University of California, Berkeley – College of Engineering

Department of Electrical Engineering and Computer Sciences

Fall 2015 Instructors: Vladimir Stojanovic, John Wawrzynek 2015-12-18

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-
*The name of your **SECTION** TA (please circle)*	Alex | Austin | Chris | David | Derek | Eric | Fred | Jason | Manu | Rebecca | Shreyas | Stephan | William | Xinghua
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 XX numbered pages including the cover page. After you finish the exam, turn in only this booklet, and not the green sheet or datapath diagram.

- Please turn off all cell phones, smartwatches, and other mobile devices. Remove all hats & headphones. Place your backpacks, laptops and jackets under your seat.
- You have 180 minutes to complete this exam. The exam is closed book; no computers, phones, or calculators are allowed. You may use three handwritten 8.5"x11" pages (front and back) of notes in addition to the provided 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. Make sure your solution is on the answer sheet for credit.

	MT1-1	MT1-2	MT1-3	MT1-4	MT2-1	MT2-2	MT2-3	MT2-4	MT2-5
Points Possible	8	8	12	8	9	11	9	12	5

	F-1	F-2	F-3	F-4	Total
Points Possible	9	8	8	11	118

Clarifications during the exam:

MT1 - *depth question*:

1. `struct node * edges[]` should be
   `struct node ** edges`
2. line in the prologue should read
   `sw $ra 0($sp)`
3. line after the epilogue should read
   `lw $ra 0($sp)`

MT1-4: The label on Line 17 should be L2
MT2-Floating Point: real: bits 8-15. imaginary: bits 0-7
MT2-Clobbering Time:
b. The cache is **direct-mapped**
c. Memory accesses = accesses in the `for loop` at part I
F-1: c. What is the maximum number of **virtual** pages a process can use?

MT1-1: Potpourri - Good for the beginning… (8 points)

a. **True/False**:

i. The compiler turns C code into instructions ready to be run by a processor **F**

ii. The instruction `addiu $t0 $t1 0x10000` is a TAL instruction **F**

iii. The linker computes the offset of all branch instructions **F**

i. False. The compiler turns C code into assembly code, not things to be passed into processor just yet
ii. False. The immediate needs to be broken up as it is too long. Also, register names still need to be converted.
iii. False. This is done in the Assembler (pass 1/pass 2)

b. **Memory Management**

```
int global = 0;

int* func() {
    int* arr = malloc(10 * sizeof(int));
    return arr;
}

int main() {
    char* str = "hello world";
    char str2[100] = "cs61c";
    int* a = func();
    return 0;
}
```

In what part of memory are each of the following values stored?

***str: static**
str2[0]: stack
a: stack
arr: stack
arr[0]: heap

*str: Static. "hello world" is a string literal and string literals are stored in Static Data. Because this is a pointer and not an array, the pointer points to actual string literal. Thus, dereferencing the pointer leads to a value stored in Static Data.
str2[0]: Stack. Although "cs61c" is a string literal, because we are assigning into an array and the array is a separate chunk of memory stored on the Stack, the string literal gets copied into str2. Thus, str2[0] exists on the Stack.
a: Stack. We are declaring an integer pointer called a, which is stored on the stack.
arr: Stack. arr is created in a function, thus is on the Stack.
arr[0]: Heap. arr itself is stored on the stack, but we allocated memory in the Heap, which is what arr points to. So dereferencing arr yields a value stored in the Heap.

MT1-2: *C*-ing images through a kaleidoscope (8 points)

Consider a grayscale image with a representation similar to the one you worked with in Project 4, where the image is represented by a 1-dimensional array of `chars` with length `n` x `n`. Fill out the following function `block_tile`. It returns a new, larger image array, which is the same image tiled `rep` times in both the x and y direction. You may or may not need all of the lines.

For a better idea of what must be accomplished, consider the following example:

```
char *image = malloc(sizeof(char) * 4);
image[0] = 1;
image[1] = 2;
image[2] = 3;
image[3] = 4;
char *tiled_image = block_tile(image, 2, 2);
```

The contents of `tiled_image` would then look like:

```
tiled_image: [1, 2, 1, 2,
              3, 4, 3, 4,
              1, 2, 1, 2,
              3, 4, 3, 4];
```

blank 1: n * rep. This is because we have n-width per single picture and rep pictures wide.
blank 2: new_width * new_width * sizeof(char). We need a square matrix with side length new_width. The sizeof(char) is needed because we weren't given the guarantee that a char is 1 byte.

SID: ________________

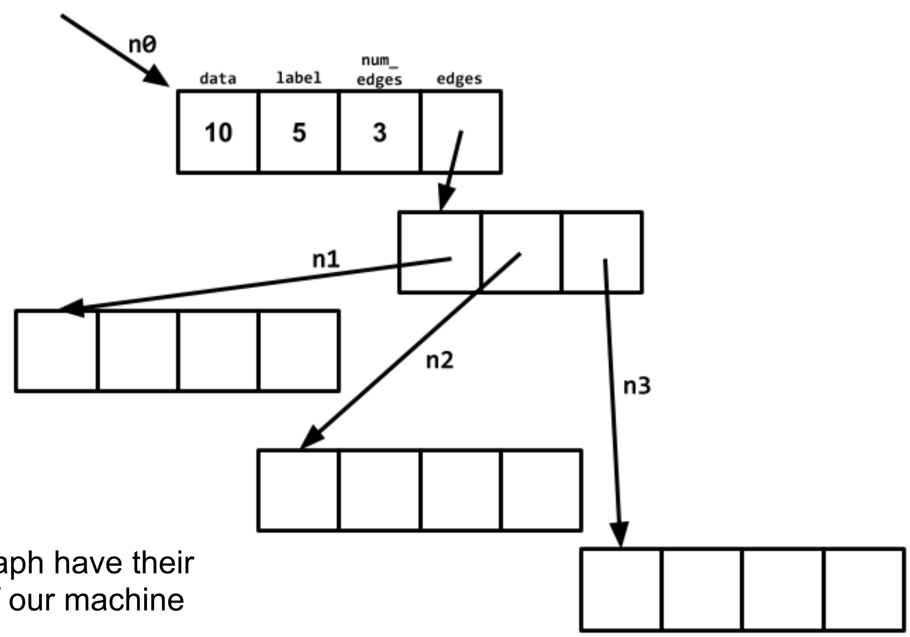
blank 3: i % n. i represents the column. We want our old_x to be the relevant column in our smaller picture. Because the column of the small picture repeats every n, we can just use modulo.
blank 4: j % n. j represents the row. With the same logic as blank 3, we can use modulo.
blank 5: j * new_width + i. Remember that our final array is 1-d, so we index into it accordingly.
blank 6: Not needed as we finished creating the array already
blank 7: Not needed as we finished creating the array already

```
char *block_tile(char *block, int n, int rep) {
    int new_width = ____n * rep____________________________________;
    char *new_block = malloc(__new_width * new_width * sizeof(char)____);
    for (int j = 0; j < new_width; j++) {
        for (int i = 0; i < new_width; i++) {
            int old_x = _______i % n_________________________________;
            int old_y = ____________j % n____________________________;
            int new_loc = ____________j * new_width + i______________;
            new_block[new_loc] = block[old_y * n + old_x];
        }
    }
    ________________________________________;
    ________________________________________;
    return new_block;
}
```

MT1-3: Easy questions have no *depth* – this one does (12 points)

We're interested in running a depth-first search on a graph, and labeling the nodes in the order we finish examining them. Below we have the struct definition of a node in the graph, and the implementation of the function in C.

```
struct node {
    int data;
    int label;
    int num_edges;
    struct node* edges[];
}
```



Note that initially, all nodes in the graph have their label set to -1. The address width of our machine is 32 bits.

```
int dfs_label(struct node* node, int counter) {
     if (node->label != -1) {
          return counter;
     }
     for (int i = 0; i < node->num_edges; ++i) {
          counter = dfs_label(node->edges[i], counter);
```

```
    }
    node->label = counter++;
    return counter;
}
```

Implement `dfs_label` in TAL MIPS. Assume `node` is in `$a0` and `counter` is in `$a1`. You may not need all the lines provided.

```
dfs_label:
# prologue
    addiu $sp $sp _-12_        We need to make space for 3 registers to save on the stack 3 registers * 4 bytes/register = 12 bytes
    sw $ra ($sp)
    sw $s0 4($sp)
    sw $s1 8($sp)
# base case
    addiu $t1 _$0_ _-1_        Load in the -1 for comparison for the base case
    __lw $t0 4($a0)__          Load in node->label for comparison for the base case
    __addu $v0 $0 $a1__        Load in $a1 (counter) into the return value register to since epilogue doesn't do that
    bne $t0 $t1 epilogue
# loop
    addu $s0 _$a0_ _0_         Save node ($a0) in a saved register because of Caller/Callee
    addiu $s1 $0 0
loop:
    lw $t0 _8($s0)_            Load in node->num_edges into $t0
    beq _$t0_ _$s1_ _fin_      If i ($s1) is greater than $t0 (node->num_edges), we stop
    lw $a0 _12($s0)_           # load edges into $a0
    sll $t0 $s1 _2_            Shift by 2 to go from index to byte offset (left shift by 2 == multiply by 4)
    _addu $a0 $a0 $t0_         # load the next node
    lw $a0 _0($a0)_            # into $a0
    jal dfs_label
    addu $a1 _$v0_ _$0_        Put the return value ($v0) of dfs_label in to $a1 (counter)
    addiu $s1 $s1 1
    j loop
fin:
    sw $a1 _4(s0)_             Put counter into node->label
    _addiu $a1 $a1 1_          Increment counter (node->label = counter++)
    addu $v0 $0 $a1
epilogue:
    lw $ra ($sp)
    lw $s0 4($sp)
    lw $s1 8($sp)
    addiu $sp $sp _12_         Move the stack pointer back how much we moved it earlier
    jr $ra
```

SID: _______________

MT1-4: Can't reveal this MIPS-tery (8 points)

```
 0| Mystery:
 1|          add $t0, $a0, $0
 2|          add $t1, $a1, $0
 3|
 4|          la $s0, L1
 5|          lw $s1, 12($s0)
 6|          addi $s2, $0, 6
 7|          addi $s3, $0, 0
 8|          addi $s4, $0, 1057        #$s4 contains 0b0100 0010 0001
 9|          sll $s4, $s4, 11
10|
11| L1:      beq $s3, $s2, L2
12|          addu $s1, $s1, $s4
13|          sw $s1, 12($s0)
14|          addu $t1, $a3, $t0
15|          addi $s3, $s3, 1
16|          j L1
17| Done:
```

Important to note that \$s4 is set up so that it changes each of the 3 register fields, adding one to every single register field every time

a. When the above code executes, which line is modified? How many times?

Line 14, 6 times We load in L1 into \$s0 on line 4. Then we sw \$s1, 12(\$s0), meaning 3 instructions after L1 or line 14. We have a counter \$s3 which starts at 0 and ends when \$s3 == \$s2, and since \$s2 starts at 2, we modify line 14, 6 times.

b. Assume we run this block of code with `$a0 = 1` and `$a1 = 1`; what is the value in `$t2` at the end of the code execution? How about `$t3`?

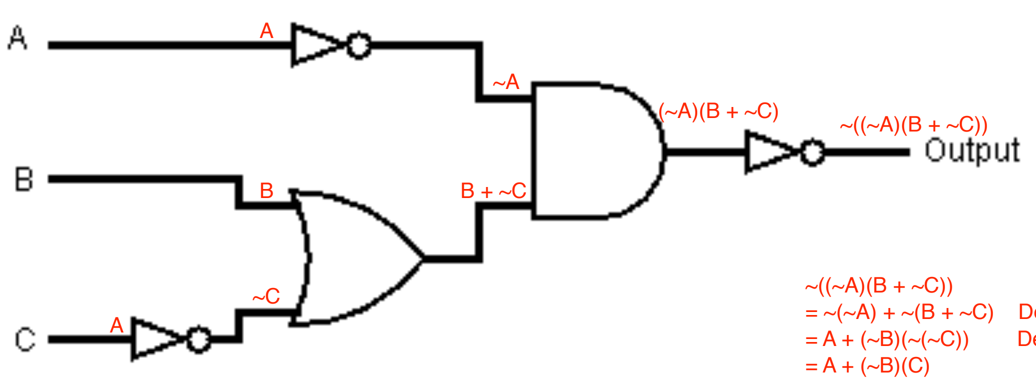
\$t2 = 2, \$t3 = 3 See part c as to how the temporary registers get affected. \$t2 = \$t1 + \$t0 = 1 + 1 = 2
\$t3 = \$t2 + \$t1 = 2 + 1 = 3

c. In three sentences or less, how does this code affect the temporary registers?

It takes the arguments \$a0 and \$a1 and stores then in registers \$t0 and \$t1. Then for the remaining temporary registers, it sets register \$t_{n} to \$t_{n-1} + \$t_{n-2}.

MT2-1: Synchronous Finite State Digital Machine Systems (9 points)

a. The circuit shown below can be simplified. Write a Boolean expression that represents the function of the simplified circuit using the minimum number of AND, OR, and NOT gates.

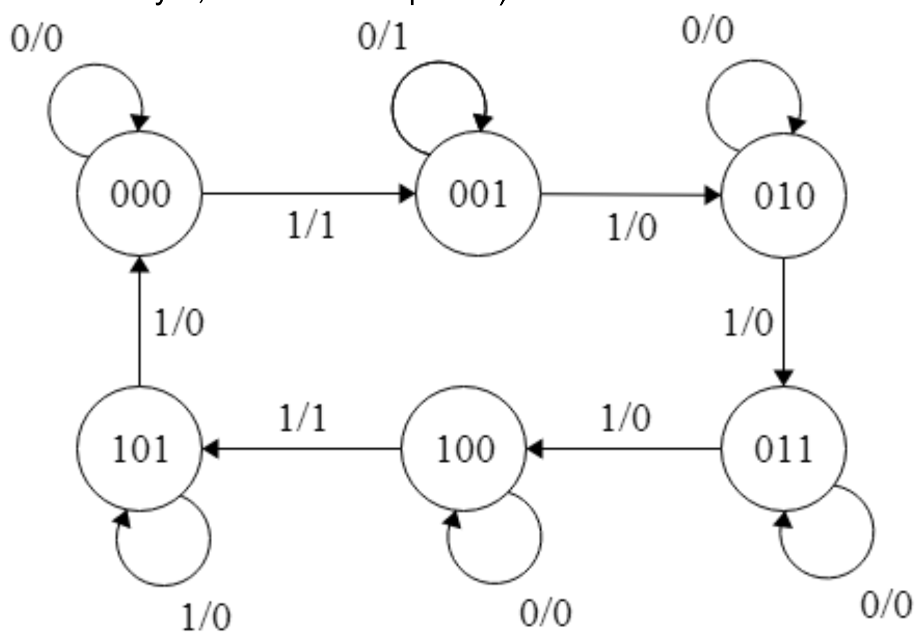


~((~A)(B + ~C))
= ~(~A) + ~(B + ~C)   DeMorgan's Law
= A + (~B)(~(~C))   DeMorgan's Law
= A + (~B)(C)

~(~(A) (B+~(C)) = A + ~(B+(~C)) = A + (~B)C

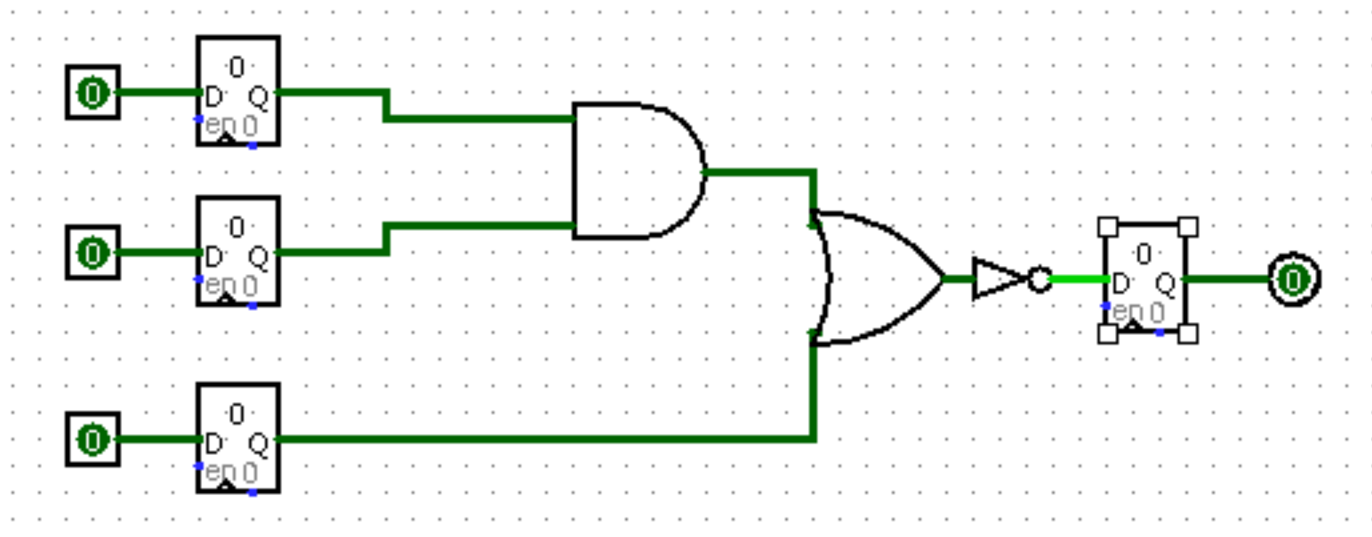
b. Consider the finite state machine below which has 6 states, and a single input that can take on the value of 0 or 1. The finite state machine should output 1 *if and only if* 6 + the sum of all the input values is not divisible by 2 or 3. One transition has been provided; complete the remainder of the diagram.

(Hint: If the sum of the inputs is a multiple of 6, then we have 6 + the sum of the inputs = 6n for some n. As 6n is divisible by 2, 6n cannot be prime.)



All transitions going to 001 and 101 should output 1, as we get something in the form of 6n+1 and 6n+5 respectively. If you look at the 4 other cases, 6n, 6n+2, 6n+3, 6n+4, all numbers in these forms are divisible by either 2 and/or 3, and therefore can never be prime for n >= 1 (Which is what we have as we add 6 to the sum of all our inputs). Therefore for all transitions going to 000, 010, 011, and 101, we should output 0.

c. Consider the following circuit. Assume registers have a CLK to Q time of 60ps, a setup time of 40ps, and a hold time of 30ps. Assuming that all gates have the same propagation delay, what is the maximum propagation delay each individual gate could have to achieve a clock rate of 1GHz.



300 ps We have a clock rate of 1 GHz, which means that we have a maximum clock period of 1000 ps. Following the relevant formula, the critical path involves 3 combinational logic, one CLK-to-Q and one setup time, which equals 3 * CL + 60 ps + 40 ps = 1000 ps. Solving for CL, we see that 3 * CL = 900 ps, which means that CL = 300 ps.

MT2-2: Do stray off of the well-worn *datapath* (11 points)

Suppose we have a new instruction, `bmeq`. We branch if the value in memory at the address in `$rs` equals the value in `$rt`. The instruction format is as follows:

```
bmeq ($rs) $rt offset
```

Use the datapath diagram provided as a reference for input and output names. Assume we are working with a non-pipelined single cycle datapath.

a. Write the register transfer language that represents the logic of this command.

`If MEM[$rs] == $rt PC = PC + offset << 2; else PC = PC + 4`

Basically, we want to branch to PC + offset << 2 (since MIPS stores the offset in words) if the memory at $rs equals the value at $rt. Otherwise we don't branch like any other instruction and do PC += 4.

b. You are given a new control signal, BMEQ, which is `1` when it is a BMEQ instruction and `0` when it is not. In the following table, please fill in the inputs, control signal, and output destination for any additional MUXes you would need in order for this instruction to work correctly. You might not need all the lines.

Inputs: ReadData1, ReadData2, ReadData, AluOut, MemoryData, PC
Output destinations: Addr, ReadReg1, ReadReg2, WriteAddr, InputA, InputB, ReadAddr, WriteData

Control Signal	Input0	Input1	Output Destination
BMEQ We want to choose between ALU (load/store) and also rs1 or ReadData1	**ReadData1**	**ALUOut**	**ReadAddr**
BMEQ We want our InputA into our ALU to either be ReadData1 (rs1) or MEM	**ReadData1**	**MemoryData**	**InputA**

We aren't writing to a register so X for RegDst and 0 for RegWr, branch offset is either forward or backward so SIGN, ALU Src is 0 because we want to use the ALU to compare. ALUCtr should actually be sub since we want to compare Mem and rs2. We aren't writing to Memory, nor are we writing to register, so MemtoReg does not matter. We are branching but not jumping.

d. Fill in the values for the control signals for this new instruction. Use X if the signal does not matter. For ExtOp, write SIGN for sign-extension and ZERO for zero-extension.

Reg Dst	ExtOp	RegWr	ALU Src	ALU Ctr	Mem Wr	Memto Reg	Branch	Jump	BMEQ
X	**SIGN**	**0**	**0**	**SUB**	**0**	**X**	**1**	**0**	1

e. In ≤ 1 sentence, why can't this instruction work with a normal pipelined 5-stage MIPS datapath?
The execute stage now requires a data to be loaded in from memory.

MT2-3: Enough *stalling*, that will only slow you down (9 points)

Consider the standard 5-stage pipelined MIPS CPU with instruction fetch, register read, ALU, memory, and register write stages. Register writes happen before register reads in the same clock cycle, branch comparison is done during the register read stage, there is a branch delay slot, and forwarding is implemented.

For the following stream of instructions, assume that `$t0` is not equal to `0`, so the branch is not taken.

```
0| start:    lw $t0 0($a0)
1|           beq $t0, 0, end
2|           addiu $t0, $t0, 10
3|           sw $t0 0($a0)
4| end:
```

a. For each pair of instructions, circle whether the CPU needs to be stalled for the execution of the second instruction, and if so, for how many cycles.

```
i.  0| start:  lw $t0 0($a0)
    1|         beq $t0, 0, end
```
stall for 2 cycles

$t0 isn't ready until the end of MEM stage
IF ID EX ME WB
IF ID EX ME WB
If branch comparison occurs in the ID stage, we need to forward the result of ME to the start of the IF stage, aka stall for 2 cycles:
IF ID EX ME WB
IF ID EX ME WB

```
ii. 1|         beq $t0, 0, end
    2|         addiu $t0, $t0, 10
```
no stall

IF ID EX ME WB
IF ID EX ME WB
We know if the branch is taken in ID, but because of the Branch Delay Slot, we already have an implicit stall. Thus we know if the branch happens before we execute Line 2.

```
iii. 2|        addiu $t0, $t0, 10
     3|        sw $t0 0($a0)
```
no stall

We can forward the result of $t0 (EX) to the beginning of the next EX, $t0 is at a minimum needed by the ME stage, so no stall is needed.
IF ID EX ME WB
IF ID EX ME WB

Logic in each stage of the pipeline has the following timing:

Instruction Fetch	Register Read	ALU	Memory	Register Write
150 ps	100 ps	100 ps	200 ps	100 ps

The pipelining registers in between stages have the following timing:

Clock-to-Q	Hold time	Setup

30 ps	20 ps	30 ps

b. What is the minimum clock period, in picoseconds, for which the processor can run?
260 Because of our pipeline, we clock to the slowest stage. This means that the longest CL is from Memory, resulting in 200 + CLK-to-Q + Setup = 260 ps.

c. What is the time required, in picoseconds, that it takes for the CPU, starting from the first stage of the lw instruction, to finish the execution of the final sw instruction? You may use the variable stall_cycles in place of the sum of your answers for question a, and clock_period in place of your answer for question b.

(8 + stall_cycles) * clock_period

We have 4 instructions and thus 8 cycles assuming no stalls. This is because IF ID EX ME WB, the last instruction doesn't theoretically start until at least cycle 5 because of pipelining and Branch Delay Slot.

d. Which timing values, if lowered independently (all other timing remain the same), will allow us to increase the frequency of the CPU? Circle all that apply.

Pipelining Register Clock-to-Q, Pipelining Register Hold time, Pipelining Register Setup time, Instruction Fetch, Register Read, ALU, Memory, Register Write

pipelining register clock-to-q, pipelining register setup time, memory (can be either circled or not circled) Basically all the parts that contributed to the critical path.

MT2-4: If you do well, it's *clobbering* time! (12 points)

The information for one student in regards to clobbering a single midterm is captured in the data of the following *tightly-packed* struct:

```
typedef struct student {
   int studentID;
   float oldZScore;
   float newZScore;
   int clobber;         /* a value equal to 1 if a student clobbers,
                           0 if otherwise */
} student;
```

We run the following code on a 32-bit machine with a 4 KiB write-back cache. `importStudent()` returns a `struct student` that is in the course roster and that has not been returned by `importStudent()` previously. For simplicity, assume `importStudent()` does not affect the cache.

```
int ARR_SIZE = 512; //Class size rounded down for simplicity
student *61CStudents = (student *) malloc (sizeof(student) * ARR_SIZE);

/* Assume malloc returns a cache block aligned address */
for (int i = 0; i < ARR_SIZE; i++) {                        <=== part I
   61CStudents[i] = importStudent()  <- what does import student do?
}

for (int i = 0; i < ARR_SIZE; i++) {                        <=== part II
   if (61CStudents[i].oldZscore > 61CStudents[i].newZscore){
     61CStudents[i].clobber = 0;
   } else {
```

MT2-4: If you do well, it's *clobbering* time! (12 points)

The information for one student in regards to clobbering a single midterm is captured in the data of the following *tightly-packed* struct:

```
typedef struct student {
    int studentID;
    float oldZScore;
    float newZScore;
    int clobber;         /* a value equal to 1 if a student clobbers,
                            0 if otherwise */
} student;
```

We run the following code on a 32-bit machine with a 4 KiB write-back cache. `importStudent()` returns a `struct student` that is in the course roster and that has not been returned by `importStudent()` previously. For simplicity, assume `importStudent()` does not affect the cache.

```
int ARR_SIZE = 512; //Class size rounded down for simplicity
student *61CStudents = (student *) malloc (sizeof(student) * ARR_SIZE);

/* Assume malloc returns a cache block aligned address */
for (int i = 0; i < ARR_SIZE; i++) {                      <=== part I
    61CStudents[i] = importStudent()  <- what does import student do?
}

for (int i = 0; i < ARR_SIZE; i++) {                      <=== part II
    if (61CStudents[i].oldZscore > 61CStudents[i].newZscore){
        61CStudents[i].clobber = 0;
    } else {
```



```
        61CStudents[i].clobber = 1;
    }
}
```

a. How many bytes is needed to store the information for a single student?

The struct has 4 fields, each of 4 bytes, so 16 bytes

b. Assume that the block size is 32 B. What is the tag:index:offset breakdown of the cache?

The final clarifications told the students to assume a direct mapped cache. Thus, if the cache size is 4KiB (2^12), and the block size is 32 B (2^5), there must be 2^12/2^5 = 2^7 blocks. Thus, there are 7 index bits, 5 offset bits, and 20 tag bits. 20:7:5

c. At the label part I, assume that 61CStudents is filled with the correct data. What type of misses will occur from memory accesses during the process? Why?

The misses that will occur from executing the for loop at label part I will be compulsory misses, because the loop will be accessing student structs that will be accessed for the first time.

d. Suppose we run the code again and the cache block size is now 8 B long and the cache is direct mapped. For the for-loop in part II, what is the miss rate in the best case scenario (we want the highest hit rate possible)? What type of misses occur?
The if-else block in part II has 3 memory accesses. Since half of the struct fits in a block of the cache, 61CStudents[i].oldZscore would be a miss and load the first two elements of the struct into a block, 61CStudents[i].newZscore would be a miss and load the last two elements of the struct into a block, and 61CStudents[i].clobber would be a hit in the second block loaded in. Thus, the miss rate is 2/3. These misses are capacity misses because even if the cache was fully associative, the array of structs does not fit in this cache, and entries would have to be evicted due to the cache size in the fully associative case.

e. For the for loop in part II, assume that the cache block size is now 128B.

i. If the cache is direct-mapped, what is the hit rate?

8 students per block. 3 memory access per student, 1 miss and 23 hits, so 23/24

ii. If the cache is fully associative, what is the hit rate? Does associativity help? Why or why not?

It would still be 23/24, because the array is being accessed sequentially, so associativity makes no difference.

MT2-5: What is the *floating point* of *complex* numbers? (5 points)

We realize that you want to represent complex numbers, which are in the form $a + bi$, where a is the real component, b is the imaginary component, and the magnitude is $\sqrt{a^2 + b^2}$.

We create a 16-bit representation for storing both the real and imaginary components as floating point numbers with the following form: The first 8 bits will represent the real component, and the latter 8 bits will represent the complex component. Our new representation will look like:

Sign	Exponent	Significand	Sign	Exponent	Significand
15	14-12	11-8	7	6-4	3-0

Bits per field:

Sign: 1
Exponent: 3
Significand: 4
Everything else follows the IEEE standard 754 for floating point, except in 16 bits

Bias: 3

a. Convert 0xB248 into the complex number form a + bi

0xB248 -> 0b1011 0010 0100 1000

For the real part, significand is 0010, exponent is 011, sign is 1. If we convert the exponent to bias, we actually get an exponent of 0. Thus, converting the significand to normalized form gives -1.001 in binary, which is -1.125 in decimal.

For the imaginary part, significand is 1000, exponent is 100, sign is 0. If we convert the exponent to bias, we get an exponent of 1. Thus, converting the significand to normalized form gives 1.1 * 2^1 in binary (which equals 11.0… in binary), which is just 3.

Thus -1.125 + 3i is the answer.

b. What is the smallest positive number you can represent with a nonzero real component and zero complex component?
For the positive component, the smallest exponent possible is 0 and the smallest significand is 0001. Remember, this is denormalized because the exponent is zero, thus the bias is always the negative of one smaller than the positive bias (-2). Converting this to normalized form gives 0.0001*2^-2 in binary, which is 0.000001 = 2^-6

Recall the following floating point representation from the midterm:

Sign	Exponent	Significand
15	14-9	8-0

Bits per field:

Sign: 1
Exponent: 6
Significand: 9
Everything else follows the IEEE standard 754 for floating point, except in 16 bits

c. Ignoring infinities, which of the two representations presented above can represent a number with the larger magnitude.
The midterm floating point representation can represent 2^6 in magnitude. For the complex number representation, if you had the largest exponent in the real and the imaginary part possible, and took the magnitude per the equation in the directions, your exponent would be 2^5 (due to the square root). Thus, the midterm floating point representation is larger in magnitude.

F-1: You may need to context switch for this question

The system in question has 1MiB of physical memory, 32-bit virtual addresses, and 256 physical pages. The memory management system uses a fully associative TLB with 128 entries and an LRU replacement scheme.

a. What is the size of the physical pages in bytes?
Physical page size is going to be the size of physical memory divided by the number of pages in physical memory. 1MiB = 2^20 and 256 pages = 2^8, thus 2^20/2^8 = 2^12 bytes

b. What is the size of the virtual pages in bytes?
Virtual and physical pages are always the same size, thus 2^12 bytes.

c. What is the maximum number of virtual pages a process can use?
Similar to part a, the number of pages in virtual memory will be the size of virtual memory divided by the size of a page. Virtual memory in this case is 2^32 bytes, thus 2^32/2^12 = 2^20 pages.

d. What is the minimum number of bits required for the page table base address register?
The page table base register holds a physical address which is a pointer to the start of the page table for the current running process. Thus, since this register holds a physical address, and every physical address is 20 bits (because physical memory is 2^20 bytes), 20 bits are needed.

Everybody Got Choices

i. The page table is stored in main memory. True, page table must be stored in memory to be able to be used.

ii. Every virtual page is mapped to a physical page False, for a couple reasons that I can think of. Virtual memory is usually much bigger than physical memory, so every virtual page cannot be mapped to a physical page. Also, the operating system may prevent some virtual pages to be mapped to physical pages for access reasons (restricted memory regions)

iii. The TLB is checked before the page table Definitely True

iv. The penalty for a page fault is about the same as the penalty for a cache miss False. Pages are much bigger than cache blocks, and disk is much slower than memory, so reading a page from disk into memory is much slower than reading a block from memory into a cache

v. A linear page table takes up more memory as the process uses more memory False. The basic page table we cover is a linear page table, which is stored all in memory with enough indexes for all of the virtual page numbers. If the process uses more memory, more entries in the page table will be valid (valid bit set to 1), but the page table will always be the same size.

F-2: Not all *optimizations* are created equal (8 points)

For this question, you will be looking at several different versions of the same code that has been, or at least has tried to be, optimized. For each of the versions, indicate the correctness and speed with the appropriate letter:

Correctness:
A. Always Correct
B. Sometimes Correct
C. Always Incorrect

Speed:
A. Faster
B. Same
C. Slower

For reference, here is the serial version of the code:

```
#DEFINE RESULT_ARR_SIZE 8
#DEFINE ARR_SIZE 65536

result[0] = 0;
for (int i = 1; i < RESULT_ARR_SIZE; i++) {
    int sum = 0;
    for (int j = 0; j < ARR_SIZE; j++) {
        sum += arr[j] + i;
    }
    result[i] = sum + result[i - 1];
}
```

a. Version 1:

```
result[0] = 0;
#pragma omp parallel
for (int i = 1; i < RESULT_ARR_SIZE; i++) {
    int sum = 0;
    for (int j = 0; j < ARR_SIZE; j++) {
        sum += arr[j] + i;
    }
    result[i] = sum + result[i - 1];
}
```

Correctness: The answer is actually A, always correct, contrary to the published solutions. This is simply due to the fact that because this code block is wrapped in #pragma omp parallel, every thread will simply be overwriting the result array with the same numbers in each index. This can be proved formally by induction, but the simple explanation is that since result[0] is always 0 for all threads, and every thread will calculate the same sum in the inner for loop, then result[1] must be the same for all threads, even if they overwrite one another. If result[1] must be the same for all threads, and they all calculate the same sum in the inner for loop, then result[2] must be correct for all threads. This same argument can be made for the rest of the result array. result[0] is the key element that makes this code always correct.

Speed: C, Slower. Due to the overhead of spawning multiple threads to complete the same iterations for all of the for loops.

b. Version 2:

```
result[0] = 0;
#pragma omp parallel for
for (int i = 1; i < RESULT_ARR_SIZE; i++) {
    int sum = 0;
    for (int j = 0; j < ARR_SIZE; j++) {
        sum += arr[j] + i;
```



```
    }
    #pragma omp critical
    result[i] = sum + result[i - 1];
}
```

Correctness: B, Sometimes correct. This is because of the data dependency on the previous element of result in the last line of code. If result[i-1] is not calculated, result[i] will be wrong. However, in theory, if all of the threads lined up and were scheduled such that result[1] was calculated, then result[2], then result[3], etc., then this code would be correct.

Speed: A, faster. Because we split up the outer for loop between threads, this outer for loop can be done concurrently, which would be faster than the naïve version. The critical section just applies to adding to the result array.

c. Version 3:

```
result[0] = 0;
for (int i = 1; i < RESULT_ARR_SIZE; i++) {
    int sum = 0;
    #pragma omp parallel for reduction(+: sum)
    for (int j = 0; j < ARR_SIZE; j++) {
        sum += arr[j] + i;
    }
    result[i] = sum + result[i - 1];
}
```

Correctness: A, always correct. This is because the inner for loop is parallelized to many threads, and preserves the correctness of the sum variable with the reduction keyword. The result array is written to sequentially by in the outer for loop with no false sharing/race conditions.

Speed: A, faster. This is because we can assume that the parallel for in the inner for loop speeds up the execution of the loop enough to dwarf any overhead. Thus, this code would run faster than the naïve version.

d. Consider the correctly parallelized version of the serial code above.

i. Could it ever achieve perfect speedup? F. Amdahl's law says that speedup is limited by the non-parallelizable portion of any code, and that perfect speedup is not possible.

ii. What law provides the answer to this question? Amdahl's Law, per above

F-3: *Map* and *Reduce* are 2nd degree friends - when you also Combine (8 points)

Imagine we're looking at Facebook's friendship graph, which we model as having a vertex for each user, and an undirected edge between friends. Facebook stores this graph as an adjacency list, with each vertex associated with the list of its neighbors, who are its friends. This representation can be viewed as a list of degree 1 friendships, since each user is associated with their direct friends. We're interested in finding the list of degree 2 friendships, that is, an association between each user and the friends of their direct friends.

You are given a list of associations of the form `(user_id, list(friend_id))`, where the `user_id` is 1st degree friends with all the users in the list.

Your output should be another list of associations of the same form, where the first item of the pair is a `user_id`, and the second item is a list of that user's 2nd degree friends. **Note:** a user is not their own 2nd degree friend, so the list of second degree friends must not include the user themselves.

Write pseudocode for the mapper and reducer to get the desired output from the input. Assume you have a set data structure, with `add(value)` and `remove(value)` methods, where `value` can be an item or a list of items. You can iterate through a list with the `for item in items` construct. You may not need all the lines provided.

MapReduce

```
map(user_id, friend_ids):
    for friend in friend_ids:
        emit(friend, friend_ids)
```

In our map phase, we want to somehow associate a user with their second degree friends. The key here is to recognize that every friend in a user_id's friend_ids list is a second degree friend to every other friend in that list (linked through the user_id). Thus, we want to emit the tuples (friend, friend_ids) to signify that friend is second degree friends with everyone else in a user_id's friends list.

```
reduce(key, values):
    second_degree_friends = set()
    for value in values:
        second_degree_friends.add(value)
    second_degree_friends.remove(key)
    emit(key, second_degree_friends)
```

In reduce, we simply combine all of the second degree friends lists which correspond to a specific user, and then remove that user from the list of second degree friends.

Potpourri

a. We have a hard drive with a controller overhead of 5 ms. The disk has 12000 cylinders, and it takes 2 ms to cross 1000 cylinders. The drive rotates at 2400 RPM, and we want to copy half a MB of data. Our hard drive has a transfer rate of 500 MB/s. What is the access time of a read from disk?

We want to add up the times that it takes to access this disk. We add them up in the following order: Seek + Rotation + Transfer + Controller. The number of tracks in the seek time is always number of tracks/3, Rotation time is averaged by taking half of the time to rotate around a disk.

12000/3 * 1/(24000/60/2) + 2/1000) + 1000 * 1000*((1/2)/500) + 5ms = 15.25 ms

b. I launched a new online app at the start of this year (2015), and I want to have at least three nines of availability per year. Up until today, my app has been available at all times this year. However, some malicious hackers crashed my app for today; it took me 4 hours to get it back up again. For the rest of this year, what is the most downtime I can have on my app to meet my availability goals, rounded to the closest hour? (There are 8760 hours this year)

We want the ratio of downtime to number of hours in the year to be 0.999, as this is the availability of the app. Thus, we solve for the equation (4+x)/8760 = 0.999 = 5 hours

Another way to think about it is that we know that the app can only be down for 0.001 of the year. 8760*0.001 = 8.76, so we round up to 9 hours of downtime total per year. Since the app was already down for 4 hours today, we know we have at most 5 more hours of downtime for the rest of the year.

c. c. If a receiver checks the header and the checksum is correct, what does it do? (In ≤ 1 sentence)

Ack

When a packet header and contents have been verified to be correct by a receiver, it will Ack that the packet has been received (send an Ack packet back to the sender). The receiver can then continue to process the packet payload.

d. For the standard single-error correcting Hamming code presented in class, is the 12-bit code word 0x61C corrupted? What is the correct data value in decimal format?

0x61C = 0b0110 0001 1100 P1: 0 ^ 1 ^ 0 ^ 0 ^ 1 ^ 0 = 0 P2: 1 ^ 1 ^ 0 ^ 0 ^ 1 ^ 0 = 1 P4: 0 ^ 0 ^ 0 ^ 0 ^ 0 = 0 P8: 1 ^ 1 ^ 1 ^ 0 ^ 0 ^ 0 = 1 0x61C = 0b0110 0001 1100 => 0b0110 0001 1000 = 0b1000 1000 = 128 + 8 = 136

e. True/False

i. Raid 4 allows for concurrent independent writes to disk. F
Raid 4 is limited by writes to the parity disk. Since every write to any of the disks in the system must also write the parity disk, concurrent independent writes are not possible in Raid 4

ii. Raid 5 allows for concurrent independent writes to disk. T
Raid 5 solves the problems that Raid 4 has with a parity disk by spreading the parity blocks among the disks, so concurrent independent writes can happen.

iii. Raid 5 allows for concurrent independent reads to disk. T
Reading is not an issu

iv. IP guarantees delivery F
IP provides best-effort delivery in the network layer, and provides no guarantees of end to end packet delivery. It is up to higher layers in the network stack (TCP) to detect and deal with packet failures.