CS61C Summer 2013 Final Exam Rubric

Your Name: ______Peter Perfect_______________ **SID:** _00000000__________

Your TA (Circle): Albert Kevin Justin Shaun Jeffrey Sagar

Name of person to your LEFT: ___Sammy Student_____________________

Name of person to your RIGHT: ___Larry Learner________________________

This exam is worth 90 points and will count for 26% of your course grade.

The exam contains 7 questions on 14 numbered pages. Put all answers in the spaces provided. Some pages are intentionally left blank for scratch space.

Question 0: You will receive 1 point for properly filling out this page as well your login on every page of the exam.

Question	Points (Minutes)	Score
0	1 (0)	
1	23 (48)	
2	9 (20)	
3	15 (30)	
4	12 (20)	
5	17 (36)	
6	13 (26)	
Total	**90 (180)**	

All the work is my own. I had no prior knowledge of the exam contents nor will I share the contents with others in CS61C who have not taken it yet.

Signature: ________________________

Question 1: *Potpourri – Hard to Spell, Nice to Smell* (23 points, 48 minutes)

a) (3 points) MOESI on Through This Problem

Our computer has two cores, each with a 32B direct-mapped cache with 16B blocks using write-back and write-allocate policies. The MOESI protocol is implemented with invalidation of other caches on write, and the caches are empty at the beginning of the program. `arr` is a block-aligned array of `int`s.

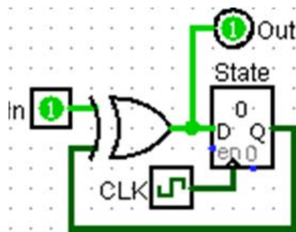
Fill out the status of the blocks in each cache. Indicate any memory locations that are not up-to-date as well. The first two rows have been done for you, using the abbreviations "C" for cache and "B" for block.

After Operation	Core 1 \$ State	Core 2 \$ State	Out-of-date Mem Locations
C1: read from `arr[0]`	B0: Exclusive B?: Invalid	B?: Invalid B?: Invalid	none
C2: write to `arr[5]`	B0: Exclusive B?: Invalid	B1: Modified B?: Invalid	`arr[5]`
C2: write to `arr[2]`	B0: Invalid B?: Invalid	B0: Modified B1: Modified	arr[5], arr[2]
C1: read from `arr[3]`	B0: Shared B?: Invalid	B0: Owned B1: Modified	arr[5], arr[2]

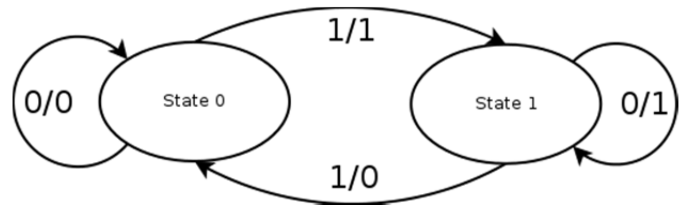
Follows directly from MOESI state definitions with cache invalidations.
+0.25 points per element in box, question score rounded up to nearest multiple of 0.5.

b) (4 points) Answer the following questions based on the FSM circuit shown below:

i. (2 points) Draw the FSM state diagram (assume the initial state shown) in the space below:



ANSWER:



-0.5 points per incorrect transition, -0.5 for incorrect binary state representation, min score of 0.

ii. (2 points) Let $t_{\text{setup}} = t_{\text{hold}} = 50$ ps and $t_{\text{XOR}} = 20$ ps. If we run this FSM on a 4-GHz processor and the input arrives t_{hold} after the clock triggers, what are the maximum and minimum $t_{\text{clk-to-q}}$ for the register to ensure proper functionality?

Min: ___30ps_____ Max: __180ps__

For proper functionality: $t_{\text{hold}} \le t_{\text{clk-to-q}} + t_{\text{XOR}} \le \text{T} - t_{\text{setup}}$
So min = $t_{\text{hold}} - t_{\text{XOR}}$ = 30 ps and max = $\text{T} - t_{\text{setup}} - t_{\text{XOR}}$ = 180 ps.

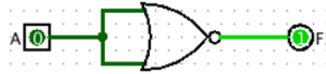
+1 point per blank, all or nothing.

---

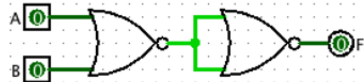
c) (3pts) Implement the following truth table functions using only NOR gates (fewer is better)

A	B	F1	F2	F3
0	0	1	0	1
0	1	1	1	0
1	0	0	1	1
1	1	0	1	1

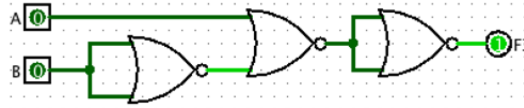
F1 (0.5pts):



F2 (0.5 pts):



F3 (2pts):



F1 = A', which can be accomplished with A NOR A or A NOR 0.
F2 = A OR B, which can be accomplished with F1(A NOR B) = (A NOR B)'.
F3 = A+B' using product of sums, which can be accomplished with F2(A,F1(B)).

(-0.5 points for using more gates than the solution. This penalty was applied max once for all circuits)
(-1.5 for not using NOR gates or not drawing gates)
(-1 for correctly writing A OR B' or A OR (A NOR B) for F3 but drawing an incorrect diagram)

---

d) (5 points) Hamm It Up

i. (2 points) What's the correct data word given the following SEC Hamming code: `0101000`?

Parity group 1: xor(0,0,0,0) = 0. Parity group 2: xor(1,0,0,0) = 1. Parity group 4: xor(1,0,0,0) = 1. So error in bit 6 means correct code word is 0101010, with data word 0010.

-0.5 pts for writing down the entire code word. The problem clearly says *data* word.

_____0010____

You are analyzing a new error detection/correction scheme for 4 bits of data ($d_1d_2d_3d_4$), where you store *two* parity bits ($d_1d_2d_3d_4p_1p_2$) with p_1 = xor(d_1,d_2,d_3,d_4) and p_2 = xor(d_1,d_2,d_3,d_4,p_1).
For example, if the data bits were 1011, the code word would be 101110.

Notice that the 6th bit (p_2) should *always* be 0 due to parity.

ii. (1pt) What fraction of all code words is valid? ____1/4______
Each data word maps to exactly 1 valid code word. 2^4 valid code words in 2^6 code words.

iii. (1pt) What is the minimum Hamming distance between valid code words? _____2_______
Change 1 data bit and p_1 to get to another valid code word.

iv. (1pt) What is the *maximum* number of errors we can *detect*? _____2_______
Can detect 1 error in $d_1d_2d_3d_4p_1$ as well as a simultaneous error in p_2 (should always be 0).

e) (5 points) There's Loot to be Had… Let's RAID It!

You are using RAID with 8 equally-sized disks and block-striping in 32-bit chunks.

i. (1pt) Which RAID level(s) can you *not* use? (-0.5 per wrong answer) ____1, 2, 3____
Need to be able to block stripe.

ii. (1pt) With which viable RAID level(s) can you to store the *most* data? _____0_______
No redundancy, so can use all of disk space for data.

(3,4,5 accepted if 0 was written for part i)

iii. (1pt) With which viable RAID level(s) can you store the *least* data? _____6______
RAID 6 requires TWO parity blocks per stripe.

(1 accepted if it was not included for part i)

iv. (2pts) If using RAID 5, how many disk reads and writes are there for writing 8 bytes of data within the same stripe?

Rd: __3___ Wr: __3___

8 B of data spans two disks. In order to calculate new parity block, need to read both old data blocks and old parity block. Then write new data blocks (2) and parity block.

1 pt per blank. -1 for 2/2 or 4/4

f) (3 pts) Solve for the *maximum controller overhead* to meet the following specifications:

We need disk latency under 18 ms while reading 800 B of data. The hard drive spins at 6000 rev/min with a seek time of 2.5 ms and transfer rate of 80 KB/s (SI prefix). Don't forget units!

__0.5ms______

controller overhead ≤ disk latency – seek time – rotation time – transfer time
rotation time = 0.5/rpm * (60 sec/1 min) = 0.005 s = 5 ms
transfer time = data size/transfer rate = 0.01 s = 10 ms
controller overhead ≤ 18 – 2.5 – 5 – 10 ms = 0.5 ms

(+1 point for correct rotation time calculation of 5ms)
(+1 point for correct transfer time calculation of 10ms)
(+1 point for correct formula Latency = Seek + Rotation + Transfer + Overhead)
(-0.5 for minor incorrect math. This penalty was applied once for the whole problem)

Question 2: *MIPStifying* (9 points, 20 minutes)

Answer the questions below about the following MIPS function. Answer each part separately, assuming each time that `mystery()` has not been called yet.

```
        mystery:
1               andi  $a0, $a0, 3
2               ori   $t0, $0,  1
3               sll   $t0, $t0, 6
4       Lbl1:   beq   $a0, $0,  Lbl2
5               sll   $t0, $t0, 5
6               addi  $a0, $a0, -1
7               j     Lbl1
8       Lbl2:   la    $s0, Lbl3
8               lw    $s1, 0($s0)
9               add   $s1, $s1, $t0
10              sw    $s1, 0($s0)
11      Lbl3:   add   $v0, $0,  $0
12              jr    $ra
```

a) (0.5 pts) Which instruction (number) gets modified in the above function? ___11________

All-or-nothing. Credit was given for writing the MIPS instruction (`add $v0, $0, $0`).

b) (1 pt) Write an equivalent *arithmetic* (not logical) C expression to instruction 1.

`a0 =` ___a0 % 4____

0.5 pts given for using the modulus operator with wrong arguments. 0 pts for using bitwise and (&) because the question specifically stated not to use logical expressions.

c) (2 pts) Which instruction field gets modified when mystery is called with `$a0 = 3`? ___rs________

Executing mystery with $a0 = 3 results in $t0 being shifted left by 21. The 1 bit in $t0 was aligned with the last bit of the rs field, so the addition incremented rs by 1, changing $0 to $at.

1 pt was given for off-by-one errors (rt), and 0.5 pts were given for opcode and rd.

d) (0.5 pts) How many times can `mystery(2)` be called before the behavior of `mystery()` changes?

___0 or 1_____

Calling mystery(2) increments rt of line 11, which changes the return value of the function. Based on both the argument that we can't claim that the function changes before it is executed and the point value of the question, either 0 or 1 was accepted for full credit.

e) (2 pts) How many times can `mystery(0)` be called before the behavior of `mystery()` changes?

______31_____

Calling mystery with $a0 = 0 adds 1 to the shamt field, which is unused in an add instruction. The behavior of the function won't change until the sum overflows into the rd field, which will take 32 times. Thus, mystery can be called 31 times before the behavior changes.

0.5pts were deducted from an answer of 32.

f) (3 pts) A program calls `mystery` with the following sequence of arguments: 0, 1, 2, 3, 4, 5. What MIPS instruction gets stored in memory?

__add $a0, $at, $at___

The first instruction takes the modulus of $a0 by 4, so it was equivalent to calling the function with arguments 0, 1, 2, 3, 0, 1. Thus, rs and rt incremented by 1 while rd and shamt are incremented by 2.

1.5 pts were given for finding the correct instruction and 1.5 pts were given for finding each correct argument. Those who wrote the correct instruction but no arguments received a penalty of 0.5 pts. A common mistake was to forget about the modulus—those answers received 2pts. Partial credit was given in certain cases if work was shown.

Question 3: *To Be Without Parallel… Means You're Slow* (15 points, 30 minutes)

a) SIMD and OpenMP

Four CEOs are playing the board game Monopoly, where the object of the game is to own properties and gain profits from them. You are in charge of keeping track of the CEOs' finances and wish to parallelize this task. All memory accesses are valid. Assume `sizeof(int) = 4`.

```
int balance[4];              // global array of balances
Property props[NUMPROPS];    // global array of properties
```

A property is defined by the following struct:

```
typedef struct {
    int owner;     // the index of the CEO who owns this property
    int profit;    // collectable money
} Property;
```

Every round we must give each CEO the amount of money that he has earned from each of his properties. To do this, we add the property's profit into the CEO's balance and then set the profit to zero for the next round using the following function (+0.5 points per line – part i):

```
void collect_profits() {

    for(int i = 0; i < NUMPROPS; i+=2) {
        balance[props[i].owner] += props[i].profit;
        props[i].profit = 0;
        balance[props[i+1].owner] += props[i+1].profit;
        props[i+1].profit = 0;
    }
}
```

i. (1 pt) Perform a 2-fold unrolling of the loop by filling in the blank spaces above. You may edit the looping conditions if you need to (cross out and write in changes). Assume `NUMPROPS` is a multiple of 2.

-0.5 pt if NUMPROPS edited improperly
-0.5 pt if don't increment i+=2 each iteration
+0.5 pt per line, 0 pt minimum

ii. (2 pts) `_mm_loadu_si128` loads 128-bits of data into a vector. If we wish to use **three** of these instructions to load from `props[]` every iteration of our loop, how many fold must we unroll the original loop? Answer `n` if performing an `n`-fold unrolling. _____6_______

Since our Property struct is 64 bits wide, we can load two of them at a time with each 128 bit SIMD _mm_loadu_si128 instruction. Therefore we can load 6 properties with 3 _mm_loadu_si128 instructions, which would require a 6-fold unrolling.

iii. (1 pt) You slap a `#pragma omp parallel for` statement on the original `for` loop. Circle the effect on execution below and provide a one phrase/sentence explanation. No credit without explanation.

Correct; Faster	Correct; Slower	**(circled)** Almost always Incorrect	Segfault

Explanation: We have a data race for writes to balance.

0.5 pts for claiming "data dependency" on balance.

iv. (1 pt) Now you additionally add a `#pragma omp critical` statement around all writes to `balance[]`. What is the new effect on execution? No credit without explanation.

Correct; Faster	**(circled)** Correct; Slower	Almost always Incorrect	Segfault

Explanation: Our threads effectively proceed sequentially, but they do compute the correct result. Slower due to cache invalidations on writes to balance.

0.5 pt for "false sharing", because writes to the same location in balance by different threads wouldn't be considered false sharing.

b) MapReduce

Suppose that given a large social network dataset, you wish to generate recommendations for yourself by looking at the "Likes" of your friends. You wish to exclude your own likes so that the recommendations are useful. Unfortunately, your input dataset consists of everyone on the social network, not just your friends. Assume that you have access to a global person_id for yourself, **`YOUR_ID`**.

Conceptually, you can think of the `map` as filtering the overall dataset to just you and your friends, while the `reduce` will filter out common "Likes" between you and your friends.

You have access to the following special methods:

`list1 = removeAll(list1, list2)` // `list1` now contains only elements that were in `list1` // but not in `list2`

`list_ret = sortOnValues1(list_arg)` // assuming each element in list is a tuple, returns // copy of `list_arg` sorted on the 1st element of the tuple

Feel free to access members of lists and tuples using **array syntax**.

i. (1 pt) The input to the `map` function is (key = person_id, value = (friend_list, likes)). How many times will a friendship between two people show up in the input data? ______2_____

Once as (you, friend) and once as (friend, you).

ii. Fill in the MapReduce functions below using Java-like pseudocode:

```
map(key, value){
    friend_list = value[0];
    likes = value[1];
    person_id = key;
    for (friend: friend_list) {
        if (person_id == YOUR_ID) {
            emit (( YOUR_ID, ___friend______), (0, likes));
        } else if (___friend==YOUR_ID__________) {
            emit ((__YOUR_ID_____, _____person_id___), (1, likes));
        } //  1 point per above blank, -1 for swapping YOUR_ID, person_id
    }
}
reduce(key, values){
    emitValues = ____sortOnValues1(values)____________; // 1 point
    emit(key, removeAll(_emitValues[1][1]___, __emitValues[0][1]___));
                        //     2 points                2 points
}
```

Only emit from map if you are in the friend pair. Include 0 or 1 in value to differentiate whether they are your likes or your friend's likes. Sort on this 0/1 value in reduce to remove your likes from your friend's likes.

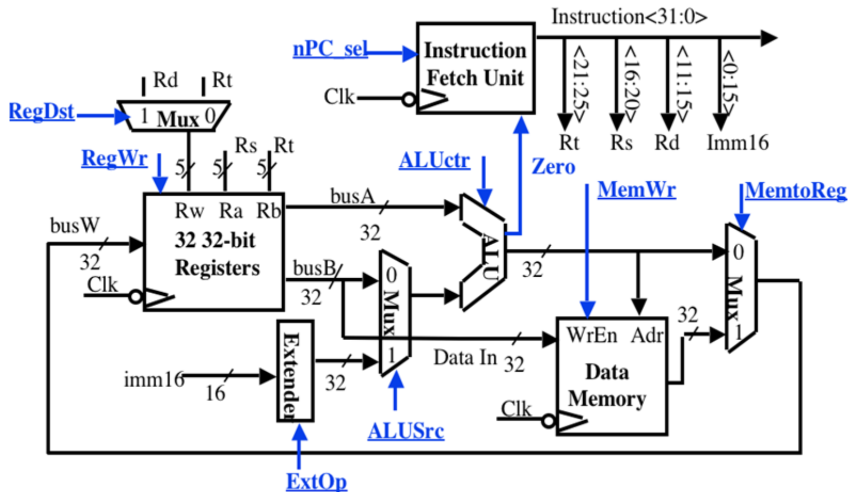
-1 for using something like friend_list.contains(YOUR_ID) (produces too many emits)
-1 for indexing errors
-0.5 for VERY minor indexing errors
-1 for incorrect ordering in final emit (but otherwise correct)
-0.5 for using values instead of emitValues

<u>Question 4</u>: *Off the Beaten Datapath* (12 points, 20 minutes)

Add the instruction `stones` (<u>st</u>ore <u>one</u> <u>s</u>maller) to the single cycle datapath, which stores a 1 at the address of the smaller value between two specified registers. **Ignore pipelining.**

That is,

```
unsigned x, y;
char *p = NULL;
if (x < y)
    *(p+x) = 1;
else
    *(p+y) = 1;
```



NOTE: A surprisingly large amount of students misinterpreted the instruction and updated the minimum register itself (rather than the memory address that the register contained). The sample C code should have made the correct behavior unmistakable; however it was still missed in about 20% of the exams. If the behavior of the instruction was misinterpreted, we deducted a flat 3 points for the mistake and then graded the problem as if we had asked for the misinterpreted behavior.

a) (2 points) Write out the assembly syntax and RTL for this instruction. Don't forget about the PC!

Syntax: **stones $rs, $rt** **RTL:** **Mem[(R[rs] < R[rt]) ? R[rs] : R[rt]] = 1; PC = PC+4;**

Syntax was worth .5 points, all or nothing. If actual registers were used instead of the general $rs,$rt, credit was still granted. If the "$" symbols were left off, credit was still granted. We were lenient here.

RTL was worth 1.5 points; updating the PC was worth .5 points, and the RTL for the memory update was worth 1 point. If the RTL was syntactically incorrect, but still had the correct logic, full credit was given. Minor errors in logic received .5 point deductions.

b) (5 points) Change as *little as possible* in the datapath above **(draw your changes on the figure)** to enable `stones`. List all your changes below (be concise!). Your modification may use MUXes (define what select bits refer to what inputs), wires, constants, and up to one new control signal, but <u>nothing else</u>. You may not need all of the provided boxes. You cannot modify the ALU (there is no `min` operation). (2 point per cascaded MUX, 1 point for other MUX).

There were 3 MUXes that needed to be added to ensure this instruction worked correctly and other instructions would still behave properly. Points were awarded if the complete behavior of each MUX (where the inputs came from, where the output went, where the select bit came from and to which input it mapped) could be determined from the description, the diagram, or both. The lack of each of these details resulted in a half point penalty. The MUXes described in (i) and (ii) below were worth 2 points each, while the MUX in (iii) was worth 1 point. The use of additional unnecessary hardware received a .5 point penalty. The use of disallowed hardware (such as comparators or changing the ALU) received only 1 point, as this removed the need for two of the MUXes. Describing the addition of a new control signal was not graded until part (c).

(i)	MUX to select between busA (S=1) and busB (S=0) for input addr to Data Mem. The select bit is the LSB of the ALU output after the sltu instruction (we want the minimum value).
(ii)	MUX to select between output of MUX from above (S=1) and ALUOut (S=0) for AddrIn to DataMem. Select bit is new STONES control signal.
(iii)	MUX to select between constant 0x1 (S=1) and busB (S=0) for Data In into DataMem. Select bit is new STONES control signal.
(iv)	NOT NEEDED

c) (5 points) We now want to set all the control lines appropriately. List what each signal should be, using 0, 1, X, or an intuitive name. Include any new control signals you added. (0.5 points / signal)

Every signal was worth .5 points, with the exception of ALUctr which was worth 1 point. 1 point was granted for using the instructions sltu, sub, or subu, and .5 was given for the instructions slt, or the words "less than" or "greater than". Signals must have been unambiguous; 0 or 1 for nPC_Sel was marked incorrect. Signals that did not have an effect on the instruction must have been marked X for credit. Using an extraneous control signal nullified the points for the STONES control signal. STONES was the name of the control signal we came up with; other names were still given full credit.

RegDst	RegWr	nPC_sel	ExtOp	ALUSrc	ALUctr	MemWr	MemtoReg	STONES
X	0	PC+4	X	0	sltu	1	X	1

Question 5: *Tread Carefully, Thread Carefully* (17 points, 36 minutes)

Examine the function prototype and MIPS implementation below.

```
// sets *value = (*value) * 2^pow using shifting instructions
int multMemPow2(int *value, unsigned int pow);

      multMemPow2:
1              lw   $v0, 0($a0)     # load value
2      loop:   beq  $a1, $0, exit   # exit condition
3              sll  $v0, $v0, 1     # multiply by 2
4              addi $a1, $a1, -1    # decrement counter
5              sw   $v0, 0($a0)     # store result
6              j    loop
7      exit:   jr   $ra
```

We are using a 5-stage MIPS pipelined datapath with separate I$ and D$ that can read and write to registers in a single cycle. Assume no other optimizations (no forwarding, no branch prediction, etc.). The default behavior is to stall when necessary. Branch checking is done during the Execute stage.

For parts (a)-(c), let `pow=1`. When we ask for clock cycles to execute `multMemPow2`, we mean from the instruction fetch of `lw` up to and including the write back of `jr`.

a) (1 point) How many instructions are executed in `multMemPow2`?

`pow=1`, so execute `lw, beq, sll, addi, sw, j, beq, jr` → 8 instr ______8______
+0.5 pt if answered 13 for pow=2.

b) (4 points) How many clock cycles does it take to execute `multMemPow2`?

(See pipelining table below) _____18______

Potential Structural Hazards: none, MIPS is great!

Potential Data Hazards: 1-3 (taken care of by stalls for branch control hazard),
3-5 (needs 1 stall because already one instr in-between)

Potential Control Hazards: beq (2 stalls because only know next instr *after* EX stage),
j (1 stall because get target addr in ID stage)

So for the 8 instructions listed above, you get 8 instr + 4 fill/drain + 6 stalls = 18 cycles.

Cycle #	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
lw	F	D	X	M	W													
beq		F	D	X	M	W												
sll					F	D	X	M	W									
addi						F	D	X	M	W								
sw								F	D	X	M	W						
j									F	D	X	M	W					
beq											F	D	X	M	W			
jr														F	D	X	M	W

The number in the blank was ignored unless you did not show work or had an egregious math error.

+1 pt for recognizing the 3-5 data hazard stall. -0.5 pt if wrong # of stalls.
+1 pt for recognizing the j stall. -0.5 pt if wrong # of stalls.
+1 pt for recognizing the two beq double stalls. -0.5 pt if wrong # of stalls.
+0.5 pt for the 4 cycle pipeline fill/drain.
+0.5 pt for answer to part (b) > answer to part (a).

c) (3 points) Consider the following optimizations *separately*. How many FEWER cycles are taken for the addition of each optimization?

i. (1.5 points) Forwarding ______1_______

Only eliminates 3-5 data hazard stall. Full credit given if answer matched # of data hazard stalls shown in part (b).

ii. (1.5 points) Branch Prediction of always taken ______2_______

Only eliminates 2nd beq stalls. Flushing on an incorrect prediction happens after the EX stage, so it's the same 2 stalls for the 1st beq. Full credit given if answer matched # of stalls for 2nd beq shown in part (b). +0.5 pt if answer was sum of beq stalls shown in part (b).

d) (2 points) Suppose we introduce only jump delay slots and want to move a loop instruction into the new jump delay slot after instruction 6 (`j loop`). For the following candidate instructions, answer **C** for "changes behavior," **S** for "causes additional stall(s)," or **G** for "good choice":

Instr 3: ____C_(0.5 pt)____	Instr 4: ____S_(1 pt)______	Instr 5: ____G_(0.5 pt)_____
Storing different $v0 in sw	Data hazard on $a1 with beq +0.5 pt if answered G	Works!

For the following questions, assume we are executing the `multMemPow2` simultaneously on TWO processors in the same shared-memory machine with **`*value=1` and `pow=2`**.

e) (3 points) List ALL possible values of `*value` after execution: __4, 8, 16_____

For `pow=2`, each thread performs one load and two stores. One thread will *always* read `*value=1` and stores 2 and 4 in `*value` as it executes. The other thread can read `*value=1`, `2`, or `4` and will thus store 4, 8, or 16 at the end. Even if the first thread stores last, it will store 4.

+1 pt for each correct number, -0.5 pt for each *additional* number.

Stanley Stanfurd thinks he can fix the data race problem by replacing the `lw` with `ll` and `sw` with `sc`. Assume `sc` compares against the value from the last `ll` call.

f) (4 points) List ALL possible values of `*value` after execution of this new version: ____2, 4______

So the interesting thing is that regardless of what `pow>0` is, `sc` will only work *once*, because you don't call `ll` again! This saves us from the headache of tracking `$v0`, since its value also gets clobbered by `sc`. Again, one thread will *always* read `*value=1` and stores only 2 in `*value` as it executes. So the other thread can read `*value=1` or `2` and will thus store 2 or 4 at the end.

Max score if you did not recognize that there should be fewer possible values in (f) and (e) was 2 pt.

Responses:	0 (1 pt)	0,1 (2 pt)	0,2 (3 pt)	2 (2pt)
	4 (2 pt)	4, 8, 16 (1 pt)	4, # other than 2 (3 pt)	

Question 6: *It's Virtual Insanity!* (13 points, 26 minutes)

Our 32-bit uniprocessor machine has 1 GiB of RAM with 1 KiB pages, a fully-associative TLB that holds 8 entries and uses LRU, and a direct-mapped, write-back *data* cache with 32 B blocks and 32 slots. The *instruction* cache is 256 B and fully-associative with 32 B blocks.

a) (1 point) What is the maximum number of valid entries in the page table for a single process? Answer in IEC.

1 Mi-entries

Page table valid entries set by size of PM. 1 GiB / 1 KiB = 1 Mi-entries.

+0.5 points for 1MiB, +0.5 points for 2^{20}

b) (1 point) What is the TLB Reach of our system? ___8 KiB_____

8 TLB entries that refer to a 1 KiB page each. TLB Reach = 8*1KiB.

No partial credit – full credit given for 2^{13} B.

Examine the following function. Assume the entire program's code takes the entirety of one page and `sizeof(int)=sizeof(int *)=4`.

```
void addConst(int *ptr, char c) {
    for(int i = 0; 1; i+=4)
        ptr[i] += c;
}
```

***If all of (c,d,f) were answered as if for i++, -2 points total after the scores for those are added (min 0)**

c) (2 points) If `ptr[]` lives in disk and `ptr[0]` is page-aligned, what is the TLB hit rate for data accesses only?

___127/128___

Lives in disk means TLB miss on `ptr[0]`. Loop jumps 4 ints = 16 B per iteration. 1 KiB/16 B = 64 array indices accessed per page. Since += is a read and a write, 1 TLB miss per 128 memory accesses in a page.

+ 1 pt total for answering 63/64 based on missing that += does read and write
+ 2 pt total for answering 511/512 if answering based on i++ interpretation*

d) (2 points) If `ptr[]` lives in disk and `ptr[0]` is page-aligned, what fraction of D$ misses are also TLB misses?

____1/32_____

From part (c), 1 TLB miss every 128 memory accesses. Lives in disk means not in cache. 32 B/16 B = 2 array indices accessed per cache block. Since += is a read and a write, you have 1 D$ miss per 4 memory accesses in a cache block. The fraction is then 4/128 = 1/32.

Partial credit only for slight arithmetic errors with work shown, which were evaluated individually.

e) (1 point) If ptr[0] is in physical memory, what is the *minimum* value of i that could cause a **page fault**?

_____4_______

ptr[0] is in valid entry in page table, but no mention of where in page. If we assume it is one of the last 3 integer spaces in the page, then the next loop iteration (ptr[4]) can cause a page fault if that page is invalid.

+ 0.5 pt for 256 based on assuming page alignment
+ 1 pt for answering 1 based on i++ interpretation*

f) (1 point) If ptr[0] is in physical memory, what is the *minimum* value of i that could cause a **protection fault**?

_____0_______

Even with ptr[0] in valid entry in page table, no mention of access rights. The loop both reads and writes, so will cause a protection fault if the process is missing one of those access rights for that page.

+ 0.5 pt for 4
+ 0.5 pt for answering 1 based on i++ interpretation*

g) (2 points) If ptr[0] is in physical memory, what is the *maximum* value of i that causes the first **cache miss** in the loop?

____256______

For maximum, assume I$ already holds all instructions of the loop and that the D$ is filled with the first entries of ptr[]. The D$ holds 32 * 32B = 1 KiB of data = 256 ints.

+ 0.5 pt for answering 32 based on a proposed scenario of one block in the cache
+ 0.5 pt for answering 64, the max number of iterations

h) (3 points) If ptr[0] is in physical memory, what is the *maximum* value of i that causes the first **TLB miss** in the loop? You may leave your answer as a product.

___7*2^8_____

For maximum, assume TLB is full of page entries needed for our function. Each page holds 1 KiB = 2^8 ints. Need one page for code/instructions, so 7 remain for ptr[] entries. So first TLB miss (and replacement) will occur when i = $7*2^8$.

+ 2 pt for answering 2048 based on ignoring the code page
+ 1 pt for answering 256 based on assuming only the page holding ptr[0] was in physical memory
+ 1.5 pt for answering 512 based on counting the number of iterations