CS 220 **Final Exam** **Spring 21** **Name:**________________________________

Show all work to get credit and partial credit. Some answers are better (full credit) than other answers (partial credit). Type up your answers in your favorite word processing tool (Word, LibreOffice, Latex) but make sure to convert to a PDF file before submitting. It is probably easiest to just use this LibreOffice document. Create a directory in your course github repository named **`FinalExam`** (spelled exactly like that) and put your PDF file in there named ***your_username*.pdf** (for example **ehar.pdf**) Unless otherwise stated in a question put all code inline in your exam document. Make sure to restate the question and that your answers are clearly numbered. Some non-code questions it might be easier for you to write up a solution on paper. Feel free to take a picture of the paper and insert the image.

By submitting this exam you are affirming that you have received no outside help on this exam. You may of course use your notes, class videos, and code, man pages, and other resources mentioned in the questions.

1. [4] Consider the 32-bit integer **`0x12345678`**. On a *little-endian* architecture byte 0 in the integer is

   __________ and byte 3 is ___________ (express answers in hex). Hint: We never covered the terms big and little endian in class, but it is in your ARM Assembly textbook reading. Explain your answer by explaining big-endian and little-endian.

   Question was poorly worded so didn't grade.

2. [4] Declare a variable **`ptr`** as a pointer to a long. Initialize **`ptr`** so it points to *dynamic memory* and initialize the value in memory to 6.

```
long *ptr = malloc(sizeof(long));
*ptr = 6;
```

3. [4] Declare a variable **`ptr`** as a pointer to a long. Initialize **`ptr`** so it points to *stack memory* and initialize the value in memory to 6.

```
long x = 6;        // need local stack memory
long *ptr = &x;
```

4. [4] Express **3.14159** in IEEE 754 single-precision floating-point format in hexadecimal. Show all work on how you derived the number.

   40490FD0 for full credit needed to explain rounding bits

5. [4] In a Virtual Memory system the **TLB** is a small fully-associative **cache**

   for the (**MMU or page table**).

6. Assume we have a 32KiB direct mapped instruction cache with 32-bit addresses, $\mathbf{b_{31}b_{32}...b_3b_2b_1b_0}$

and 32-bit words (instructions). Assume there are 2048 blocks (rows, lines) and the cache is word addressable (not byte addressable). The 32KiB is for instructions and does not include extra memory for tag and valid bits.

Each block is (215 KiB) / (2**11 blocks) = 2**4 bytes, lower two bits are zero so we have two bits for the offset.**

b_{31}...$b_{15}$$b_{14}$...$b_4$$b_3$$b_2$00 TAG INDEX OFFSET

a. [2] True/False. In this cache design bits **b_1** and **b_0** are always zero. ____________. Explain.

True because instructions are 32 bits and on 4 byte boundary.

b. [2] Which bits in the address contain the block index? ____________________

In **red** above

c. [2] How many instructions are stored in each block? ____________________

Offset is two bits, so four instructions per block.

d. [2] Which bits in the address contain the tag? _______________________

In **yellow** above

e. [2] Which bits in the address contain the block offset? _______________________

In **green** above

7. [5] Assuming we are representing a float in IEEE 32-bit format what decimal number does **0xBF400000** represent? (Show all work)

-.75 **Just about everyone got this right**

8. [10] In class we wrote a program **`dec2bin.c`** (decimal to binary) that printed the IEEE 754 floating-point hexadecimal representation of a floating-point number. You should have used this program to check your previous answer for converting **3.14159** to hex. For this question write a program **`bin2dec.c`** that does the opposite. It reads a hexadecimal number as a command line argument and prints it as a floating-point number. This program is very short, much like **`dec2bin.c`**. However it is slightly more tricky than **`dec2bin.c`**. Here is a hint, look at the man pages for the functions **`strtol`** and **`strtoll`**. Also, as I implemented this I also had to be reminded that https://stackoverflow.com/questions/28097564/why-does-printf-promote-a-float-to-a-double . Here are a couple of example executions of bin2dec.

```
./bin2dec 402df851
2.718281

./bin2dec 3f800000
1.000000
```

Turn in a file named **`bin2dec.c`** in your **`FinalExam`** directory. Add comments judiciously to clearly explain how the program works.

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {

     // read in 32 bits. Use long long so we don’t
     // accidentally lose bits
     long long x = strtoll(argv[1], NULL, 16);

     // convert to a 32 bit float
     float  *f = (float *) &x;

     // print as a double because that’s all printf will do
     double d = *f;
     printf("%f\n", d);
}
```

9. [10] Other than making your programs run faster, the effect of caches are largely hidden from programmers. Before this class you may not have even been aware of what a CPU cache is. Another reason why it is important that, as a programmer, to understand some basic computer hardware such as caches is to help understand some cybersecurity related attacks. Spectre and Meltdown are software attacks that exploit caches by tricking the CPU into executing instructions and loading private data into a cache and then figuring out what that private data is (for example, a password or credit card number). Watch the two brief videos below (maybe a few times) and write a short half page summary of the Spectre and Meltdown vulnerabilities. Explain the difference between the two attacks and highlight terminology that you learned this semester. I have watched dozens of videos on Spectre and Meltdown and these two videos are good short descriptions of the attacks. The speaker is a non-native English speaker so please excuse any minor grammar issues.

a. **https://www.youtube.com/watch?v=JSqDqNysycQ**
b. **https://www.youtube.com/watch?v=q3-xCvzBjGs**

I liked Ganesh’s answer.

The Spectre and Meltdown vulnerabilites are essentially ways of exploiting CPU cache functionality and CPU optimizations in order to access otherwise private data. Despite being very similar in approach, there are a few key differences that make Meltdown much more dangerous than Spectre, as it can access protected memory of programs other than the one running the exploit, while Spectre can only access the same program's memory.

The way Spectre works is that an array is declared, and then an "out-of-bounds" element is accessed inside an if-statement which checks the bounds. Since the index being accessed is larger than the size of the array, we expect the code inside the if statement to not run. However, the concept of "speculative execution" means that the CPU runs the code inside the *if* statement anyways, and undoes said execution once the if statement is found to be false. However, this undo does not remove the stored data from the CPU cache, meaning there is a "private character" stored in the CPU cache. Spectre then tries to access all possible characters, and times how long each access takes. If one access takes a significantly shorter length of time than the other accesses, we know that the character we accessed was the same as the character that was in the "secret" byte. Thus, we have successfully accessed one byte of private data. Doing this for many different indices allows for a significant amount of private data to be accessed, which naturally is a large security vulnerability.

Meltdown operates on a similar concept. However, instead of accessing an out of bound array index, it instead reads a character from a private address. Naturally, this causes a segmentation fault, as the program is trying to access private memory. However, the concept of out-of-order execution means that some transient instructions, located after the address access, are still executed. Additionally, the character accessed is still stored in the CPU cache before the segmentation fault is thrown. This means that a similar time-based attack to that used in Spectre can be used in order to deduce what the character that was accessed actually is, meaning that repeating this process allows for reading entire blocks of private data. Additionally, since we are accessing a private address and not just going out of bounds on an array, we can access data from anywhere in memory, not just limited to the currently running program.

10. We wrote an iterative version for computing $\mathbf{x^y}$ as a C program (**fasterxtoy.c**) below.

```
int xtoy(int x, int y) {
    int currsqr = x, rv = 1;

    while (y > 0) {
        if (y & 1)
            rv *= currsqr;
        currsqr *= currsqr;
        y >>= 1;
    }
    return rv;
}
```

Here is the object dump of the **xtoy** function after I translated it to assembly and has been linked against a main program.

```
104a8:      e1a02000       mov     r2, r0
104ac:      e3a03001       mov     r3, #1
104b0:      e52d4004       push    {r4}     ; (str r4, [sp, #-4]!)
104b4:      e3510000       cmp     r1, #0
```

```
104b8:      da000006      ble     104d8 <endwhile>
104bc:      e2014001      and     r4, r1, #1
104c0:      e3540000      cmp     r4, #0
104c4:      0a000000      beq     104cc <endif>
104c8:      e0030293      mul     r3, r3, r2
104cc:      e0020292      mul     r2, r2, r2
104d0:      e1a010a1      lsr     r1, r1, #1
104d4:      eafffff6      b       104b4 <while>
104d8:      e1a00003      mov     r0, r3
104dc:      e49d4004      pop     {r4}     ; (ldr r4, [sp], #4)
104e0:      e12fff1e      bx      lr
```

a. [2] What instruction addresses contain the loop?

104b4 – 104d4

b. [4] When computing **`xtoy(2,5)`** how many times is the **`cmp`** instruction in the loop condition executed?

There are two compare instructions. Only the one at address 104d4 is the loop condition. The variables y is a 3 bit 5 (101) so we check each bit, and one final check when it reaches 0. So four times.

c. [4] When computing **`xtoy(2,n)`** how many times is the **`cmp`** instruction in the loop condition executed? Write an expression in terms of **n**.

Using python notation because equations in Word stink

`math.ceil(math.log2(n)) + 1`

d. [2] The **`push`** instruction is really shorthand for the instruction

```
str r4, [sp, #-4]!
```

Explain what the syntax [sp, #-4]! means. Hint: see ARM Programmer's guide.

Subtract four from the stack pointer before storing r4, and also update the stack pointer to the new value.

e. [5] Assume we have a direct mapped cache with four blocks that can hold only one instruction per block. If we were computing **`xtoy(2,5)`** what would the hit ratio of the cache be? Explain why the performance is so good or bad or mediocre.

Since the loop body cannot fit into the cache every instruction access is a miss, hence the hit ration is 0%.

f. [10] Assume we have a direct mapped cache with <u>four blocks and two instructions per block</u>. If we were computing **`xtoy(2, 5)`** what would the hit ratio of the cache be? Explain answer by determining if each instruction access is a hit or miss and list the block index for each address and the word offset.

Notes	Hex Address	binary	Block index	Offset	Loop iter 1 y = 101	Loop iter 2 y = 10	Loop iter 3 y = 1	Loop quit	
	A8	10101000	1	0	M				
	AC	10101100	1	1	H				
	B0	10110000	2	0	M				
beginwhile	B4	10110100	2	1	H	M	M	M	
	B8	10111000	3	0	M	H	H		
	BC	10111100	3	1	H	H	H		
	C0	11000000	0	0	M	H	H		
	C4	11000100	0	1	H	H	H		
If body	C8	11001000	1	0	M	XXXXXXX	H		
	CC	11001100	1	1	H	H	H		
	D0	11010000	2	0	M	M	M		
b	D4	11010100	2	1	H	H	H		
Endwhile	D8	11011000	3	0				M	
	DC	11011100	3	1				H	
	D0	11010000	0	0				M	

# hits = 20 and the total number of memory accesses is 33 so 20/33 = 60.6 %.

It is very easy to make a small mistake here. Anywhere in the 55% to 65% range would work.

g. [5] Assume we were computing **`xtoy(2,5)`** on a real ARM processor with 256 Sets, four blocks per set, and 8 instructions per block. What would the cache hit ratio be?

This is a big cache. With 8 instructions per block we would only have about 1 miss out of 8 consecutive access and no cache conflicts. Since the whole program is only 15 instructions two memory access would fetch the whole function into cache. SO the cache hit rate would be 31/33 = 94%

11. [5] Write a C function **clear** that takes two parameters **vec** and **n. vec** is an array of integers and **n** is the number of integers in the array. The function **clear** should initialize the array to all zeros. Do not call the C function **memset** or **memcpy**.

```
extern void clear(int vec[], int n);
void clear(int vec[], int n) {
    int i = 0;
    while(i < n) {
        vec[i] = 0;
        i++;
    }
}
```

12. [10] Write the function **clear** from the previous problem as an ARM assembly language function. Put the code in a file named **clear.s**.

```
clear:
        mov r2, #0
        mov r3, #0
while:
        cmp r2, r1
        bge endwhile
        str r3, [r0]
        add r0, r0, #4
        add r2, r2, #1
        b while
endwhile:
        bx lr
```

13. [5] Explain why or why not the instruction addresses in function **clear** exhibit temporal locality.

Yes because instructions in loop will get executed again,

14. [5] Explain why or why not the instruction addresses in function **clear** exhibit spatial locality.

Yes because consecutive instructions get execute often.

15. [5] Explain why or why not the data addresses in function **clear** exhibit temporal locality.

`No because the same data address does not get fetched more than once. The variable i is in a register and not in memory.`

16. [5] Explain why or why not the data addresses in function **clear** exhibit spatial locality.

Yes because consecutive data addresses in the array are accessed.

17. Answer questions about the following C structure declaration. Put all answers in a file named **color.c** in your **`FinalExam`** directory.

```
typedef struct {
     char red;
     char green;
     char blue;
} Color;
```

a. [2] In the main program, declare a variable named **`red`** that is a pointer to a **`Color`** and initialize it to the value red = 255, green = 0, blue = 0 from memory that is dynamically allocated.

b. [2] IN the main program, declare an array named **`colors`** with 100 items, where each item is a pointer to **`Color`**. Just declare the array, don't give it any values.

c. [10] In the main program, initialize the array so that it contains 100 random colors, where each color component is a random value between 0 and 255 (inclusive).

For the most part everyone pretty much got these.

18. [10] Assume we have a virtual memory system with a 4KiB page size, a four entry fully-associative TLB, and a page table for some program **P**. The current state of the TLB and page table are shown below.

Assume we are using the age of a TLB entry to kick out the oldest valid page (*Least Recently Used*, LRU). For example, in the TLB below an age of 4 means it is the oldest entry. If it gets referenced again then its age gets reset to 1 and all of the other ages are incremented. **All numbers in the tables are in hex.**

TLB

Valid	Tag	Physical Page	Age
1	2	10	2
1	1	4	4
1	A	8	3
1	F	3	1

Page Table for program P

Index	Valid	Physical Page or Disk
0	1	9
1	1	4
2	0	Disk
3	1	2
4	0	Disk
5	0	Disk
6	1	8
7	1	F

Assume we have the virtual address sequence **`0xA022, 0x3124, 0x41AB, 0x332E`**

Label each address access as either a TLB hit or a miss keeping in mind that a previous address access may have caused a TLB miss that will have updated the TLB and the age for each entry. If a page needs to be brought in from disk then number the new page by incrementing the largest page number used so far. For example, start with **0x11** (because the largest physical page number used above is **0x10**).

Address	TLB Hit or Miss	Physical Address
A022	**H**	**8022**
3124	**M**	**2124**
41AB	**M**	**101AB**
332E	**H**	**232E**

Since the pages are 4KiB the lower 12 bits are the page offset. The upper bits are the tag for the TLB and the index for the page table. So in the address sequence the first hex digit is the tag or index.

`0xA022 – tag is 0xA that's a TLB hit and physical address is 0x8022. Reset the age to 1 (or 0).`

Valid	Tag	Physical Page	Age
1	2	10	3
1	1	4	4

1	A	8	1
1	F	3	2

0x3124 – tag is 3, that's a TLB miss so go to index 3 in page table. That is on physical page 2. So the physical address is 0x2124. But now also bring this in to the TLB by kicking out the oldest entry.

Valid	Tag	Physical Page	Age
1	2	10	4
1	3	2	1
1	A	8	2
1	F	3	3

0x41AB – This is a TLB miss and page table index 4 shows the physical page swapped to disk causing a page fault. We need to update the page table as well as the TLB. To update the page table we need to allocate a physical page by using the next highest page number 0x10 (which is one after the highest physical page of 0xF).

Here is the new page table with changes in green.

Page Table for program P

Index	Valid	Physical Page or Disk
0	1	9
1	1	4
2	0	Disk
3	1	2
4	1	10
5	0	Disk

6	1	8
7	1	F

And here is the new TLB

Valid	Tag	Physical Page	Age
1	4	10	1
1	3	2	2
1	A	8	3
1	F	3	4

0x332E – This is a page table hit. Thank goodness.