EECS 370 Final Exam

Fall 2019

Notes:

- Closed book. 1 8.5x11 inch sheet of paper as notes.
- **10 problems on 14 pages**. Count them to be sure you have them all.
- Calculators without wireless are allowed, but no PDAs, Portables, Cell phones, etc.
- This exam is fairly long: *don't spend too much time on any one problem*.
- You have **120 minutes** for the exam.
- Some questions may be more difficult than others. You may want to skip around.
- **Partial credit cannot be given if work is not shown.**
- **Write legibly and dark enough for the scanners to read your work.**

Write your uniqname on the line provided at the top of each page.

You are to abide by the University of Michigan/Engineering honor code. Please sign below to signify that you have kept the honor code pledge:

I have neither given nor received aid on this exam, nor have I concealed any violations of the Honor Code.

Signature: ____

Name: ____

Uniqname: ____

First and last name of person sitting to your **right** (write the symbol ⊥ if at the end of a row):

____

First and last name of person sitting to your **left** (write the symbol ⊥ if at the end of a row):

____

Problem 1: Multiple choice. Points: 9

Clearly write the correct **capital** letter in the blank shown.

1) Consider the following table of events that might occur during a memory access:

	Cache	**TLB**	**Page fault**
1.	miss	hit	yes
2.	hit	miss	no
3.	miss	miss	yes
4.	miss	hit	no
5.	hit	hit	no

Which of the following is the most likely correct about the frequency of the above events occurring? Assume the cache is physically addressed. **[3]**

____

	Least Frequent				**Most Frequent**
A)	1	2	3	5	4
B)	1	3	2	4	5
C)	3	1	2	5	4
D)	3	4	2	1	5
E)	4	5	3	2	1

2) Under what conditions would you expect to have a smaller number of bytes transferred between the cache and memory with a write-through cache (as opposed to a write-back cache)? **[3]**

____

A. The program has no spatial locality but high temporal locality
B. The program has high spatial locality but no temporal locality
C. The program has high spatial locality and high temporal locality
D. The program has no spatial locality and no temporal locality
E. Never

3) Consider the use of a virtually addressed (VA) cache as opposed to a physically addressed (PA) cache. Which of the following is true: **[3]**

i) It takes less time on average to process a hit on a VA cache than a PA cache because the VA cache doesn't need to wait for the TLB access.
ii) In a VA cache the set index will need to be larger (more bits) than a PA cache to account for the page number generally being larger (more bits).
iii) A VA cache has less overhead associated with a context switch than a PA cache.

____

A. Only i
B. Only iii
C. Only i and ii
D. Only ii and iii
E. All of i, ii, and iii

Problem 2: Short answer Points: 25

1) Provide a 32-bit binary string which represents the number -4.125 as an IEEE float. Include all 32 bits. **[6]**

1100 0000 1000 0100 0000 0000 0000 0000

2) Consider the following access pattern: A, B, B, C, A. Assume that A, B, and C are memory addresses each of which are in a different block of memory. Further, assume A, B, and C are generated in a uniformly random way (each block is equally likely) and that the LRU replacement algorithm is used. Compute the probability that the second instance of "A" will be a hit for the following caches. Briefly show your work.

   a) The cache has 8 lines and is direct-mapped. **[4]**

   Since it is direct mapped, the second instance of “A” will only hit if “B” and “C” are not in the same set as “A”

   The chance of that happening is ⅞ * ⅞ = 49/64

   b) The cache has 8 lines and is 2-way set associative. **[4]**

   Since it is 2-way set associative, the second instance of “A” will only hit if “B” and “C” are not both in the same set as “A”

   The chance of that happening is 1 - 1/4 *1/4 = 15/16

Short answer continued

3) Say you have an LC2K program running on your project 3 pipeline with the following characteristics:

- 30% of instructions are loads
- 10% are stores
- 20% are adds
- 15% are nors
- 25% are branches
- 30% of the instructions are dependent on the instruction immediately before them in program order. You may assume that is true no matter what instructions are involved.
- 55% of branches are taken.

What would be the expected CPI of your program using detect-and-forward to resolve data dependencies and detect-and-stall for branches? Clearly show your work. **[6]**

1 + (0.3 * 0.3 * 1) + (0.25 * 3) = 1.84

4) Say you have an ISA where all instructions are 32-bits and which has 16 general-purpose registers and all immediate values are 16 bits. If your instruction set consisted of nothing other than instructions that used two general-purpose registers and one immediate, how many opcodes could your ISA support? Clearly show your work. **[5]**

Each instruction is 32 bits.
16 bits are reserved for the immediate
4 bits reserved for each of the 2 registers

This leaves 8 bits for the opcode

This means there are 2^8 possible opcodes

Problem 3: True/False Points: 12

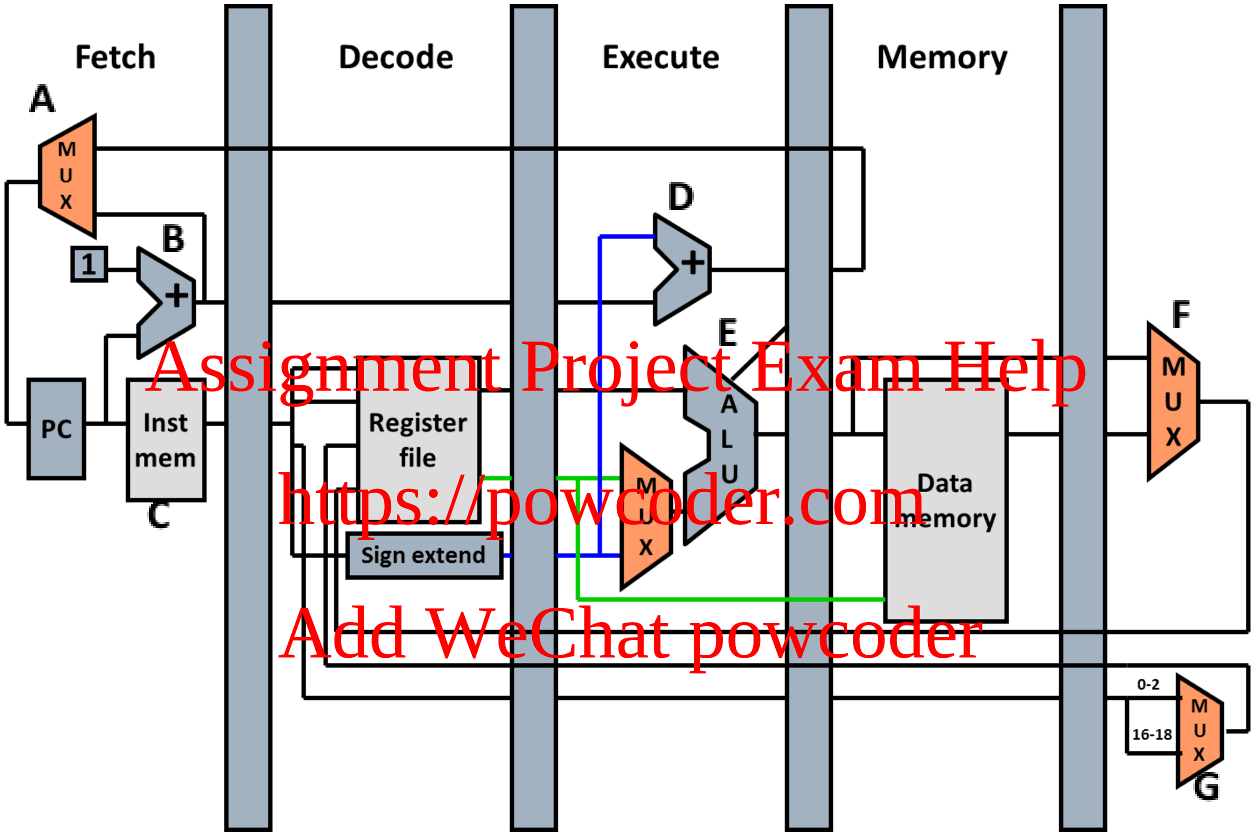
Circle the correct answer. -2 per wrong or blank answer, minimum zero.

In the five stage pipeline, not having internal forwarding in the register file (as we did in project 3) means we need to squash 3 instructions on a mispredicted branch as opposed to 2 in lecture.	True [✓ False]
Pipelining generally increases the number of instructions completed per unit time when compared to an otherwise equivalent multi-cycle implementation.	[✓ True] False
A clock with a 20ns period has a frequency of 50MHz.	[✓ True] False
Data hazards can generally be removed using simple branch prediction	True [✓ False]
An XOR gate can be created using only AND gates	True [✓ False]
Flash memory is non-volatile	[✓ True] False
The “direction predictor” of a branch predictor predicts if a branch is taken or not taken.	[✓ True] False
In the 3C’s cache model, a “compulsory” cache miss can’t be avoided by changing any of the block size, cache size, or associativity.	True [✓ False]
A 1-bit branch predictor will get about a 0% hit rate on a branch that alternates between taken and not-taken forever (so T, N, T, N, T, N…)	[✓ True] False
A TLB is used to predict the address of a branch	True [✓ False]
Dennard scaling is the claim that each transistor will generally use the same amount of power no matter how small the transistor is.	True [✓ False]
A TLB can be reasonably thought of as being a cache of the page table.	[✓ True] False

Problem 4: Pipeline defects Points: 12

The EECS 370 staff has fabricated the Project 3 pipeline onto a real silicon chip! However, no manufacturing process is perfect, so students have been asked to verify their processor by running some LC-2K instructions and observing the behavior of the pipeline.

Below is the pipeline diagram for your reference. The letters indicate hardware components that you are trying to verify. For each of the scenarios below, state which component(s) could be working incorrectly.



a. You notice that for only **add** and **nor** instructions, the answer is saved into **regB** rather than **destReg**. What component(s) could be working incorrectly? **[3]**
G

b. You notice that every pipeline register has been filled with the first instruction and that no other instruction ever gets executed. What component(s) could be working incorrectly? **[4]**
B, C

c. You notice that the pipeline will always take the branch when executing a **beq** instruction, even if the registers are not equal. What component(s) could be working incorrectly? **[5]**
A, E

Problem 5: More page tables Points: 16

Consider a byte-addressable system with 28-bit virtual addresses and a 3-level page table. Assume pages are 256 bytes in size and each page table entry is 4 bytes. Each of the 2nd-level page tables fits across exactly **two** pages, while each of the 3rd-level page tables fits into a **single** page. There is no such restriction on the 1st-level page table.

a) How many address bits are used for each field? **[4]**

1st-Level Table Index	2nd-Level Page Table Index	3rd-Level Page Table Index	Page Offset
27 - 21 (7 bits)	20 - 14 (7 bits)	13 - 8 (6 bits)	7 - 0 (8 bits)

b) Consider the following code

```
typedef struct Mosaic{
    int dimensions[14]; //56 bytes
    int color;          //4 bytes
    short opacity;      //2 bytes
}Mosaic;

Mosaic gallery[4096];

for(int i = 0; i < 4096; ++i) {
    gallery[i].opacity = 50;
    gallery[i].color = BLACK;
}
```

At the beginning of the program, the entire 1st level page table is in memory, but the 2nd and 3rd level page tables are not, and neither are any data pages. The array "gallery" starts at address 0x80000. You may assume the physical memory is large enough to hold the page tables and the data.

After the program finishes execution, how many ***physical pages*** will be used to hold each of the following? Briefly show your work.

i) 2nd-level page tables? **[4]**

# 3rd level pages that fit into a 2nd-level page table is 128 (64 per page) . Therefore, we need 64 / 16 = 1 page. → 2 pages, since this is the size of the 2nd level page table

ii) 3rd-level page tables? **[4]**

# Data pages that fit into a 3rd-level page table is 64. Therefore, we need 1024 / 64 = 16 pages.

iii) The `gallery` array? **[4]**

# Mosaics that fit into a page is 4. Therefore, we need 4096 / 4 = 1024 pages.

Problem 6: Hazardous LC2K Points: 24

Your friend implemented a function in LC2K. Their pipeline uses **detect-and-forward** to handle data hazards and **speculate and squash** (predict always not taken) to handle control hazards.

```
0  | func  lw    0  6  plsOn  // r6 = +1
1  |       lw    0  1  size   // r1 = size
2  |       lw    0  7  mnsOn  // r7 = -1
3  |       add   1  7  3      // r3 = size - 1 (right)
4  |       add   0  0  4      // r4 = 0 (left)
5  | lp    beq   3  4  endLp  // if (left == right) end loop
6  |       lw    3  1  testA  // r1 = mem[start + right]
7  |       lw    4  2  testA  // r2 = mem[start + left]
8  |       beq   1  2  contn  // if (left == right) continue
9  |       beq   0  0  retZr
10 | contn add   4  6  4      // left++
11 |       add   3  7  3      // right--
12 |       beq   0  0  lp     // branch back to loop
13 | endLp add   0  6  5      // r5 = 1 (return True)
14 |       beq   0  0  end    // branch to end
15 | retZr add   0  0  5      // r5 = 0 (return False)
16 | end   halt               // end
17 | mnsOn .fill -1
18 | plsOn .fill  1
19 | size  .fill  5
20 | testA .fill  1           // test array = [1, 2, 3, 2, 1]
21 |       .fill  2
22 |       .fill  3
23 |       .fill  2
24 |       .fill  1
25 |
```

a) How many stalls are added due to data-hazards throughout the lifetime of the program? Briefly justify your answer. **[4]**

3 noops (1 after line 2 and 1 after line 7. Line 7 executes twice)

b) How many stalls are added due to control-hazards throughout the lifetime of the program? Briefly justify your answer. **[4]**

18 noops (6 branches are taken - line 8 (twice), line 12 (twice), line 5 (once), line 14 (once))

c) Write two lines we could swap with each other to reduce the total number of stalls that would occur due to data hazards. **[4]**

Swap ___0____ and ___2___ (or 3 and 4)

Hazardous LC2K (Continued)

You wonder what this function does. Your friend left some cryptic comments but no documentation for its specific purpose. You know that the function runs on a test array (testA) and the return value is found in r5. You ask yourself questions in order to figure out the functionality of the program.

d) What is the value of r5 when the program halts? **[3]**

______1______

e) What would the value of r5 be when the program halts if testA was instead `[1,5,3,2,1]`? **[3]**

______0______

f) In 10 words or less, describe what the return value of the function is in terms of the array. Your answer should be something like "returns the maximum value in the array". **[3]**

Returns whether the array is a palindrome, is the same forwards and backwards, etc.

g) This program does some unusual—and likely not intended—things if the array size is even. In 15 words or less, describe what happens. **[3]**

The function will always return 0. If the array is an even sized palindrome,

the function will eventually compare uninitialized memory past the array to the data/instructions before the array and return 0 if they find they are unequal (which is very likely).

It will only segfault if the uninitialized values after the array mirror the values before it exactly until you go out of the memory range and will not loop forever.

If it is not a palindrome, it will return 0 like expected.

Problem 7: Stack 'em up Points: 18

Consider the 8-bit stack-based ISA named "HOB". There are 3 instruction formats (bit 0 is the least-significant bit). Memory is byte-addressed.

R-type instructions (add, nand):
bits 7-5: opcode
bits 4-0: unused (should all be 0)

I-type instructions (pushi):
bits 7-5: opcode
bits 4-0: value (a 5-bit, 2's complement number with a range of –16 to 15)

A-type instructions (push, pop, beq)
bits 7-5: opcode
bits 4-0: address (in the range of 0 to 31)

Assembly language Instruction	**Opcode in binary (bits 7, 6, 5)**	**Action**
add (R-type)	000	Pop the top two values off of the stack and push the sum of those two 2's complement numbers back on the stack.
nand (R-type)	001	Pop the top two values off of the stack and push the bit-wise NAND of those two values back on the stack.
push (A-type)	010	Take the byte stored at the memory location specified by the address field and push it onto the stack
pop (A-type)	011	Take the value at the top of the stack and pop it (remove it from the stack). Place that value into memory location specified by the address field.
pushi (I-type)	100	Push the sign-extended 8-bit value stored in the value field onto the stack.
beq (A-type)	101	If the two values on the top of the stack are equal, pop them and branch to the location specified by the address field. Otherwise do nothing (don't change anything on the stack.)
halt (R-type)	110	Increment the PC (as with all instructions), then halt the machine (let the simulator notice that the machine halted).

Stack 'em up continued

a) Translate the following into *hexadecimal*. **[4]**

i. push 10 0x___4A______

ii. beq 3 0x___A3______

b) Write a program in HOB assembly which computes 6+8 and puts the result in memory location 7. The stack should be empty after your program runs. Your program may be no more than 8 lines long **[4]**

pushi 6 (or 8)
pushi 8 (or 6)
add
pop 7
halt

c) Assume memory and the stack are initialized as shown. Show the final values in the memory and stack assuming the program is executed as HOB machine code starting at PC=0. All values not shown may be assumed to be zero. Show your work. **[10]**

Initial values:

Memory	
0	0x00
1	0xA3
2	0xC0
3	0x64
4	0xC0
5	0xC0
6	0xC0

	Stack
	0x02
	0x04
	0x06
	0x00
	0xFF
	0x03
Bottom of Stack	0x80

Final values:

Memory		
0	0x00	add
1	0xA3	beq 3
2	0xC0	halt
3	0x64	pop 4
4	0x0	add
5	0xC0	halt
6	0xC0	halt

	Stack
	0x02
Bottom of Stack	0x80

Problem 8: Caches Points: 11

Answer the following questions about a system with a 16-byte fully-associative cache using 32-bit addresses.

a) Fill out this table for caches with different sized blocks. **[6]**

	4-byte blocks	8-byte blocks
Number of bits in block offset	2	3
Number of bits in tag	30	29
Number of bits for LRU in each cache line	2	1

b) Simulate the following sequence of reads for each of the previous cache architectures from part a and determine whether each is a hit or miss. Assume an initially empty cache that is using LRU replacement. -1 per wrong or blank answer, minimum 0. **[5]**

	4-byte blocks	8-byte blocks
0x1008	Miss	Miss
0x1000	Miss	Miss
0x2004	Miss	Miss
0x2000	Miss	Hit
0x1008	Hit	Miss
0x2004	Hit	Hit

Problem 9: CMOV Points: 12

Your boss has claimed that adding a "CMOV" instruction to the LC2K ISA will improve performance on certain programs. In particular she says that the following program can be improved with a CMOV instruction:

```
          beq 0 3 tom  //Assume reg0 is always 0

          add 1 1 1

tom  add 2 2 2
```

The CMOV instruction she would like to consider uses the following encoding:

```
bits 24-22: opcode (replace noop)
bits 21-19: reg A
bits 18-16: reg B
bits 15-3:  unused (should all be 0)
bits 2-0:   dest reg
```

Where a CMOV will set the dest reg equal to reg B if and only if reg A is not equal to 0. Otherwise the instruction just acts as a noop.

a) Rewrite the code above so that it does the same thing but uses a CMOV rather than a beq using no more than 4 instructions. You may freely use register 7 as a scratch register. **[7]**

```
add   1 1 7
add   2 2 2
cmov 3 7 1
```

```
add 0 0 7
cmov 3 1 7
add 1 7 1
add 2 2 2
```

b) Using 25 words or less, explain under what circumstances the code in part a might run better on a pipelined machine than the original code. *You'll get one point for writing "I don't know" and nothing else or leaving this blank.* **[5]**

If we were to resolve our branches with speculate-and-squash and we mispredicted the beq in the original code (or if we resolved control hazards with detect and stall), executing the original code would require 10/11 cycles due to the added noops. In the code from Part A with the cmov, we would only need 7 cycles.

Problem 10: Looping round two Points: 14

A program is generating accesses on a processor with a cache. The cache has a block size of 8 bytes. The only data structure in memory for program is an array with “size” elements (other data is in registers). Each array element is 1 byte. There are two input parameters (**stride** and **size**) to your program. Your program reads the array according to the parameter stride in the following pattern (yes, this is an infinite loop).

```
byteAddress = 0;
while (1) {
                read array[byteAddress % size];
                byteAddress += stride;
}
```

Ignore the accesses during the first time through the array (i.e. give the miss rate after the program has been running for a very long time and has gone through the entire array many times).

a) Assume the CPU cache holds 256 bytes of data and is **direct-mapped** Fill in each entry of the following table with the CPU cache’s **miss rate** for a given combination of **stride** and **size**.
[7 points, -1 per wrong/blank answer, minimum 0]

	size		
stride	**256**	**384** (256+128)	**512**
4	0	1/3	1/2
8	0	2/3	1
16	0	2/3	1

b) Now assume the CPU cache holds 256 bytes and is **fully associative and LRU**. Fill in each entry of the following table with the **miss rate** for a given combination of **stride** and **size** .
[7 points, -1 per wrong/blank answer, minimum 0]

	size		
stride	**256**	**384** (256+128)	**512**
4	0	1/2	1/2
8	0	1	1
16	0	0	0