University of California, Berkeley – College of Engineering

Department of Electrical Engineering and Computer Sciences

Spring 2013 Instructor: Dan Garcia 2013-05-14

☹ CS61C FINAL ☺

After the exam, indicate on the line above where you fall in the emotion spectrum between "sad" & "smiley"...

Last Name	Answer Key
First Name	
Student ID Number	
Login	cs61c-
Login First Letter (please circle)	a b c d e f g h i j k m o p
Login Second Letter (please circle)	a b c d e f g h i j k l m n o p q r s t u v w x y z
*The name of your **SECTION** TA (please circle)*	Justin Alan Paul Sagar Sung-Roa Zachary
Name of the person to your Left	
Name of the person to your Right	
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. ***(please sign)***	

Instructions (Read Me!)

- This booklet contains 9 numbered pages including the cover page. Put all answers on these pages; don't hand in any stray pieces of paper.
- Please turn off all pagers, cell phones & beepers. Remove all hats & headphones. Place your backpacks, laptops and jackets at the front. Sit in *every other* seat. Nothing may be placed in the "no fly zone" spare seat/desk between students.
- You have 180 minutes to complete this exam. The exam is closed book, no computers, PDAs or calculators. You may use two pages (US Letter, front and back) of notes and the green sheet.
- There may be partial credit for incomplete answers; write as much of the solution as you can. We will deduct points if your solution is far more complicated than necessary. When we provide a blank, please fit your answer within the space provided. "IEC format" refers to the mebi, tebi, etc prefixes.
- **You must complete ALL THE QUESTIONS, regardless of your score on the midterm.** Clobbering only works from the Final to the Midterm, not vice versa. You have 3 hours... relax.

Question	**M1**	**M2**	**M3**	**Ms**	**F1**	**F2**	**F3**	**F4**	**Fs**	**Total**
Minutes	20	20	20	60	30	30	30	30	120	180
Points	**10**	**10**	**10**	**30**	**22**	**23**	**22**	**23**	**90**	**120**
Score	10	10	10	30	22	23	22	23	90	120

M1) ***Hacker's Delight*** (10 pts, 20 min)

We present **mystery**, a new helper routine for your C programming.
In parts (a) and (b), you'll show us you know how to use **mystery** from C.
In parts (c) and (d), you'll show us you understand its limitations.

```
mystery:  ori   $v0 $0 0x0
          beq   $a0 $0 done
          la    $t0 mystery
          lw    $t1 0($t0)
          addiu $t1 $t1 0x1
          sw    $t1 0($t0)
          lw    $v0 0($a0)
done:     jr    $ra
```

```
main() {
    char A[4], char4 = '4';
    int pi[] = { 3, 1, 4, 1, 5, 9 }
    float float4 = 4;
           // part (a)
    …      // more code, function calls, etc.
}
```

a) If you're at "`part (a)`" in the C code, show a single call to **mystery** so that it returns 4.

&pi[2] // or (pi+2)

printf("Here is mystery returning four … %d\n", mystery(__________));

b) Complete the documentation of **mystery** for a fellow programmer. Use good abstraction – don't tell the user *how* it does what it does, just tell them *what* it does and *how* it's to be used.

"When called with a *non-NULL* argument **arg**, the subroutine **mystery** …

increments its internal counter and returns the data at arg's memory address

____".

"When called with a *NULL* argument, the subroutine **mystery** …

returns the # of times it's been called with a non-NULL address (i.e., the # of mystery-assisted memory reads)

____".

"Overall, **mystery** is a subroutine used to …

record and report the number of memory reads

____".

c) We'd like to know if there is a limit to the # of times **mystery** can be called with a *NULL* argument (so that it still does what you described in part b). If there *is*, state what the limit is and what happens if it's called one more time. If there *isn't* a limit, write N/A (not applicable) in both blanks.

"With a NULL argument, **mystery** may be called at most **N/A** times. Calling it once more…

N/A

____".

d) We'd like to know if there is a limit to the # of times **mystery** can be called with a non-*NULL* argument (so that it still does what you described in part b). If there *is*, state what the limit is and what happens if it's called one more time. If there *isn't* a limit, write N/A in both blanks.

"With a non-NULL argument, **mystery** may be called at most **$2^{16}-1$** times. Calling it once more…

changes mystery so that it resets its counter to 0 but now puts the return value into v1 instead of v0, so you couldn't use mystery anymore since v0 would be garbage.

____".

M2) ***Cache Money, y'all*** (10 pts, 20 min)

This C code runs on a 32-bit MIPS machine with 4 GiB of memory and a single L1 cache. Vectors **A,B** live in different places of memory, are of equal size (**n** is a power of 2 and a [natural number] multiple of the cache size), block aligned. The size of the cache is C, a power of 2 (and always bigger than the block size, obviously).

```
// sizeof(uint8_t) = 1
SwapLeft(uint8_t *A, uint8_t *B, int n) {
   uint8_t tmp;
   for (int i = 0; i < n; i++) {
      tmp = A[i];
      A[i] = B[i];
      B[i] = tmp;
   }
}
```
	```
	// sizeof(uint8_t) = 1
	SwapRight(uint8_t *A, uint8_t *B, int n) {
	   uint8_t tmpA, tmpB;
	   for (int i = 0; i < n; i++) {
	      tmpA = A[i];
	      ______________________
	      tmpB = B[i];
	      ______________________
	      B[i] = tmpA;
	      ______________________
	      A[i] = tmpB;
	      ______________________
	   }
	}
	```

Let's first just consider the **SwapLeft** code for parts (a) and (b).

a) If the cache is **direct mapped** and the *best* hit:miss ratio is "H:1", what is the block size in bytes? (H+1)/2 ______

b) What is the *worst* hit:miss *ratio*? 0 ______:______ 4n (actually, 0:any-non-zero is ok)

c) Fill in the code for **SwapRight** so that it does the same thing as **SwapLeft** but improves the (b) hit:miss ratio. You may not need all the blanks.

d) If the block size (in bytes) is *a*, what is the *worst* hit:miss ratio for **SwapRight**? 2a-1 ________:________ 2a+1

e) We next change the cache to be **2-way set-associative**, and let's go back to just considering **SwapLeft**. What is the **worst** hit:miss ratio for **SwapLeft** with the following replacement policies? The cache size is *C* (bytes), the block size is *a* (bytes), LRU = Least Recently Used, MRU = Most Recently Used.

LRU and an **empty cache**	**MRU** and a **full cache**
2a-1 : 1 ________:________	0 : 4n (0:any-non-zero is ok) ________:________

M3) ***What is that Funky Smell? Oh, it's just Potpourri…*** (10 pts, 20 mins)

a) How many non-negative `float`s are < 2 ? 2^{30} ____________ *(you must show your work above for credit)*

b) What's the biggest change to the PC as the result of a `jump` on a 32-bit MIPS system? **256 mebibytes** ____________________
(answer in IEC format, like 16 kibibytes or 128 gibibytes)

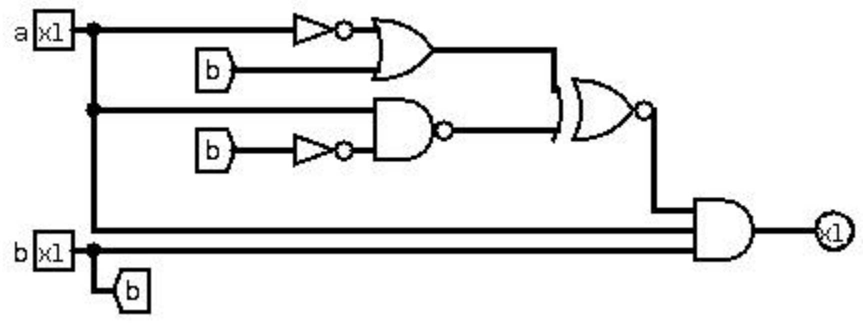
c) Fellow 61C student Ben Bitdiddle was told to write a function **`count_az`** that takes an input string of lower-case letters (only '`a`' through '`z`') and returns an array of the number of occurrences of all letters; a histogram if you will. The returned array will be zero-indexed and the indices will correspond to their respective order in the alphabet (i.e. `a = 0, b = 1, …, z = 25`). E.g., if the input `str` is "`baaadd`", the output array will look like the right column of the table on the right. Fix all the errors; we should be able to call it like this: `myAZ = count_az(str); yourAZ = count_az(str);`

`index`	`count[index]`
`0` (for 'a')	`3`
`1` (for 'b')	`1`
`2` (for 'c')	`0`
`3` (for 'd')	`2`
`…`	…
`25` (for 'z')	`0`

```
 1 int count_az(char *str) {
 2
 3     int count[26];                // Create the count array
 4
 5     while(*str) {                 // Go through the whole string
 6
 7         int index = &str – 0x97;  // The 97 is from the MIPS green sheet…
 8
 9         count[index]++;           // Increment the appropriate bucket
10
11         str++;                    // Go to the next character
12
13     }
14
15     free(str);                    // Free the string storage
16
17 }
```

Line #	Add Change Remove	Additions / Changes / Removals
1	**Change**	`int *count_az(char *str)`
3	**change**	`int *count = (int) malloc(sizeof(int)*26);`
4	**add**	`for ( int letter = 0; letter < 26; letter++ ) count[letter] = 0;`
7	**change**	`int index = ( (int) (*str) ) – 97;`
15	**change**	`return(count);`

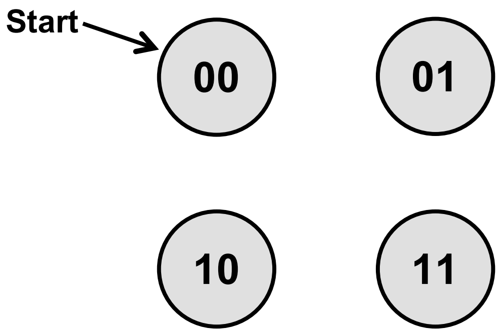
F1) Madonna revisited: *"We Are Living in a Digital World…"* (22 pts, 30 mins)



a) Rewrite the following circuit using the minimum number of AND, OR, and NOT gates: ________**ab**________
You must show your work above to earn points for this problem.

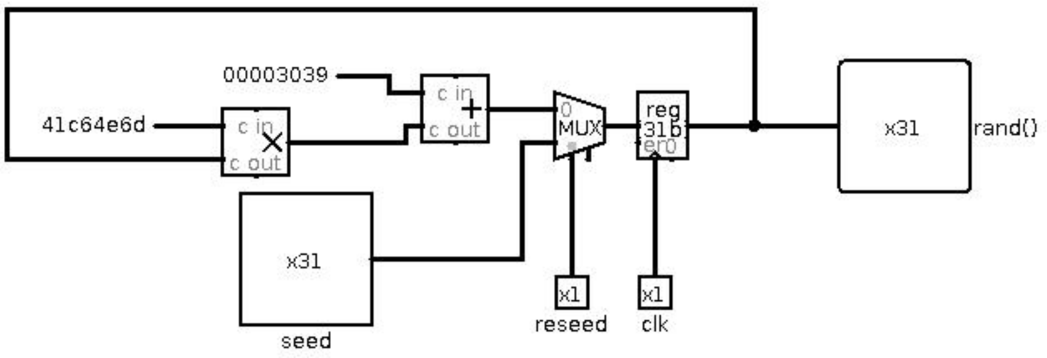
b) How many n-to-m logic gates (i.e. n bits of input, m bits of output) are there if we allow our circuit to produce outputs that are 0, 1, or high-Z? **$3^{(m*2^n)}$** ________________

c) You are an intern at a massive hardware firm, working on branch predictors. Start by predicting branch-not-taken and only switch to branch-taken after seeing TWO consecutive branches that are taken. Similarly, switch to branch-not-taken after seeing two consecutive branches that are not taken. Complete the FSM using as few states as possible (you may not need all of the provided states). Output a 1 for branch-taken and a 0 for branch-not-taken.



d) The following circuit outputs a new pseudo random number every cycle. You are responsible for selecting the cheapest (longest clock-to-q) register that will result in correct behavior when the circuit is clocked at 10 MHz. Assume that clock-to-q > hold time. **33 ns** ________________

- All available registers have a setup time of 2 ns
- Adders and multipliers have propagation delays of 30 ns
- Muxes have a propagation delay of 5 ns
- The signals seed and reseed are generated 4 ns after the rising edge of the clock



F2) *V(I/O)rtual Potpourri …* (23 pts, 30 mins)

For the following questions, assume the following:

- 32-bit virtual addresses
- 1 MiB pages
- 512 MiB of physical memory with LRU page replacement policy
- Fully associative TLB with 32 entries and an LRU replacement policy

a) How many entries does a page table contain? 2^{12} entries

b) How wide is the *page table base register*? 29 bits

```
int histogram[MAX_SCORE];
void update_hist(int *scores, int num_scores) {
    for (int i = 0; i < num_scores; i++)
           histogram[scores[i]] += 1;
}
```

Assume that only the code and the two arrays take up memory, ALL of code fits in 1 page, the arrays are page-aligned (start on page boundary), and this is the only process running.

c) If **update_hist** were called with **num_scores** = 10,
how many page faults can occur in the worst-case scenario? 11

d) In the best-case scenario, how many iterations of the loop can occur before a TLB miss?
You can leave your answer as a product of two numbers. $30*2^{18}$

e) For a particular data set, you know the scores are clustered around fifty different values, but you still observe a high number of TLB misses during `update_hist`. What pre-processing step could help reduce the number of TLB misses?

Sort the scores.

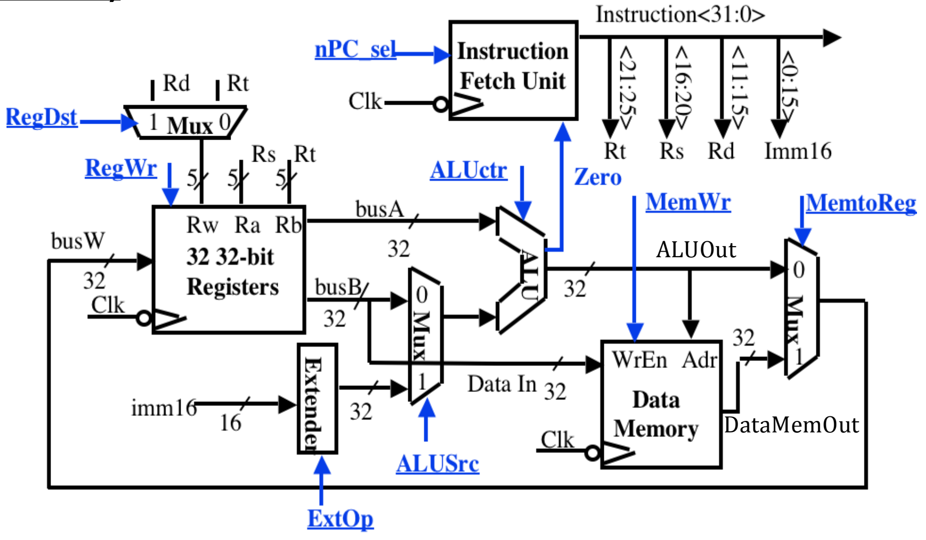
F3) *Datapathology* … (22 pts, 30 mins)

Consider the single cycle datapath as it relates to a new MIPS instruction, memory add:

```
madd rd, rs, rt
```

The instruction does the following:

1) Reads the value of *memory* at the address stored in `rs`.
2) Adds the value in the register specified by `rt` to the memory value and stores the resulting value in `rd`.



Ignore pipelining for parts (a)-(c).

a) Write the Register Transfer Language (RTL) corresponding to `madd rd, rs, rt`

R[rd] = Mem[R[rs]] + R[rt]; PC = PC + 4

b) Change as *little as possible* in the datapath above **(draw your changes right in the figure)** to enable `madd`. List all your changes below. Your modification may use muxes, wires, constants, and new control signals, but nothing else. (You may not need all the provided boxes.)

(i)	**Add a mux to select between busA and ALUOut for Adr into Data Mem.**
(ii)	**Add a mux to select between busA and DataMemOut for top input into ALU**
(iii)	

c) We now want to set all the control lines appropriately. List what each signal should be, either by an intuitive name or `{0, 1, "don't care"}`. Include any new control signals you added.

RegDst	RegWr	nPC_sel	ExtOp	ALUSrc	ALUctr	MemWr	MemtoReg	**DataAddr**	**ALUSrc2**	
1	**1**	**"+4"**	**X**	**0**	**"add"**	**0**	0	**"busA"**	**"Mem"**	

d) Briefly (one sentence) explain why `madd` CANNOT be run on the standard 5-stage MIPS pipeline.

A signal (DataMemOut) from the MEM stage is needed in the EX stage.

e) Let's fix our datapath so we can pipeline `madd`! If each stage currently takes 100 ps and an additional ADDER will have a delay of 50 ps, fill in the following table for two possible solutions: (1) a 6-stage pipeline and (2) a 5-stage pipeline with an extended MEM stage. Assume a memory access takes the full 100 ps. Leave your throughput answers as fractions.

	6-stage pipeline	Extended MEM
Latency:	**600 ps**	**750 ps**
Throughput:	**1/(100ps)**	**1/(150ps)**

F4) *What do you call two L's that go together?* (22 pts, 30 mins)

Suppose we have `int *A` that points to the head of an array of length `len`. Below are 3 different attempts to set each element to its index (i.e. `A[i]=i`) using OpenMP with `n>1` threads. Determine which statement (A)-(E) correctly describes the code execution and provide a one or two sentence justification.
Answers without a one or two sentence justification will receive NO credit.

a)
```
#pragma omp parallel for
for (int x = 0; x < len; x++){
      *A = x;
      A++;
}
```

A) Always **Incorrect**
B) Sometimes **Incorrect** ← **ANSWER**
C) Always **Correct**, Slower than Serial
D) Always **Correct**, Speed relative to Serial depends on Caching Scheme
E) Always **Correct**, Faster than Serial

Justification:

The for loop work is split across threads, but there is a data race to increment the pointer A. However, if the threads happen to complete work in disjoint time intervals and in-order, we may get the correct result.

b)
```
#pragma omp parallel
{
      for (int x = 0; x < len; x++){
            *(A+x) = x;
      }
}
```

A) Always **Incorrect**
B) Sometimes **Incorrect**
C) Always **Correct**, Slower than Serial ← **ANSWER**
D) Always **Correct**, Speed relative to Serial depends on Caching Scheme
E) Always **Correct**, Faster than Serial

Justification:

Our code computes the correct result but will be slower than the serial equivalent due to duplication of work.

c)
```
#pragma omp parallel
{
      for(int x = omp_get_thread_num(); x < len; x += omp_get_num_threads()) {
            A[x] = x;
      }
}
```

A) Always **Incorrect**
B) Sometimes **Incorrect**
C) Always **Correct**, Slower than Serial
D) Always **Correct**, Speed relative to Serial depends on Caching Scheme ← **ANSWER**
E) Always **Correct**, Faster than Serial

Justification:

Here we will encounter false sharing. Although our result will be correct, the speedup will depend on how well our caching scheme handles processors working closely in memory and the order of thread execution.

F4) *What do you call two L's that go together?* (Continued)

The moving average (a type of low pass filter) is an operation commonly used to smooth noisy data. Here we compute a *centered moving average* of width **WIDTH** on an array of data of size , where each element in our output array is the average of the *current* element, the *previous* **(WIDTH-1)/2** elements, and the *next* **(WIDTH-1)/2** elements. **Assume that WIDTH is odd for simplicity** and use zeros where "required" elements do not exist.

```
Example Input of        :     float[] A =      [ 7, 2, 3, 4, 8, 6 ]
Output for WIDTH=3:           float[] result = [ 3, 4, 3, 5, 6, 4.6666667 ]
```

Fit this problem to the MapReduce paradigm using a single map and reduce by filling in the blanks below. You may assume that you have access to the global variables **WIDTH** and **SIZE**. We expect you to use C syntax with the addition of a few java-like pseudocode elements (e.g. arrays have `.length`).

```
// receives data one element at a time
// Inputs: (key) is index i, (value) is A[i]
map(int key, float value){
     for (int x = key–floor(WIDTH/2); x < key+floor(WIDTH/2); x++) {

          context.write( x, value);

     }

}

// outputs elements of centered moving average
// Outputs MUST be of the form:
//    (key) is index i
//    (value) is moving average of width WIDTH centered at i
reduce(int key, float[] values){
     float total = 0;

     // do not emit keys that do not exist in output array
     if ( (key >= 0) && (key < SIZE) ) {
          for (int x = 0; x < values.length; x++)
               total += values[x];

          context.write(        key        ,   total/values.length    );

     }

}
```