CS61c Summer 2014 Final Exam

Read this first: This exam is marked out of 100 points, and amounts to 30% of your final grade. There are 9 questions across 16 pages in this exam. The last question is extra credit – it will not be included in our calculation of the final class curve.

You may use two double sided 8.5x11" page of handwritten notes, as well as the MIPS green sheet provided to you as references throughout this exam. Otherwise your only resource is what you know or can deduce during the examination period. Answers should be placed inside the spaces provided on the test, and should be no longer than is necessary to communicate the solution. Misplaced or exceedingly verbose solutions may be marked down.

You may find this exam difficult. If you do, then *do not panic*. Remember that different questions have different weights, and some may be more difficult than others. Answer those questions and subquestions in which you are most confident first, then move on to more difficult questions. We recommend that you skim over the entire exam before attempting to answer any questions. Also be sure to read all questions carefully before answering them, and do not hesitate to ask for clarifications if you are unclear on what a question is asking.

There are some true-or-false questions in this exam. Correct answers to these questions will receive full credit, but incorrect answers will full *negative* credit. E.g. if a T/F question is worth 1 point then a correct solution yields +1 points, but an incorrect answer yields -1 points. Note that you may get negative points in total on such a question if you choose randomly.

Your Name: _____

Your TA: Andrew David Fred Hokeun Kevin

Login: _____

Login of the person to your left: _____

Login of the person to your right: _____

Question	Points	Time (minutes)	Score
M1	12	21	
M2	12	21	
M3	12	21	
M4	12	21	
F1	12	21	
F2	16	30	
F3	12	21	
F4	12	21	
F5	1	3	
Total	101	180	

M1: The Smorgasbord

(a) How many bits do you need to represent all the characters in c?

(b) If we were to modify the IEEE standard so that floats did not have an implicit leading 1, how would the number of representable numbers change (more, fewer, or no difference)? Briefly explain why.

(c) In an a.out file's text section, you see the left six bits of an instruction is 0x04. As a result of executing this instruction:

What is the most amount of bytes that your PC can change? _____

What is the absolute least amount of bytes that it can change? _____

(d) Write two git commands that will track the file "final.c" and commit those changes with the message "DONE".

(e) Lets say we created a new form of MIPS where there are a total of 64 - 32 bit registers, but we still have a 32 bit address space. To make this change possible, we remove the rd field for R-types and assume that we will write to rs. For example **add $t2, $t1** adds $t1 and $t2 together and stores the result in $t2. We use however many bits we need for rs, rt and shamt and use the remaining bits for funct.

How many different operations can we have with this new version of MIPS? (Leave your answer as an expression) Just for clarification: **addu $t1, $t2** and **addu $t4, $t5** are considered the same instruction

M2: Call me once, return to you. Call me twice, return to ... who?

Here is a MIPS function to be deciphered. Annotate this function with the values of the specified registers after each instruction executes. Please use the following formats:

- $R[15:0]$ - gives the less significant half of R.
- 0×13 - gives 13 consecutive 0s.
- 1×12 - gives 12 consecutive 1s.
- $0 \times 28 \mid 1 \mid 0 \times 3$ - gives 0b1000 (8).

	mystery:	addiu $v0 $ra $0	$v0: $R[31:0]$
		la $t0 mystery	
1		addiu $t2 $0 -1	$t2:
2		sll $t2 $t2 28	$t2:
3		nor $t2 $0 $t2	$t2:
4		and $ra $t2 $v0	$ra:
5		sra $ra $ra 2	$ra:
6		lui $t2 0x800	$t2:
7		or $ra $t2 $ra	$ra:
		sw $ra 4($t0)	
		jr $v0	

Here **mystery** is called for the first time in the following way. Assume that we are not using a delayed-branching MIPS.

```
0x00000100:       ...
0x00000104: foo:  jal mystery
0x00000108:       ...
```

(a) If **mystery** is only called this one time, what address will it return to?

(b) What value will it return?

Suppose **mystery** is called again in the following location.

```
0x0000100C:       ...
0x00001010: bar:  jal mystery
0x00001014:       ...
```

(c) If **mystery** is called this second time, what address will it return to?

(d) What value will it return?

(e) What is the maximum byte address that **mystery** can be called from for the first time still function as previously described? Please leave answers relative to powers of 2.

M3: A is for apple, C is for cache

We would like to apply the concept of caching to computed data, defining a simulated cache in software, to cache some expensive computations. We will be looking at an implementation of a software N-way set associative LRU cache. The cache sets are contiguously packed into an array of single-element blocks. Instead of entire cache lines, single computed values should be stored.

```
/* Defines a single one-element block in the cache. */
typedef struct { int valid; int tag; int value; } block;

/* Contains the entire cache's settings and data. */
typedef struct { block * blocks; size_t size; size_t associativity; } cache;

/* Resizes the given cache to the given size and associativity
 * Also initializes the given cache if it was not initialized previously */
void cache_resize(cache * sim, size_t new_size, size_t new_associativity);

/* Copies the contents of one cache to another cache. Assumes both caches
 * have been initialized. Does not necessarily respect the LRU ordering
 * from the source cache */
void cache_copy(cache * dst, cache * src);

/* Returns the cached value of a tag or 0 if the tag is not in the cache
 * "contains" is used to alert when a tag was not found */
int cache_get(cache * sim, int tag, int * contains);

/* Puts a value with the given tag into the cache
 * Also updates the LRU of the set it is put in */
void cache_put(cache * sim, int tag, int value);

/* Updates the LRU of the set starting at array index "base_index"
 * The LRU block at array index "base_index + way" is shifted up to "base_index" */
void cache_update_LRU(cache * sim, size_t base_index, size_t way);
```

Fill in all of the following sections. Each definition is pasted directly into every instance where it is used. You may use assume any memory allocations always succeed.

SECTION_A		SECTION_B	
SECTION_C		SECTION_D	
SECTION_E		SECTION_F	

```
void cache_resize(cache * sim, size_t new_size, size_t new_associativity) {
    cache alt;
    alt.size = new_size;
    alt.associativity = sim->associativity = new_associativity;
    alt.blocks = SECTION_A;
    if(SECTION_B) {
        cache_copy(&alt, sim);
    }
    sim->size = new_size;
    free(SECTION_C);
    SECTION_D;
}

void cache_copy(cache * dst, cache * src) {
    for(size_t n=0; n<src->size; n++) {
        if(SECTION_E) {
            cache_put(dst, src->blocks[n].tag, src->blocks[n].value);
```

```
        }
    }
}

int cache_get(cache * sim, int tag, int * contains) {
    size_t base_index = (tag % (sim->size / sim->associativity)) * sim->associativity;
    for(size_t n=0; n<sim->associativity; n++) {
        if(sim->blocks[base_index+n].valid && sim->blocks[base_index+n].tag == tag) {
            cache_update_LRU(sim,base_index,n);
            SECTION_F = 1;
            return sim->blocks[base_index].value;
        }
    }
    SECTION_F = 0;
    return 0;
}
```

The remaining function implementations are provided for your convenience, but have no sections in need of completing.

```
void cache_put(cache * sim, int tag, int value) {
    int contains = 0;
    cache_get(sim, tag, &contains);
    if(!contains) {
        size_t base_index = (tag % (sim->size / sim->associativity)) * sim->associativity;
        block * place = sim->blocks + (base_index + sim->associativity-1);
        place->value = value;
        place->tag = tag;
        place->valid = 1;
        cache_update_LRU(sim, base_index, sim->associativity-1);
    }
}

void cache_update_LRU(cache * sim, size_t base_index, size_t way) {
    block tmp = sim->blocks[base_index+way];
    for(size_t n=way; n>0; n--) {
        sim->blocks[base_index+n] = sim->blocks[base_index+n-1];
    }
    sim->blocks[base_index] = tmp;
}
```

M4: Cache, gotta hit it! Missing it would be painful.

(a) Following statements are related to cache optimization. Specify whether each of the following statements would generally be true or false. (Reminder: incorrect answers on T/F questions are penalized with negative credit)

T / F – Assuming the same cache size and the same block size, increasing set associativity of a cache reduces conflict misses.

T / F – Assuming the same set associativity and the same block size, increasing the size of a cache reduces compulsory misses.

T / F – Smaller caches have shorter hit time than larger caches.

T / F – Adding a lower level cache reduces miss penalty.

T / F – Adding a lower level cache increases hit time.

T / F – Increasing set associativity increases hit time.

For following questions through (b) to (d), we will be working on a 32-bit byte addressed MIPS machine, with a single direct mapped 1KiB cache, write-through and no-write allocate policies, and 16B blocks. The cache is initially cold for each question.

Consider the function `matrix_multiply` listed below, which multiplies the matrix A by itself and stores the result into the matrix C. Note that starting addresses of each Matrix are as in comments.

```
#define N    4
int A[N * N]; // starts at address 0x10000
int C[N * N]; // starts at address 0x20000

void matrix_multiply() {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            for (int k = 0; k < N; k++) {
                C[i*N+j] += A[i*N+k] * A[k*N+j];
            }
        }
    }
}
```

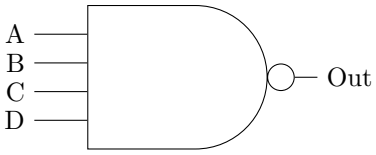
(b) What is the cache hit rate of the function `matrix_multiply` while i = 0?

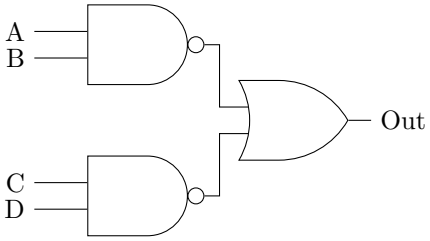
(c) If our cache is two-way set-associative with LRU replacement policy, what would be the hit rate of the function `matrix_multiply` while i = 0?

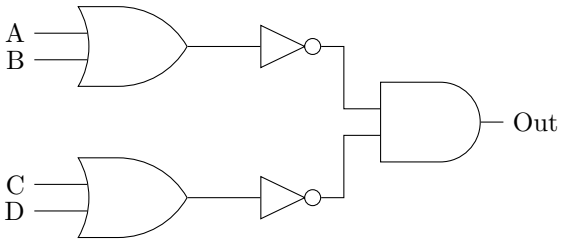
(d) If our cache is two-way set-associative with LRU replacement policy and its block size is 32B, what would be the hit rate of the function `matrix_multiply` while i = 0?

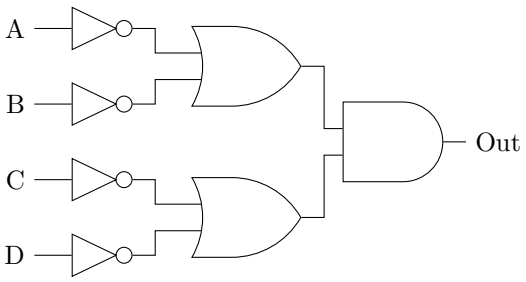
F1: True or false, but not the kind you like

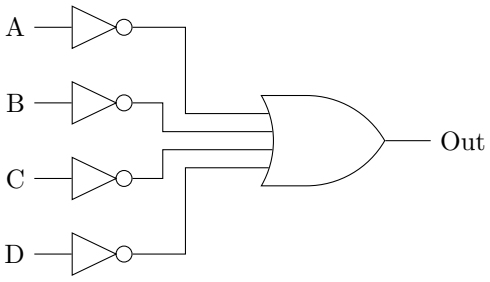
(a) Sometimes circuits are equivalent, even though they appear to be quite different. There is one group of equivalent circuits listed below, circle the circuits that belong to that group. For full credit, you must show your work.









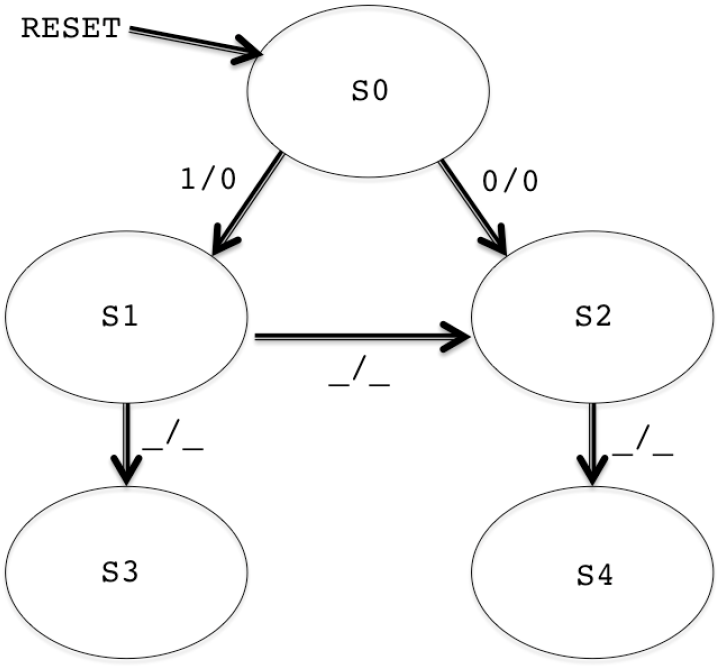


(b) 2-input NOR gates are said to be complete because any Boolean function can be computed with them. Prove this fact. Hint: implement a subset of the standard gates (AND, NOT, OR, NOR, NAND, XOR, XNOR) using just NOR gates, then apply a standard boolean algebra technique using these gates.

(c) We want to implement a very simple finite state machine that determines its next state by the result of and AND operation on the current state and the input. The output is always the current state. Assume registers have a CLK to Q delay of 5ns, a setup time of 2ns, and a hold time of 3ns. To achieve a clock rate of 25MHz, what is the maximum propogation delay that a NOR gate could have, assuming we are implementing AND as a combination of one or more of the gates built in part (b)?

(d) Complete the state diagram for a finite state machine that outputs 1 if and only if it has just seen the input sequence 101 and it has never seen the input sequence 001. You may add more arrows or more states as you see fit. Provide a brief description of each state.

```
Example
Input : 1101010100101
Output: 0001010100000
```



F2: Datapath-ology

We want to extend the familiar MIPS datapath to expedite memory operations such as **memcpy**, **strcpy**, **memset**, and such. We will implement 3 new R-Type instructions, all of which use counters to allow quickly scanning through memory values. We will use two 0-indexed counters named **NL** and **NS**, respectively for loads and stores.

rn	Resets both memory counters.
lbn $rd, $rt, ($rs)	Computes an address at **NL** offset from $rs. The contents of memory at this address are loaded into $rt. **NL** is stored into $rd. Then **NL** is incremented.
sbn $rd, $rt, ($rs)	Computes an address at **NS** offset from $rs. The contents $rt are stored into memory at this address. **NS** is stored into $rd. Then **NS** is incremented.

For sanity's sake, we will assume that $rd and $rt are never equal for the duration of this topic.

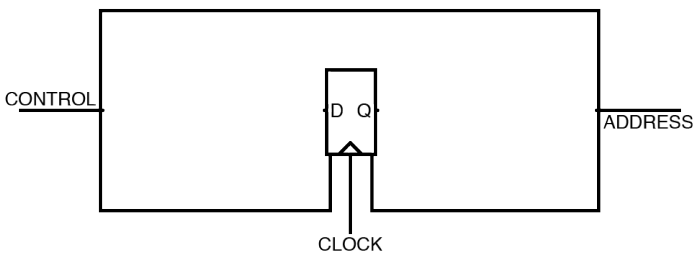
Fill out the RTL descriptions for these three operations.

rn	
lbn $rd, $rt, ($rs)	
sbn $rd, $rt, ($rs)	

Before we implement these instructions, let's review how they work. Complete the **strcpy** MIPS code below to copy the string at address **$a0** to address **$a1** using only TAL MIPS instructions and these new instructions. Write only one instruction in each of the three blank lines. Assume we are not using delayed-branch MIPS.

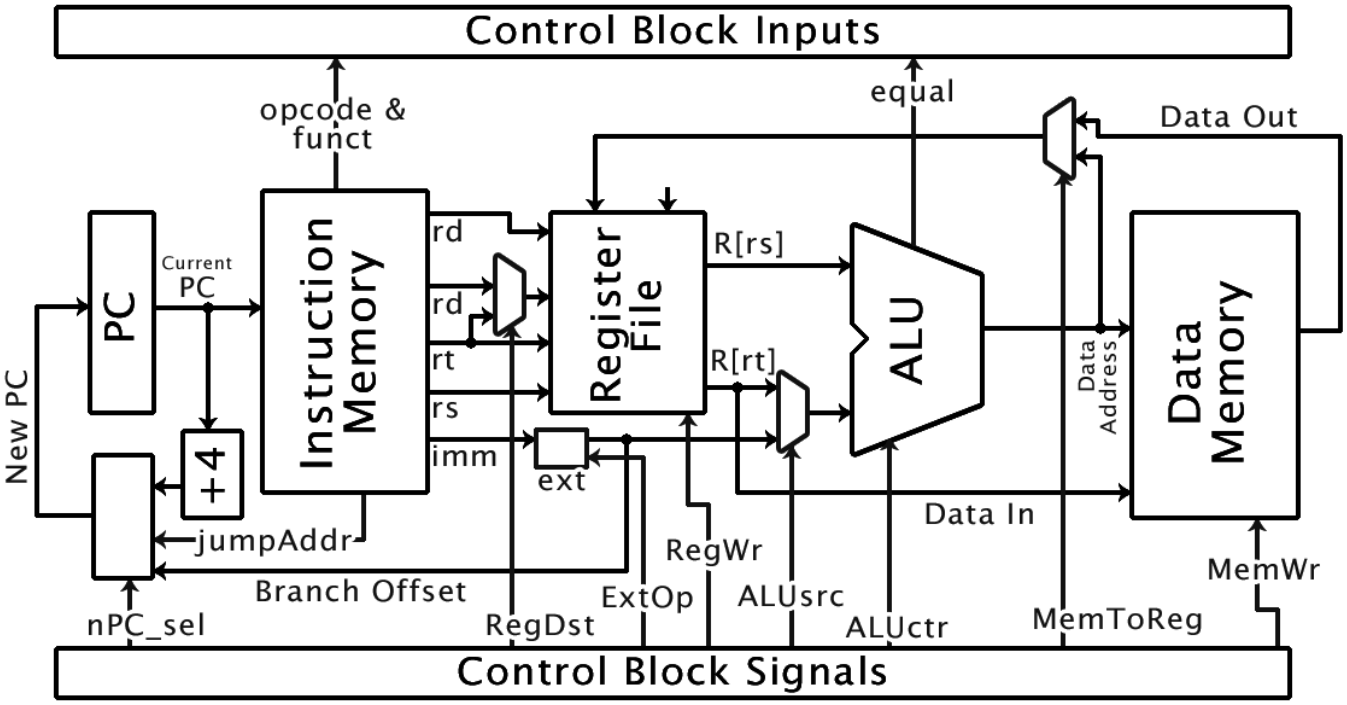
	`strcpy:  rn`
1	
2	
3	
	`jr $ra`

Now, let's implement these instructions in the datapath. First, we want to implement counters for use in the datapath. We are going to base our counters off of simple registers that do not include **enable** or **clear** pins. Assume **address** is 32 bits, the register holds 32 bits, **clock** is the clock pulse, and **control** is some size less than 32 bits. Your counter implementation should support counter operations from all three instructions. Do not gate the clock!



We have the familiar single-cycle MIPS datapath, ready for modification to support these operations. To get you started, we have added a second register write port, and a new associated input **rd**, indicating that this register write port always writes to **rd**.

Not counting the changes already made, list the most minimal set of changes to the following datapath diagram required to enable these three instructions. In this case, minimal refers to the number of added or modified components, as well as the number of added or modified control signal options. Please list one component addition or modification per line. For each change, list any new control signal that you might have introduced. You may not need all the provided boxes. Also draw your changes onto the datapath diagram.

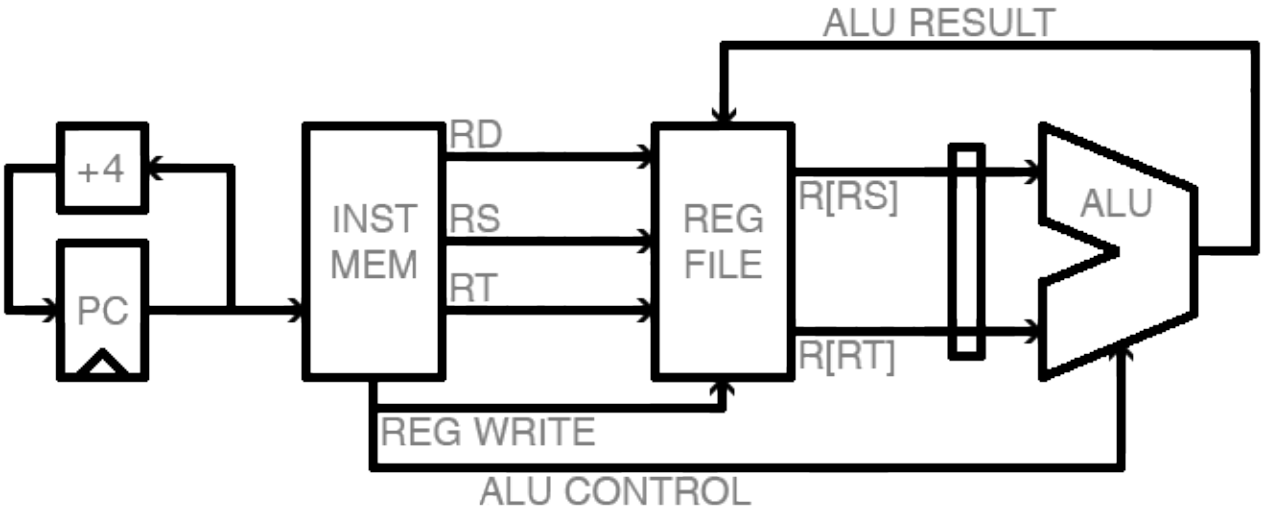


	New / Modified Component	New / Modified Control Signal
1		
2		
3		
4		
5		

Fill out the control signals for each of these three instructions. List an explicit value for boolean values (0 or 1), list an X for don't cares, or list an intuitive name. For example: if you added a MUX, list the meaning of the selected input instead of the numeric value of the selector. Include all new control signals you have added.

	RegDst	RegWr	ALUSrc	ALUCtr	MemWr	MemToReg				
rn										
lbn										
sbn										

Suppose we have the following simplified datapath. Note the removal of the data memory section. Also note the removal of all types of non-sequential instructions. We would like to pipeline this datapath into 4 out of the 5 standard MIPS pipeline stages. Ignoring any hazards, draw a box or circle over each wire to indicate putting a register at that spot. You can draw a single box over multiple wires to indicate putting a register on each wire. Right now, there already is a register on both wires leaving the register file.



F3: Virtually all about VM

For the following question, assume that the machine in question has the following parameters:

- 64-bit virtual addresses
- 48 bit physical addresses
- 4KiB pages
- Fully associative TLB with 128 entries and LRU replacement
- Unlimited swap space

(a) How much physical memory can the machine support at most? _____

(b) What is the maximum number of pages that a process can use? _____

(c) What is the minimum number of bits required for the page table base register? _____

(d) Suppose that you have a program that uses 1 KiB of contiguous, page aligned memory. at most, how many instances of this process can run at the same time? Assume that the operating system and page tables (including the page tables used for this program) use 1 GiB of memory and that no other processes are running. _____

(e) What if this process now uses 6KiB of contiguous, page-aligned memory? At most, how many instances of this process can run at the same time (same assumptions as in (d))? _____

Consider the following program. Assume that each of the functions fit within one page, that the stack, static, code, and heap segments all begin on page boundaries, and that no page faults occur in library calls.

```
#include <stdlib.h> //assume static linking
#include <string.h> //assume static linking
int mystrcmp (const char* const a, const char* const b, const size_t n) {
    for (int i = 0; i < n; i++)
        if (a[i] != b[i])
            return (a[i] - b[i]);
    return 0;
}
int main(int argc, char **argv) {
    const char *first = "1234567890";
    const char *second = malloc((strlen(first) + 1) * sizeof(char));
    strcpy(first,second); //assume no page faults on this function
    some_function_call(); //assume no page faults on this function
    mystrcmp(first, second);
}
```

(f) How many page faults occur in the best case scenario? _____

(g) How many page faults occur in the worst case scenario? _____

(h) For implementing software to use a low-bandwidth, high-latency network connection, would interrupt-driven I/O or polling be preferred? Correct answers without explanations will receive no credit.

(i) This subquestion involves T/F questions. Remember that incorrect answers on T/F questions are penalized with negative credit.

T / F – ECC provides protection from disk failures

T / F – A single parity bit allows us to detect any bit errors we have, but we need Hamming ECC (or something similar) to correct them.

T / F – RAID was originally designed to improve performance

T / F – All RAID configurations improve performance

T / F – All RAID configurations improve reliability

T / F – MapReduce is intended to run on a single, multi-core machine

T / F – As discussed in class, MapReduce jobs can survive if any node fails

F4: Think this question's hard? Don't worry, it's parallel, so it should be over quickly.

Consider each of the following code segments and determine which of the following statements is true about the correctness/performance of the given code, when run on the hive machines:

I. Always Incorrect

II. Sometimes Incorrect

III. Always Correct, slower than serial

IV. Always Correct, faster than serial

You must also provide a *brief* explanation (1-2 short sentences) detailing why you chose the answer you chose – unjustified solutions will receive no credit. You may assume that the code segments are correct when run serially.

(a)
```
#pragma omp parallel for
for (i = 0; i < 512; i++){
    for (j = 0; j < 512; j++){
        A[i + j*512] = i + j*512;
    }
}
```

(b)
```
for (i = 0; i < 512; i++){
    #pragma omp parallel for
    for (j = 0; j < 512; j++){
        A[i + j*512] = i + j*512;
    }
}
```

(c)
```
#pragma omp parallel
for (int i = 1, *lA = A; i < 512*512; i++, lA++){
    *lA = i + lA[-1];
}
```

(d) Lagrange interpolation is a useful technique for fitting a degree n polynomial to points $(x_0, y_0), \ldots, (x_n, y_n)$ by summing together $n+1$ different Lagrange polynomials, with each different Lagrange polynomial having the form

$$L_k(x) = y_k \frac{(x - x_0) \cdots (x - x_{k-1})(x - x_{k+1}) \cdots (x - x_n)}{(x_k - x_0) \cdots (x_k - x_{k-1})(x_k - x_{k+1}) \cdots (x_k - x_n)}$$

Starting from the following naive function:

```
/** Evaluates the Kth Lagrange polynomial generated by the N different
 *  inputs (X[0],Y[0]),(X[1],Y[1]),...,(X[N-1],Y[N-1]) at C. */
float eval_lagrange(float *X, float *Y, float c, size_t n, size_t k) {
    float retval = 1;
    for (size_t i = 0; i < n; i += 1) {
        if (i == k)
            continue;
        retval *= c - X[i];
        retval /= X[k] - X[i];
    }
    return retval * Y[k];
}
```

Complete the following SIMD-ized version of the function, optimizing for performance, and assuming that `n` is a multiple of 4.

```
float eval_lagrange_fast(float *X, float *Y, float c, size_t n, size_t k) {
    float retval = 1, m[4];
    size_t i;
    __m128 ret_vec = _mm_set1_ps(1);

    for (i = 0; i < ____________________; i += 1) {

        if (____________________)
            continue;
        ret_vec = _mm_mul_ps(ret_vec, _mm_sub_ps(_mm_set1_ps(c),

                                                 _mm_loadu_ps(____________)));

        ret_vec = _mm_div_ps(ret_vec, _mm_sub_ps(_____________________________,

                                                 _mm_loadu_ps(____________)));
    }

    for (____________________; i < ____________________; i += 1) {
        if (i == k)
            continue;
        retval *= c - X[i];
        retval /= X[k] - X[i];
    }

    ____________________________________________________________;

    return ____________________________________________________;
}
```

F5: What does *this* have to do with CS?

"There are three kinds of lies: lies, damned lies, and statistics." – ____________________