University of California, Berkeley – College of Engineering

Department of Electrical Engineering and Computer Sciences

Spring 2016 Instructors: Vladimir Stojanovic, Nicholas Weaver 2016-05-09

☹ CS61C FINAL ☺

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

Last Name	Perfect
First Name	Peter
Student ID Number	
CS61C Login	cs61c-cs
*The name of your **SECTION** TA (please circle)*	Alex | Chris | Howard | Jack | Jason | Rebecca | Stephan | William
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 - *You should read this*:

- No electronics.
- Any paragraph that is formatted in *italics* is not important for doing the exam problems. You may skip reading them and still be able to finish the exam.
- There may be partial credit for incomplete answers; write as much of the solution as you can. When we provide a blank, please fit your answer within the space provided.

	MT1-1	MT1-2	MT1-3	MT1-4	MT2-1	MT2-2	MT2-3
Points Possible	6	16	16	12	6	15	12

	MT2-4	MT2-5	MT2-6	F-1	F-2	F-3	F-4	Total
Points Possible	11	4	3	10	10	15	9	145

This page is intentionally left blank

TOP SECRET

Hello Special Agent:

We have a special assignment for you today. This past weekend, your Superior Officers, Professor Vladimir Stojanovic and Dr. Nicholas Weaver, were reported missing from their offices. We suspect that some unknown entity kidnapped them and have imprisoned them, and it has come to our attention that we will need agents trained in some very specific skills in order to find out: 1. Who did this? 2. Where they are being held? and 3. How do we get them back?

As such, we, the CS61C staff, are requesting your help. This mission, if you choose to accept it, will test the limits of what you've learned this semester.

Good luck!

```
**************************
* TOP SECRET//CS//REL 61C *
**************************
```

LOGIN HERE: ___

MT1: Who Did It?

It looks like at the time of the kidnapping, the professors were working on some top secret material--looks like they were writing some questions for the final! A quick pass around their office reveals notes scattered around and questions written on a whiteboard. Perhaps when the professors realized they were about to be kidnapped, they managed to sneak in some clues about the identities of the abductors.

Stage 1: Finding the Bits and Pieces of Clues

One of the notes appears to be a hastily scratched series of numbers, various pictures of what looks like the C memory layout, and what appears to be the source code of a compiler. Since the kidnappers likely don't know the difference between a bit and a byte, this would be a great place for the professors to have hidden a message about the identities of their kidnappers. You however, being a 61C student, know exactly how to crack the code hidden in these questions.

a. What is 0x820 in decimal when read as an unsigned value?

Answer: **2080**

b. What is 0x820 in decimal when read as a signed two's complement value?

Answer: **-2016**

c. For the following statements, circle **T** for true and **F** for false.

T / [✓ **F**] The output of the compiler stage in CALL cannot contain pseudo-instructions. Pseudo-instruction gets expanded in assembler

[✓ **T**] / **F** The assembler takes two passes over the code in order to solve the "forward reference" problem.

T / [✓ **F**] The assembler takes two passes over the code in order to resolve absolute addresses. absolute addresses are resolved in linker. Remember your symbol table and relocation table

T / [✓ **F**] The linker outputs assembly code and takes object code as input.
Linker outputs machine code

After you solve the questions, you notice that while they all seem disconnected, their location in the room actually point to a bookcase in the office. Upon inspecting the bookcase a bit closer, you notice that one of the books is slightly ajar, so you push it back into place.

Bam! Suddenly a secret passages opens up, leading to another room. Tentatively, you enter, where you see a bunch of pictures that looked a bit flipped--perhaps they were transposed?

Stage 2: More than a Bit Flipped

The professors left us with some images, but we need to process them first in order to read them! After trying the basic transpose algorithm from lab, we realized that it needed to be slightly modified. Implement this new block transpose algorithm in order to continue the search for the professors.

```
/* Given a Matrix of square feature blocks that are
 * 2 by 2 stored as a serialized array:
 *                 [ A1,A2,B1,B2,E1,E2,
 *                   A3,A4,B3,B4,E3,E4,
 *                   C1,C2,D1,D2,F1,F2,
 *                   C3,C4,D3,D4,F3,F4 ]
 * The block transpose of this matrix would be:
 *                 [ A1,A2,C1,C2,
 *                   A3,A4,C3,C4,
 *                   B1,B2,D1,D2,
 *                   B3,B4,D3,D4,
 *                   E1,E2,F1,F2,
 *                   E3,E4,F3,F4 ]
 * blocks_r is the number of blocks in a row
 * blocks_c is the number of blocks in a column
 * b is the width/height of a block
 * dst is a pointer to the destination array
 * src is a pointer to the source array */

void block_transpose(int blocks_r, int blocks_c, int b, int *dst, int *src) {
      int width = blocks_c * b;
      int height = blocks_r * b;
      for (int i = 0; i < blocks_c; i++) {
            for (int j = 0; j < blocks_r; j++) {
                  int src_start = i*b + j*b*width;
                  int dst_start = j*b + i*b*height;
                  for (int y = 0; y < b; y++) {
                        for (int x = 0; x < b; x++) {
                              int dst_elem = y*height + x;
                              int src_elem = y*width + x;
                              dst[dst_start + dst_elem] = src[src_start + src_elem];
                        }
                  }
            }
      }
}
```

After you fill out the code, you realize there is a single file on the desktop called SUPER_SECRET_FILE.png. You write some code to load this image into a matrix, and you run it through your program. The output is an image, with just a single phrase on it: ***Not yes.***

Who could this possibly refer to? Before you have any time to ponder the complicated possible implications of this phrase, a popup appears on the computer. Some code appears, as well as an image somewhat resembling a tree. Perhaps this is a message from our Professors?

Stage 3: Held on Charge of Tree-son?

Looking at the screen, it seems like Dr. Weaver was writing code for the search method of a binary search tree, but he was whisked away before he could finish translating it to MIPS. Here is the original C function:

```
struct bst {
      const char* str;
      struct bst* left;
      struct bst* right;
};

int search(const char * check, struct bst** root)
{
      if (*root == NULL)
            return 0;

      char c, r;
      const char *ref = (*root)->str;
      for (int i = 0; (c = *(check+i)) !=0 | (r = *(ref+i)) != 0; i++) {
            if (c < r)
                  return search(check, &((*root)->left));
            if (c > r)
                  return search(check, &((*root)->right));
      }
      return 1;
}
```

```
***************************
* TOP SECRET//CS//REL 61C *
***************************
```

LOGIN HERE: ___

a. Complete the *search* function below in MIPS. $a0 holds the address of the string you want to check and $a1 holds the address of the root pointer of a well-formed binary search tree. Strings will **only** consist of lowercase letters with a null terminator at the end. Assume that uninstantiated pointers have a value of 0.

s0: *root, root pointer
t5: value of left pointer
t6: check pointer
t7: ref pointer

```
search:
      addiu $sp $sp -8          #prologue
      sw    $ra    0($sp)
      sw    $s0    4($sp)

      add   $v0 $0 $0
      lw    $s0    0($a1)
      beq   _____1._____
      lw    $t5    4($s0)
      add   $t6 $a0 $0
      lw    $t7    0($s0)
loop:
      lb    _____2._____
      lb    _____2._____
      ________3.________
      ________4.________
      slt   $t3 $t1 $t2
      bne   _____5._____
      bne   _____6._____
      bne   _____7._____
      li    $v0    1
      j     done
left:
      lw    $t5    4($s0)
      sw    $t5    0($a1)
      jal search
      _____8._____
right:
      _____9._____
      sw    $t5    0($a1)
      jal search
done:
      _______10.________        #epilogue
      lw    $ra     0($sp)
      lw    $s0     4($sp)
      addiu $sp $sp 8
      jr    $ra
```

t1: c
t2: r
t3: 1 if c &lt; r

for sw t5,0(a1), see explanation below

The most tricky part here is the content in a1. Instead of storing the pointer value directly in a1, this question stores the pointer value in a memory location and stores the address of that memory location in a1. Hence you can see the double pointer in the function. In assembly, this means that you have to lw from a1 to obtain the root pointer value and when make function calls, you have to sw to a1 to set new pointer values.

Multiple Choice: Circle answer

1. A. $a1 $0 done
   B. $v0 $0 done
   [✓ **C. $s0 $0 done**]

corresponds to (*root == null)

2. [✓ **A. $t1 0($t6) , $t2, 0($t7)**]
   B. $a0 0($t1) , $a1, 0($t2)
   C. $t1 0($a0) , $t2, 0($s0)

3. A. sub $t1 $t1 $t2
   B. addiu $a0 $a0 1
   [✓ **C. addiu $t6 $t6 1**]

advance both check and ref pointers

4. [✓ **A. addiu $t7 $t7 1**]
   B. add $t2 $0 $0
   C. addiu $s0 $s0 1

5. A. $t1 $0 left
   [✓ **B. $t3 $0 left**]
   C. $t5 $0 left

6. A. $t2 $t0 right
   [✓ **B. $t1 $t2 right**]
   C. $t5 $0 right

7. [✓ **A. $t1 $0 loop**]
   B. $t1 $0 done
   C. $t1 $t2 loop

if executed here, that means t2 (r) is 0, we will loop only if t1 is not 0

8. A. beq $v0 $0 done
   [✓ **B. j done**]
   C. blank

After we return from the recursive call, goes to finishing part straightaway

9. A. add $t5 $t5 8
   B. add $t5 $s0 8
   [✓ **C. lw $t5 8($s0)**]

set t5 to the value of right pointer

10. A. li $v0 $t3
    [✓ **B. sw $s0 0($a1)**]
    C. blank

b. *Before the evil villains whisked Dr. Weaver away, he asked them if he could leave a goodbye note. They accepted his request.* Weaver left this quote behind: “parting is such sweet sorrow”

Imagine $a1 originally began at 0x4000 and $s0 had an initial value of 4. If we search for the word “sorrow”, fill in what the stack frame would look like right before our program discovers that it found “sorrow”, using letters **a - h**. The binary search tree of the quote looks like this:

```
             such
           /      \
     parting       sweet
      /   \
    is     sorrow
```

a. Address of first line of prologue
b. Address of first line of epilogue
c. Address of “such” struct
d. Address of “parting” struct
e. Address of “sorrow” struct
f. Address of last line in loop label
g. Address of last line in left label
h. Address of last line in right label

4	
0x4000	
C	content of s0 after the first recursive call
G	content of ra after the first recursive call, since we are going to left after "such"
D	content of s0 after the second recursive call
B	content of ra after the second recursive call, since we are going to right after "parting"

c. *Dr. Weaver was not going to go out without a fight. After an epic struggle, the villains won and, to spite our valiant professor,* **changed all of the jal instructions to j**, *falsely believing that every call to search will now cause the stack to never be cleared after the function is completed. Weaver smirked in triumph because he knew that* **there is exactly one word that will not leave stack frames in the stack after you have finished searching for it. What is this word?**

such you won't use stack if you dont make recursive calls

Stage 4: What a MIPS-terious question...

The professors have left some MIPS code for you to analyze! Perhaps the final clue to the identities of the kidnappers is hidden in this question:

```
1          add $t0, $a0, $0     t0 now is a pointer to the
2          add $t1, $a1, $0     string
3                               t1 holds 3
4          addi $t3, $0, 4      t3 = 4
5          addi $s0, $0, 1      s0 = 1
6          addi $s1, $0, 0      s1 = 0
7
8    L1:   beq $t1, $0, Done    this is enough of a hint to tell you t1 is some
9          lb $t2, 0($t0)       counter
                                t2 is the character we are dealing with
10         add $t2, $t2, $t3
11   L2:   sb $t2, 0($t0)
12         subi $t1, $t1, 1     decrement counter and
13         addi $t0, $t0, 1     advance pointer
14         j L1
15
16   L3:   beq $s0, $0, Done
17         addi $s0, $0, 0
18
19         la $t0, L2           these two lines load the instruction at line 10
20         lw $t1, -4($t0)      into register, remember everything is just data
21         addi $t1, $t1, 407552   #407552 = 0x00063800
22         sw $t1 -4($t0)
23
24         add $t0, $a0, $0  #Resets $t0 to the original $a0
25         add $t1, $a1, $0  #Resets $t1 to the original $a1
26
27         j L1
28
29   Done:
30         move $s0, $a0
```

a. Write the MIPS instruction format representation of line 10 in binary; use the provided box below. Each box holds 4 bits.

0000	**0001**	**0100**	**1011**	**0101**	**0000**	**0010**	**0000**

b. Say that $a0 is a pointer to a null terminated string "abc", and that $a1 is set to 3. After this program runs, what is the value of the string that $s0 points to?

"efg"

You realize that someone has left a note in the corner of the desk; it seems to say that there is a typo in line 8; the line should read:

L1: beq $t1, $0, L3

For the last 4 parts of this question, use this new value for line 8

c. When you run this code using the same inputs as in part b, what is the value of $t1 right after line 21 executes? Write this 32 bit value in binary in the box below; each box contains 4 bits.

0000	0001	0101	0001	1000	1000	0010	0000

"add $t2, $t2, $t3" plus 0x00063800

0x00063800 + 0x014b5020 = 0x1518820

mind-blown? if not, read the question more carefully

d. In a sentence or two, what does this function do?
Hint: If you're stuck, trying running the program with $a0 pointing to the null terminated string "cde"

add the ascii values of the characters into $s1

the original instruction gets modified to add $s1 $t2 $s1

*You notice that there is another piece of paper with a string written on it. Thinking that this string is possibly the final clue you need, you pass this string through the program, and you get the output "**Agency**".*

Thinking this sounded familiar, you take the clues you gathered from stages 2-4, and it hits you--the identity of the kidnappers is (no points):

No Such Agency

MT2: Location -> [REDACTED]

We now know who did it, but we still have no clue as to the whereabouts of our beloved professors! Luckily, the professors seemed to have some prior warning that this was going to happen, and they were writing a complex program that mashed together some natural language processing, geolocation, and maps searching algorithms that seems up for this task. Professor Stojanovic was designing some custom photonic chips that could handle running this code, but he had to leave it unfinished on his desk - it's up to you to fill in the gaps and finish the hardware.

Scrawled on a piece of paper next to his chip is a cryptic note: "Use your HARBOR to DOCK your BOAT ~ 46.158.42.111". You realize that someone had leaked a terabyte of secret agency transmissions to Wikileaks from the past month, so you decide to fire up your project 5 and quickly modify it to collect and match all transmissions associated with that IP address.

Stage 1: Fast String Matching

Looking through some of the outputted matches by hand, you realize that all of them have encrypted data of some sort. In addition, each message sent from the address starts with a prefix: ranging from "lvl0" to "lvl5". A few of them start with something special: "tsso" *(which everyone knows is spy speak for "Top Secret Special Operation")*. It seems like we want to look closely at packets that are either type "lvl5" or "tsso".

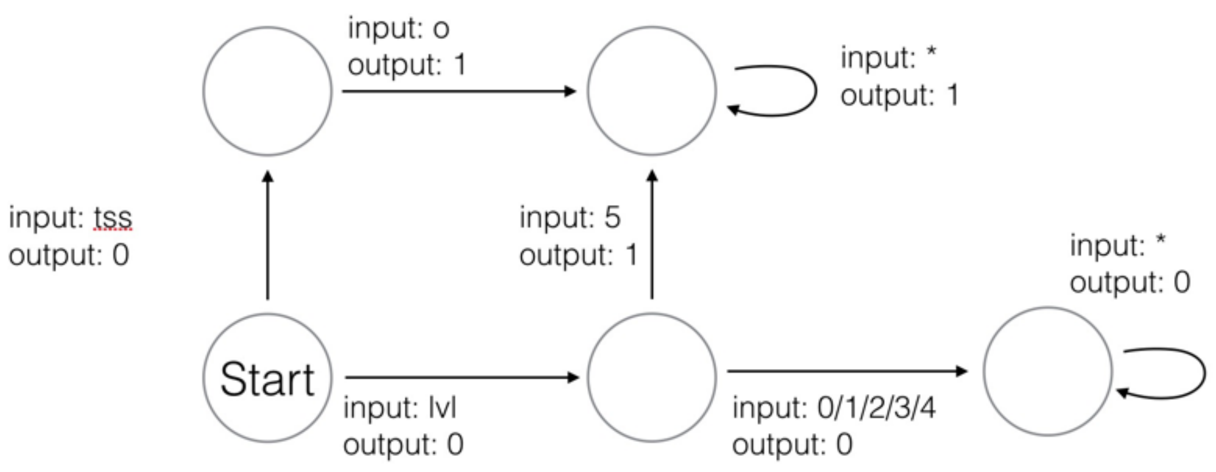
After taking stock of the situation, this is what you've gathered:

- You want to find messages that start with "tsso" or "lvl5", but not "lvl0" - "lvl4". Messages will not start with any other prefix.
- The professors left a program called "Distributed Organization of Common Keywords (DOCK)", which takes in messages and an FSM, and returns all messages where the FSM outputs a 1 at the end.
- DOCK restarts the FSM on each new message, and each message only consists of lowercase characters and numbers.

Notation note: If you want to match multiple characters consecutively, you may do this in one transition, but each character must output the same value. If there are multiple ways to get from one state to another, you may write them all on one arrow and separate them with a '/'. For example, if you want to match "abc", you would write "abc"; if you want to match "a", "b", or "c", you would write "a/b/c". If the input is '*', that means it matches anything that has not been matched.

The FSM is missing - please fill in the following FSM so that it outputs a 1 forever when it finds the prefix "tsso" or "lvl5", and outputs a 0 forever otherwise. The FSM should start **outputting a 1 at the "o" of "tsso" and the "5" of "lvl5".**

SOLUTION



It seems like all of the encrypted data is part of an image of some sort, transferred in pieces over the course of a month so the average snooper wouldn't know to put it together - but Project HARBOR was able to bring all of the messages into one place.

We've found HARBOR and DOCK, but what could BOAT be referring to? On a whim, you type ⛵ *into the computer search bar, and a splash screen starts up - this must be the program that the professors were working on. An error pops up though, and it reads "Please connect the completed Branching-Optimized Asynchronous Transformer and press the Any key". There is a device next to the I/O port that looks like it was made with the MIPS 5-stage pipeline, I guess there's more work to be done.*

Stage 2: If you don't know what to do next, guess?

Apparently, the program's algorithm will have many branches, so we want to minimize the number of cycles that branches take to execute. However, instead of a branch delay slot, one of your CS 61C tutors decided to design a BTB (Branch Target Buffer) and add it to the *Fetch* Stage of our standard 5-stage pipeline instead. **Ignore branch delay slots in this problem.**

Every entry in the BTB contains three things: (1) a PC value (2) a predicted "next" PC value and (3) a Valid bit. The BTB checks the current PC against (1): if the Valid bit (3) is set, the BTB sets the PC to the predicted "next" PC value (2) in the Fetch stage. Otherwise, PC will be set to PC+4 as usual. Our BTB has 4 sets, and is Fully Associative.

At the Execute phase of the branch, we find out whether or not the BTB was correct. If it was correct, we do nothing. If it wasn't correct, we flush all instructions incorrectly executed after the branch, and set the PC to the correct value. **For simplicity, we will not be updating the state of the BTB on a mis-prediction.** In reality, this is impractical.

a. How many bits is in our BTB, assuming that we only have a 1 bit Valid bit per entry for our **32-bit** machine? You can assume that PC and next PC are both 32-bits.

(2 * 32 + 1) * 4 = 260 bits

For the following parts, we will be dealing with a **16-bit** machine (address space), with **32-bit** registers. The first addi instruction starts at address `0x1000`. The "line numbers" below are offsets of this address.

```
+00|             addi $t7, $0, 5
+04| loop:       lw $t0, 0($s1)
+08|             andi $t1, $t0, 1
+0c|             bnez $t1, next
+10|             bgt $t0, $t7, A
+14|             addi $t0, $t0, 1
+18|             j B
+1c| A:          addi $t0, $t0, -1
+20| B:          sw $t0, 0($s1)
+24| next:       subi $s1, $s1, 4
+28|             bne $s1, $s0, loop
```

and the array value with 1 to check for parity

Here is the initial state of our BTB:

Valid	Current PC	Predicted Address
1	0x100c	0x1024
0	0x1028	0x1004
1	0x1010	0x101c

This is the array:

0x4000	0x4004	0x4008
0x6	0x1	0xC

`$s0` contains `0x4000`. `$s1` contains `0x4008`.

b. In no more than two sentences, describe what the final array will look like (in general, not just on this specific code example!)

all odd numbers unchanged; even numbers >5 decremented by 1; even numbers < 5 incremented by 1

c. Assume that we are running on the standard MIPS pipeline with the BTB. There is no forwarding, and branch prediction (the BTB) is in the Instruction Fetch stage. Let's see how our CPU performs:

i.
```
+08|          andi $t1, $t0, 1
+0c|          bnez $t1, next
```

There is (circle one of the following): **a data hazard**. no hazard.

If you said there was a hazard, how many stalls do we need? **2** stalls.

t1 will be ready after EX and WB, by which time the second instruction will be doing Decode

ii. How many more cycles would an incorrect branch prediction take than a correct one? **2** cycles.

The incorrect results will be revealed at the end of execution, by which time two cycles have elapsed.

iii.
```
+0c|          bnez $t1, next
```

# of correct branch predictions:	**1**
Average # of stalls per prediction:	**10/3**

iv.
```
+10|          bgt $t0, $t7, A
```

# of correct branch predictions:	**2**
Average # of stalls per prediction:	**0**

v.
```
+28|          bne $s1, $s0, loop
```

# of correct branch predictions:	**1**
Average # of stalls per prediction:	**10/3**

d. If our MIPS code is now self-modifying, what bit should we add to the BTB?

"dirty" bit

Stage 3: Time to Cache Out

Our hardware will have to work with many multidimensional arrays, so we'll need to make some implementation choices. Consider the 2-dimensional array `A` referencing 64*32 **doubles** (8 bytes), which you can assume is **double-word aligned**. In this problem we will be comparing the hit rate of a cache with two different ways of representing multidimensional arrays in C, as a 1-D array and a 2-D array of arrays:

// Version A	// Version B

`sum = 0;`
`for (i = 0; i < 64; i++)`
` for (j = 0; j < 32; j++)`
`   sum += A[i*32 + j]`
	`sum = 0;`
	`for (i = 0; i < 64; i++)`
	` for (j = 0; j < 32; j++)`
	`   sum += A[i][j]`

a. Given a 4KB **direct mapped** cache with 32-byte blocks, what are the best and worst case hit rates for version A?

A holds 2^11 doubles = 2^14 bytes. Fills cache 4 times
Worst case: cold cache. 4 floats/block means 3/4 hit rate

Best case: Cache filled w/ first 1/4 of the array, 100% hit. Next 3/4 of the array has a 75% hit rate. 1/4 + 3/4 * 3/4 = 13/16

Best Case: **13/16** Worst Case: **3/4**

b. Given the same cache as in part a, what are the best and worst case hit rates for version B?

For this question, we have 65 arrays. A is an array of 64 pointers, somewhere contiguous in memory. In addition, we have 64 arrays that hold data, scattered through memory, and holds 32 doubles contiguous in memory

In the worst case, each pointer maps to the same block as the first block of data:

```
P for pointer array. D for data array
P D
M M   M M    H M
M M + H H + (H H * 6)    16 misses/64 accesses
M M   H H    H H
M M   H H    H H

H M   M M    H M
M M + H H + (H H * 6)    15 misses/64 accesses
M M   H H    H H
M M   H H    H H
\______________/  pattern for the 7 pointers in the same block
        * 7
\_____________________/ pattern for each block of 8 pointers
           * 8
```

In the best case, all of the pointers are in the cache, and the rest holds the data that is first accessed. Data doesn't map to the same block as ptr.
Cache: 2^12 B – 2^8 B of pointers = 2^9 – 2^5 doubles in the cache - 100% hit
For the other 2^11-2^9+2^5 doubles, the access looks like the following.

```
P D
H M   The pointer accesses will always hit, as it is already loaded in the cache
H H   The data accesses will have a 3/4 hit rate, and we ensure that the data
H H   doesn't overwrite pointer blocks in the cache. 8 misses/64 accesses
H H
```

Best Case: **1-((1/4)(2^11-2^9+2^5))/2^12** Worst Case: **968/4096**

c. Given the same cache as in part a, what is the minimum number of **cache blocks** that we need in order to only have compulsory misses in version A and version B in the best case?

Version A will need the whole cache, as the array lies in contiguous memory and is larger than the cache

Version B needs the min between blocks needed for the pointers array and the blocks needed for one data array of 32 doubles. min(2^3,2^3)

Version A: **2^7**

Version B: **2^3**

d. Given a 4KB **fully associative** cache with 32-byte blocks, what are the best and worst case hit rates for version B?

The best case is the same as in part b

The worst case is one miss for every block, for the first time it is loaded into the cache. There are 64/8 = 8 blocks of pointers and 64*32/4 = 512 blocks of data. This gives a 520/2^12 miss rate

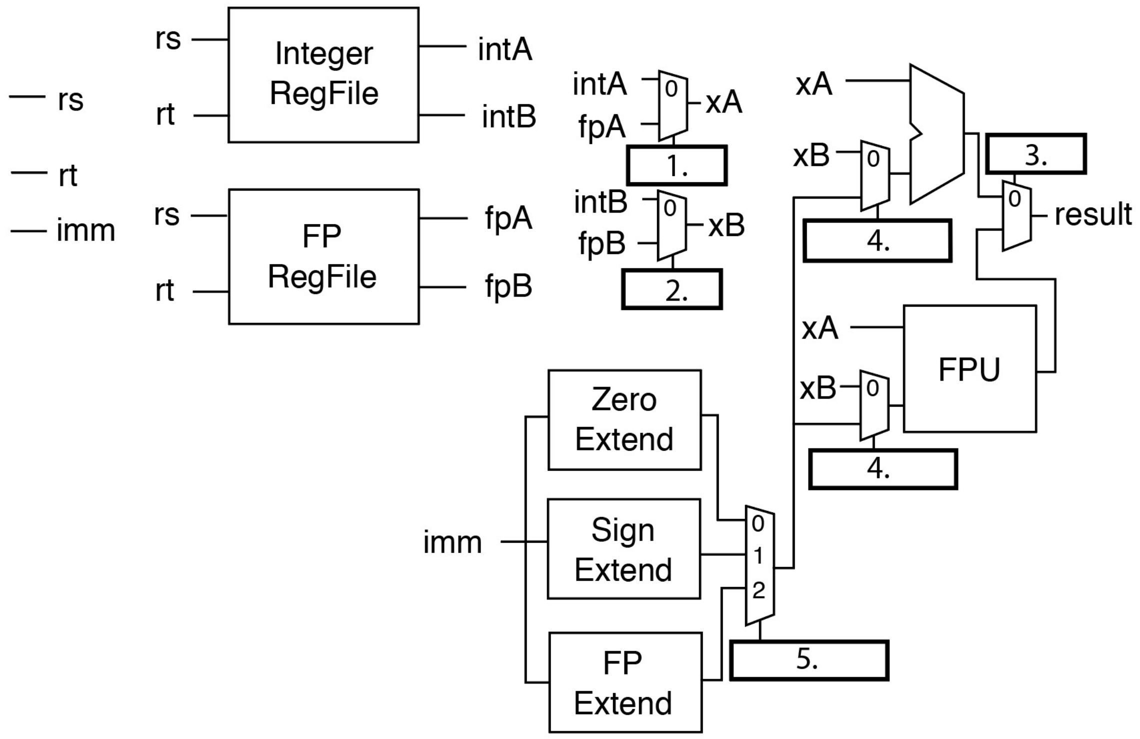
Best Case: **1-((1/4)(2^11-2^9+2^5))/2^12**

Worst Case: **1–(2^3+2^9)/2^12**

Stage 4: Stop! Datapath is Under Construction

We're almost done, but now we need the ability to do floating point calculations in order to compute coordinates. Unfortunately, the execute stage of the processor is not complete. The datapath is all set up, but the control signals are not connected. In the controller, the output pins for the control signals are there, but there's no logic driving them.

In the datapath diagram below, fill in the labels for each of the mux select inputs with one of the listed control signals. Note that there are now two register files: one for integers and one for floating point numbers. There are also two functional units: an integer ALU and floating-point unit (FPU). Then fill in the table with the value the control signals should take for each of the listed instructions, or **X if the signal doesn't matter**. Note that the input to the extender mux should be **two bits**. The specifications for the non-standard FPU instructions are given next to the table.



Write labels here:

1. float
2. float
3. float
4. use_imm
5. ext_sel

Instruction	float	ext_sel	use_imm
addu	0	X	0
addiu	0	1	1
or	0	X	0
ori	0	0	1
fadd	1	X	0
faddi	1	2	1

fadd: R-type instruction

```
FPRF[rd] ← FPRF[rs] + FPRF[rt]
```

Adds two single-precision numbers from the floating-point register file and write the result to the floating-point register file.

faddi: I-type instruction

```
FPRF[rt] ← FPRF[rs] + FPExt(imm)
```

Extend the half-precision immediate to a single-precision number. Add it to a single-precision number from the floating-point register file and write back to the floating-point register file.

You look at your handiwork, and you're quite impressed with all the fixes and additions you've made to the hardware. You try to flick up the power switch and the screen lights up - BOAT is up and operational!

Stage 5: This Question will Float Your BOAT

You start up the professor locating program, and it prints out two floating-point numbers, which are the latitude and longitude of where the professors are being held. The numbers are in IEEE half-precision (16 bit) format.

Convert the half-precision numbers provided below in binary format to their decimal floating point representation. Half-precision numbers have 5 bits for exponent and 10 bits for significand. The exponent bias is 15.

`0b 0 10100 0010000011` , `0b 1 10101 1110100100`

= 1.0010000011 * 2^(20-15)

= 100100.00011 = 36.09375

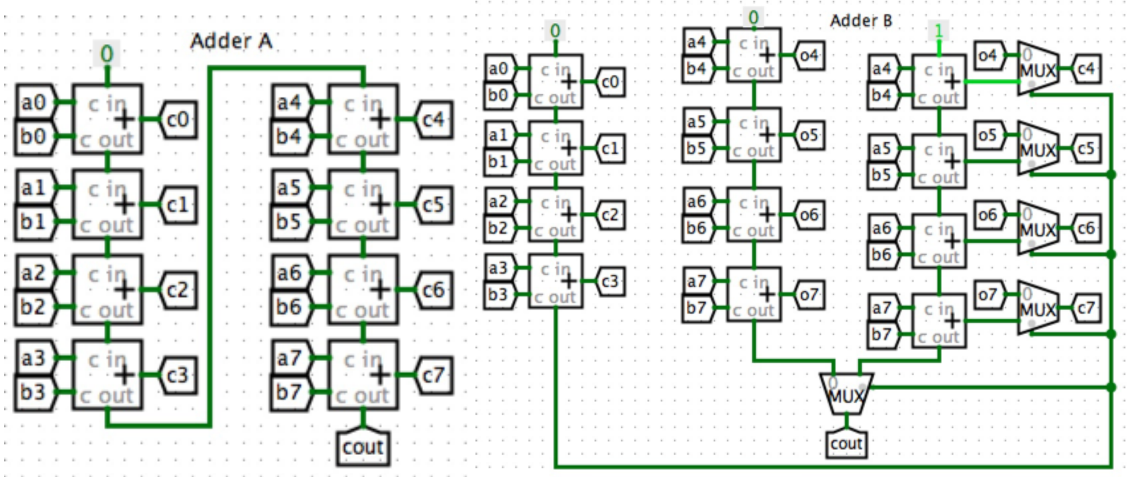
= 1.1110100100 * 2^(21-15)

= 1111010.0100 = -122.25

Success! Looking back at the program output, you see a footnote "Program running in TESTING mode, output only has 7% accuracy. Turn on PRODUCTION mode in settings." That's too bad, but not the end of the world. You change the mode in the settings and rerun the program, but, in dismay, you see that the estimated time remaining is 12 hours. You can't wait this long; there must be something else you can optimize!

Stage 6: Adding Adder Complexity

Luckily, someone left a sticky note on the screen saying, "TODO: test Adder A and Adder B, current adder is way too slow..."



It seems like Professor Stojanovic was debating the merits of two different 8-bit adder designs (each consisting of chained full-adder cells), which both take in A and B and output C (labels `c0`-`c7`). Add calculations in this case do not need to worry about overflow. You also know that the propagation delay for each MUX is approximately as long as **two full-adder delays.** You can assume that a full-adder delay from input to either output c_{out} or sum is the same.

a. Does **Adder A** calculate C correctly? [✓ **Yes**] No
If you selected "no", state a reason why in two sentences or fewer.
Normal Adder from class

b. Does **Adder B** calculate C correctly? [✓ **Yes**] No
If you selected "no", state a reason why in two sentences or fewer.
Ripple Carry Adder

c. Regardless of correctness, which runs in less time? **Groups of 4 bits done in parallel**
Adder A [✓ **Adder B**]

```
**************************
* TOP SECRET//CS//REL 61C *                                   LOGIN HERE: ___
**************************
```

Upon choosing the faster and correct adder design, you re-run the program - 11 hours remaining... it would have been too much to expect a 24x speedup just by changing the adder design. Poking around, you see a setting called "Location mode: Coarse (200 ft radius)". Might as well give it a shot, you think to yourself. After switching this mode on, the time remaining jumps down to 30 mins - makes you wonder how accurately the "Fine" mode would have worked. Regardless, you sit down and wait; after 30 minutes, a printer fires up and you read the following:

```
SEARCH COMPLETED: 29 min, 42 sec.
LOCATION MATCHED: EVANS HALL BASEMENT
```

Post-MT2: A Quick and Stealthy Extraction

As you head towards Evan's basement, you notice a screen along with a keypad on the door. You attempt to open the door, but it won't budge. Suddenly a question appears on the screen ... The only way through that door is by solving the question and entering the correct answers in the keypad.

Stage 1: Virtual Memory

For the following question, assume the following:

- 28-bit virtual addresses
- 16MiB Physical Memory with LRU replacement
- 4KiB Pages
- Fully associative TLB with 16 entries and an LRU replacement policy

a. What is the VPN:PO (Virtual Page Number: Page Offset) breakdown for VM? **16:12**

4KiB page, 12 bits offset, VPN has 28 - 12 = 16 bits

b. What is the PPN:PO (Physical Page Number:Page Offset) breakdown for Physical memory? **12:12**

physical MEM has 2^24 bytes, 24 - 12 = 12 bits

Consider the following code:

```
// Let src, dst be char*
// Assume rand is an array of integers initialized with random
// integers in [ 0, strlen(dst) - 1 ], without repeats
// max_replacements is an arbitrary number
for (i = 0; i < max_replacements; i++) {
      dst[rand[i]] = src[rand[i]]
}
```

Assume that only the code and the two `char*` take up memory, `strlen(dst) <= strlen(src)`, ALL of code fits in 1 page, TLB currently has a pointer to the code, the strings are page-aligned (starting on a page boundary), and this is the only process running. For simplicity, you can also assume `rand` does not exist in memory – you do not need an entry for the address translation of `rand`.

c. How many page faults would occur after 20 iterations in the ...

Best case: both dst and src are already in MEM, no extra page needed

Worst case: memory already filled, and src[rand[i]] and dst[rand[i]] both asking for memory locations that are not in phyiscal memory yet, creating 2 page faults on each iteration

Best Case: **0** Worst Case: **40**

d. How many iterations of the loop can occur before a TLB miss in the ...

Worst case: first page is out of bound

Best case: one entry reserved for code, so we have potentially 15 existing pages in TLB, and we can have 15 pages worth of references

Best Case: **15*2^12** Worst Case: **0**

After entering in your solutions, the keypad flashes green and the door slides open. "That was weird", you think to yourself, "what kind of secret agency uses a 61C final question for their security prompt?" As you creep along a hallway in the Evans basement, you suddenly hear footsteps behind you! You open the door to the nearest office and duck inside.

It's only after you've closed the door that you realize what was written on it: "OpenMP Consultant." Even worse, you hear someone approach the door and knock. You quickly flip on the lights, sit behind the desk, and try to appear casual. A No Such Agency agent enters and asks for help with OpenMP coding. It looks like you'll have to help her analyze it to avoid arousing any suspicions.

Stage 2: The OpenMP Doctor Is In

The No Such Agency agent shows you the following code that she's trying to parallelize. You feel a sense of déjà vu as you look over it.

```
   typedef struct {                            typedef struct {
         char *src_ip;                               char *src_ip;
         int src_port;                               int src_port;
         char *dest_ip;                              char *dest_ip;
         int dest_port;                              int dest_port;
         char *cookie;                               char *user_name;
         long timestamp;                             long timestamp;
   } request_t;                                } reply_t;

int tuple_matches(request_t *request, reply_t *reply) {
    return strcmp(request->src_ip, reply->src_ip) == 0 &&
                 request->src_port == reply->src_port &&
                 strcmp(request->dest_ip, reply->dest_ip) == 0 &&
                 request->dest_port == reply->dest_port;
}

// Assume replies and requests are valid arrays
request_t requests[num_requests];
reply_t replies[num_replies];
int num_matches = 0;
for (int i = 0; i < num_requests; i++) {
    request_t *request = requests + i;
    for (int j = 0; j < num_replies; j++) {
        reply_t *reply = replies + j;
        if (tuple_matches(request, reply) &&
                      abs(reply->timestamp – request->timestamp) <= 10) {
            num_matches++;
        }
    }
}
```

Version 1

```
int num_matches = 0;
#pragma omp parallel
{
    for (int i = 0; i < num_requests; i++) {
        request_t *request = requests + i;
        for (int j = 0; j < num_replies; j++) {
            reply_t *reply = replies + j;
            if (tuple_matches(request, reply) &&
                        abs(reply->timestamp - request->timestamp) <= 10) {
                num_matches++;
            }
        }
    }
}
```

Will this code produce the correct answer? Assume there is always a non-zero number of matches, and circle the best response.

Always Correct [✓ **Sometimes Correct**] **Never Correct**

Data race on num_matches

Will this code be faster than the serial version? Circle your response.

Faster than Serial [✓ **Not Faster than Serial**]

if you do not have omp parallel for, you are running the enclosed code in entirety in multiple threads, i.e. repeated work

Version 2

```
int num_matches = 0;
#pragma omp parallel for
for (int i = 0; i < num_requests; i++) {
    request_t *request = requests + i;
    for (int j = 0; j < num_replies; j++) {
        reply_t *reply = replies + j;
        if (tuple_matches(request, reply) &&
                    abs(reply->timestamp - request->timestamp) <= 10) {
            num_matches++;
        }
    }
}
```

Will this code produce the correct answer? Assume there is always a non-zero number of matches, and circle the best response.

Always Correct [✓ **Sometimes Correct**] **Never Correct**

data race still exists

Will this code be faster than the serial version? Circle your response.

[✓ **Faster than Serial**] **Not Faster than Serial**

Version 3

```
    int num_matches = 0;
    for (int i = 0; i < num_requests; i++) {
        request_t *request = requests + i;
        #pragma omp parallel for
        for (int j = 0; j < num_replies; j++) {
            reply_t *reply = replies + j;
            if (tuple_matches(request, reply) &&
                        abs(reply->timestamp – request->timestamp) <= 10) {
                num_matches++;
            }
        }
    }
```

Will this code produce the correct answer? Assume there is always a non-zero number of matches, and circle the best response.

Always Correct [✓ **Sometimes Correct**] **Never Correct**

Data race still exists

Will this code be faster than the serial version? Circle your response.

[✓ **Faster than Serial**] **Not Faster than Serial**

Version 4

```
    for (int i = 0; i < num_requests; i++) {
        request_t *request = requests + i;
        #pragma omp parallel for reduction(+:num_matches)
        for (int j = 0; j < num_replies; j++) {
            reply_t *reply = replies + j;
            if (tuple_matches(request, reply) &&
                        abs(reply->timestamp – request->timestamp) <= 10) {
                num_matches++;
            }
        }
    }
```

Will this code produce the correct answer? Assume there is always a non-zero number of matches, and circle the best response.

[✓ **Always Correct**] **Sometimes Correct** **Never Correct**

Will this code be faster than the serial version? Circle your response.

[✓ **Faster than Serial**] **Not Faster than Serial**

Happy with your help, the agent leaves the room - disaster averted. You look at stage 3 of your plans, and all it says is "???". Before you can start complaining about bureaucracy and vague instructions, your phone buzzes.

Stage 3: Mean Time To Final

a. Agent Jaime Bont is trying to send you vital intelligence, but her messages keep getting dropped and garbled. If she wanted to guarantee delivery of her data, what transport layer protocol should she have used? **TCP** also known as reliable transport, TCP has features to resend dropped packets

b. Agent Bont sends a possibly corrupted 8-bit message, 0xFA, that we know uses Hamming ECC format from class, class, along with a right-most bit as a parity over the entire message.

i. Circle all of the following properties that this format ensures:

	Double Error Detection	Triple Error Detection
Single Error Correction	Double Error Correction	Triple Error Correction
	None of the above	

ii. State the Hamming distance between the received message and the correct message, and give the corrected message if necessary and possible.

	p1	**p2**	d1	**p4**	d2	d3	d4	**p8**
	1	1	1	1	1	0	1	0
p1	X		X		X		X	
p2		X	X			X	X	
p4				X	X	X	X	

Hamming distance: **1** Corrected message: **0b1111** or **Impossible**

c. It turns out that Agent Bont was trying to send us specs on the datacenter.

i. Bont reports that their disk drives have no queuing delay, a SCSI interface controller with 1 ms delay, a rotation speed of 12,000 rpm, a transfer rate of 20 MB/s, a total of 16,000 cylinders, and an arm speed of 4,000 cylinders per 3 ms. On average, how long does it take to read or write 16 KB of data? Show work for potential partial credit.

1 + 2.5 (rotate) + 0.8 (transfer) + 4 (seek) = 8.3 ms total

ii. Does RAID 1 offer concurrent independent writes? **Yes** **No**
Does RAID 5? **Yes** **No**

iii. There are 8760 servers in this particular datacenter, each with 4 disks having a 0.1% annual failure rate. What's the mean time to failure, in hours? Show work for potential partial credit.

total number of failures is 8760 servers * 4 disks * 0.001 = 35.04

250 hours

No Such Agency is really understaffed, as they assigned only one technician to monitor both their datacenter and the professors' captivity. Aside from donuts and Sword Art Online reruns, the tech is also distracted by repairing disk failures.

iv. Assuming NSA holds itself to three nines of availability with MTTF from part iv), what's the mean time we have to rescue the professors upon a disk failure, to the closest minute? Show work for potential partial credit.

~15 minutes

All of the pieces are now falling into place, and after a few more steps down the corridor, you've finally reached the door to Vladimir and Nick's cell in the Evans basement! Unfortunately, the door is secured by a keypad. You'll have to enter the correct sequence of letters ("A" through "E") to open the door, and there's no way you can deduce the code, or even its length.

As you hang your head in despair, you notice something on the floor. It's an old CS 61C discussion worksheet. There's a single word hastily written on the back of the sheet: "Clickers." It seems like a long shot, but maybe Nick anticipated his abduction at the beginning of the term. Could he have made it so that each letter of the code to open the door corresponded to the most popular clicker answer in each lecture, including all of those convergences on the wrong answer?

Stage 4: Breaking Down the Door

It's worth a try. You fire up your laptop (like any good CS student, you bring your laptop along with you on a rescue mission) and effortlessly hack into the bCourses database to get the 61C clicker response data. It contains a file with everyone's clicker responses over the semester. Each line of the file is in the following format:

```
student_id lecture_number response
```

a. You decide to write a Spark job and run on it on your super secret cluster to parallelize the analysis of this file, so you can get your answer as quickly as possible. Assume that the job would require 30 minutes to run on a single machine and that 90% of the job's work can be parallelized.

At least how many machines do you need to use in your Spark job so that the job will only require 6 minutes? Assume that the parallel portion of the work exhibits perfect scaling, i.e. running it on *x* machines will take 1/*x* as much time to finish.

This is really just Amdahl's Law in reverse.
10% of the job's work is sequential, so our Spark job will have of unavoidable sequential work.

The job's parallel portion should run in 3 minutes to meet the time limit of 6 minutes. We need to reduce the running time of the job's parallel portion from 27 to 3, a ratio of 9 to 1. Therefore, we need 9 machines to run our Spark job.

Now, it’s time to write the Spark code. To start, you write the following Python code to total up the number of each response type for each lecture, with the goal of converting the input text data to output like so:

Sample Input : (Student ID, Lecture #, Clicker Response)

```
4  1  A
8  1  B
15 1  A
16 2  C
23 2  D
42 2  D
```

Sample Output: ((Lecture #, Clicker Response), Count)

```
((1, A), 2)
((1, B), 1)
((2, C), 1)
((2, D), 2)
```

```
def parseLine(line):
      tokens = line.split(' ')
      return ((tokens[1], tokens[2]), 1)

def addTogether(a, b):
      return a + b

if __name__ == '__main__':
      # Assume you have a Spark Context set up
      clicker_responses = sc.textFile("clicker_data.txt")
      totals = clicker_responses.___(A)____(parseLine).___(B)___(addTogether)
```

b. What function(s) can we use in blank (A) in the code above? Circle your response.

[✓ `map`]   `flatMap`   **Either `map` or `flatMap` Will Work**   map is one-to-one transformation

c. What function(s) can we use in blank (B) in the code above? Circle your response.

`reduce`   [✓ `reduceByKey`]   **Either `reduce` or `reduceByKey` Will Work**

reduce combines regardless of keys

d. Assuming parts b and part c are chosen correctly, are `parseLine` and `addTogether` correctly implemented so that the Spark job will convert the input data to the output? You can ignore the output ordering.

[✓ Yes]   **No**

e. Now, you want to write code to finish the job and take the output from the previous Spark snippet (assuming it is correct) and find the most common response of each lecture.

Sample Input: ((Lecture #, Clicker Response), Count)
```
((1, A), 2)
((1, B), 1)
((2, C), 1)
((2, D), 2)
```

Sample Output: (Lecture #, Most Common Response)
```
(1, (A, 2))
(2, (D, 2))
```

Fill in the blanks in the Python code below to complete the implementation.

```
def rearrange(arg):
    return (arg[0][0], (arg[0][1], arg[1]))

def reduce_func(a, b):

     if a[1] > b[1]:
         return a
     else:
         return b

if __name__ == '__main__':
    # Assume we have the output from the previous code in totals
    most_common_responses = totals.map(rearrange) \
                              .reduceByKey(reduce_func)
```

You input the data, run your code, and type the output into the keypad. With a rumble, the door slides open to reveal Professor Stojanovic and Dr. Weaver, ready to make a quick getaway. Congratulations agent, you have completed your mission. The Professors owe you one for life!

Remember to fill in the emotional spectrum on the title page. Your debriefing packet will be ready in 5-8 days. Thank you for taking CS 61C!

This page is intentionally left blank

This page is intentionally left blank