILE_W$, and $COUNT_W$. In addition, R1 has a select line ($R1_{SEL}$). If this line is 0, R1 will be loaded from the register file. If it is 1, R1 will be loaded with the output of the FP ALU. Clearly, the signal is a "don't care" if R1 is being read.

In addition, the microinstruction format has space for a condition selector and a branch address. Any operations specified by the instruction occur regardless of whether the branch is taken. The condition codes are those at the *start* of the cycle, so the branch is based on the conditions generated by the *previous* integer ALU operation.

For each instruction, please give the instruction's meaning in plain English, and also give the values for the control signals in this format:

$R1_W$,$R1_{SEL}$,$R2_W$,$REGFILE_W$,$COUNT_W$,$FPALU_F$,ALU_F,C_{in},RSEL[10:6],RSEL[5:0]

A sample instruction might look like this:

REGFILE[00110,count] <- R1+R2; count <- count+1; IF (ALU_{ZERO} = 1) GOTO test2;
encoding: 0,d,0,1,1,100,100,0,01100,000001

If count=8 at the start of the cycle, this will move the 9th element of V6 to R1, and increment count to 9. Note that both R1 and COUNT are loaded with values that are set up by other signals and computed during the cycle. However, REGFILE is indexed by the value of COUNT at the *start* of the cycle; the index is unaffected by any signals set by the current microinstruction. If ALU_{ZERO} is 1 at the *start* of the instruction, the branch will be taken. Otherwise, the next microinstruction in sequence will be executed.

a. What microcode sequence is necessary to copy the tenth element of V8 to the tenth element of V2 (recall that the tenth element has an offset of 9)?

```
count <- 001001                                    0,d,0,0,1,ddd,001,d,ddddd,001001
R1 <- REGFILE[01000,count]                         1,0,0,0,0,ddd,ddd,d,01000,dddddd
REGFILE[00010,count] <- R1                         0,d,0,1,0,000,ddd,d,00010,dddddd
```

b. What microcode sequence would you use to multiply V4 × V13, putting the result into V13?

```
count <- 000000                                    0,d,0,0,1,ddd,001,d,ddddd,000000
LOOP:
R1 <- REGFILE[00100,count]                         1,0,0,0,0,ddd,ddd,d,00100,dddddd
R2 <- REGFILE[01101,count]                         0,d,1,0,0,ddd,ddd,d,01101,dddddd
REGFILE[01101,count] <- R1*R2; count <- count - 1   0,d,0,1,1,110,101,1,01101,000001
IF (ALUzero) GOTO END                              not applicable
GOTO LOOP                                          not applicable
END:
```

For parts c & d, there isn't a single right answer for either question. Pick an improvement for each, and explain the tradeoff.

c. What is one *specific* change you could make to the hardware to make it run faster? How would this affect the number of bits needed for the microcode store?

There are several changes that will do this. The most obvious is adding a second read port and/or a separate write port to the register file. This would require more microcode bits to specify more register locations. You could also allow the indexing of the register file directly from the microcode word. This would add an additional bit to select where that half of the index comes from. Adding more condition lines (in particular, the inverses of ALUneg and ALUzero) might speed things up at the expense of requiring more condition select bits. These changes would all increase the width of the microstore, though it's not clear whether they'd require more bits or not (the height of the store might decrease).

d. What is one *specific* change you could make to the microinstruction format to use fewer bits of storage (using the original hardware)? How would this affect the overall speed of the vector unit?

One easy change would be to use two different microinstruction formats – one for branches and one for data instructions. You could also multiplex the two register select fields, allowing the specification of either field (but not both) in the same instruction. Both of these changes would slow down the vector unit by requiring more instructions.