Your Name (first last) _____ SID _____

UC Berkeley CS61C
Final Exam Fall 2019
Solutions

← Name of person on left (or aisle) _____ Name of person on right (or aisle) →

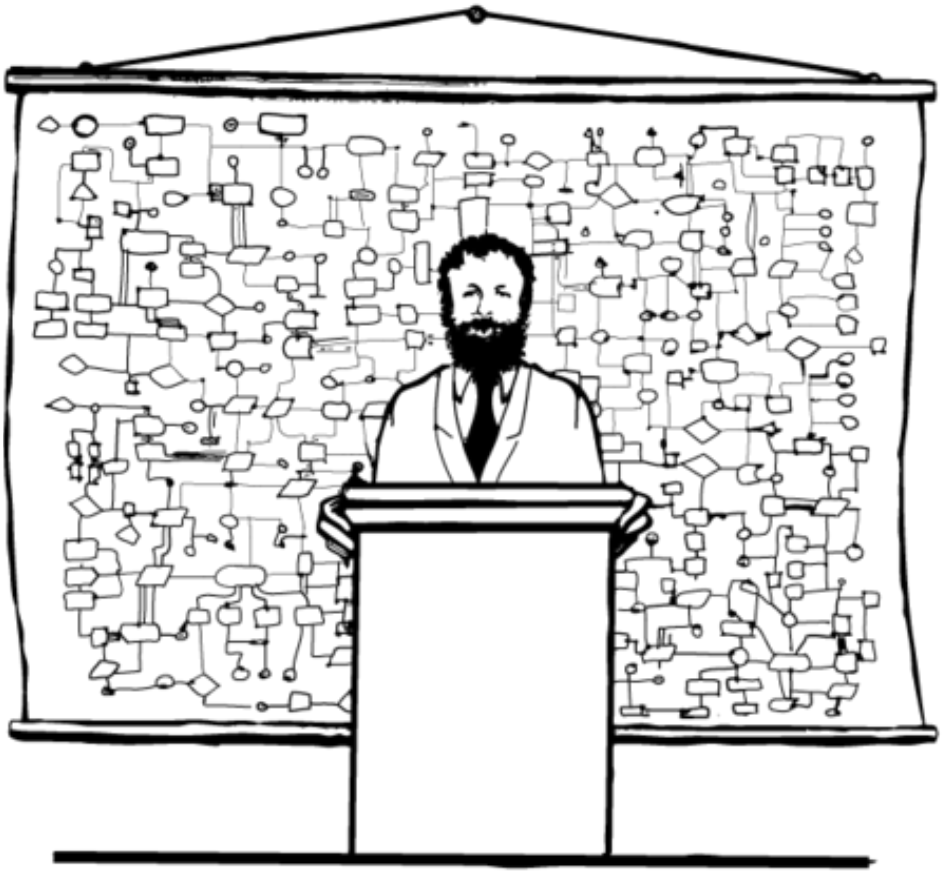
TA name _____

Fill in the correct circles & squares completely…like this: ● (select ONE), and ■(select ALL that apply)

When you see SHOW YOUR WORK, that means a correct answer *without work* will receive NO CREDIT, and your work needs to show how you were led to the answer you reached. If you find that there are multiple correct answers to a "select ONE" question, please choose just one of them.

Question	1	2	3	4	5	6	7	8	9	10	Total
Minutes	2	8	20	30	30	12	18	15	15	30	**180**
Points	5	11	14	30	30	12	18	15	15	30	**180**

Quest-clobber questions: Q2ad, Q3
Midterm-clobber questions: Q1-6



"Now that you have an overview of the system, we're ready for a little more detail"

CS61C Final Clarifications

1. All answers should be fully simplified unless otherwise stated.
2. Value of the float, not the bits
3. ASCII Value 0xFF == nbsp
4. RISCV Qa: 0 < x < 10.

Q1) CALLer/CALLee Convention… **(5 pts)**

Determine which stage of **CALL** each of the following actions happen in. Select ONE per row.	**C**ompiler	**A**ssembler	**L**inker	**L**oader
a) Copying a program from the disk into physical memory Again, that's one of the jobs of the loader.	○	○	○	●
b) Removing pseudoinstructions These are done by the assembler, so that the link editor knows the relative positions of each line of assembler, without needing to worry about pseudocode expansion.	○	●	○	○
c) Determining increment size for pointer arithmetic The assembly code has to know this -- whether `addi t0 t0 1` (for advancing a pointer to an array of `uint8_t`) or `addi t0 t0 4` (for an array of `uint32_t`)	●	○	○	○
d) Incorporating statically-linked libraries That's the job of the *linker*, it incorporates all the required statically-linked libraries into a self-contained executable.	○	○	●	○
e) Incorporating dynamically-linked libraries That's the point of dynamically-linking things at runtime, it becomes the loader's job.	○	○	○	●

Q2) Open to Interpretation **(11 pts = 2 + 3 + 4 + 2)**

Let's consider the hexadecimal value **`0xFF000003`**. How is this data interpreted, if we treat this number as…

a) an array `A` of unsigned, 8-bit numbers? Please write each number in **decimal,** assume the machine is **big endian**, and write **`A[0]`** on the left, **`A[3]`** on the right.

255, 0, 0, 3

SHOW YOUR WORK HERE

Big endian means the big part of the number is at A[0] (unlike how we would normally store the number) so that means the number reads left to right. The first byte is 0xFF, which is 2^8-1 = 255, the second and third are 0x00, which are zero, and the last is 0x03, which is 3.

b) an IEEE-754 single-precision floating point number?

$-(2^{127} + 2^{105} + 2^{104})$

SHOW YOUR WORK

1|111 1111 0|00000...011

exp=254, so number is -1.00...11 x $2^{254-127}$

$-2^{127}\ (1 + 2^{-22} + 2^{-23})$

$-(2^{127} + 2^{105} + 2^{104})$

c) a RISC-V instruction? If there's an immediate, write it in decimal.

`lb x0 -16(x0)`

SHOW YOUR WORK

The opcode is 0b0000011 and the func3 is 0b000, which corresponds to the "lb" instruction. "lb" is an I-type instruction, so we extract rd = 0b00000, rs1 = 0b00000, Imm = 0b111111110000. The register 0b00000 is the x0 register. Finally, we calculate the immediate, taking care to note that immediates are stored in two's complement signed form. Negating the immediate yields 0b000000001111+1 = 0b000000010000 = 16, so the immediate is -16. We then write the instruction in lb format.

d) a **`(uint32_t *)`** variable **`c`** in **little-endian** format, and we call **`printf((char *) &c)`**? If an error or undefined behavior occurs, write "Error". If nothing is printed, write "Blank". Please refer to the ASCII table provided on your reference sheet. For non-printable characters, please write the value in the Char column from the table. For example, for a single backspace character, you would write "**BS".**

ETX

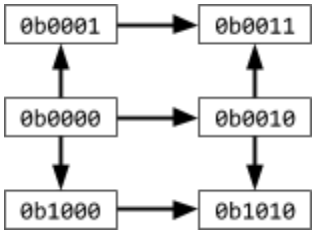
SHOW YOUR WORK

Since the data is in little-endian format, the first byte printed is 0x03, which corresponds to ETX. The second character is 0x00, which is NULL, the null terminator. printf doesn't read past the first null terminator, so we finish printing after we write ETX. Note that the VALUE of c is our number in little-endian format which is why when we do &c, we are saying that value is a string when plugged into printf.

Q3) There's a Dr. Hamming to C you... (14 pts)

We are given an array of **N unique** `uint32_t` that represent nodes in a directed graph. We say there is an edge between **A** and **B** if **A < B** and the Hamming distance between A and B is **exactly 1**. A Hamming distance of 1 means that the bits differ in 1 (and only 1) place. As an example, if the array were {`0b0000, 0b0001, 0b0010, 0b0011, 0b1000, 0b1010`}, we would have the edges shown on the right:

A	B
0b0000	0b0001
0b0000	0b0010
0b0000	0b0100
0b0001	0b0011
0b0010	0b0011
0b0010	0b1010
0b1000	0b1010



Construct an `edgelist_t` (specified below) that contains all of the edges in this graph. Our solution used every line provided, but if you need more lines, just write them to the right of the line they're supposed to go after and put semicolons between them. All of the necessary `#include` statements are omitted for brevity; don't worry about checking for `malloc`, `calloc`, or `realloc` returning NULL. *Make sure `L->edges` has no unused space when L is eventually returned.*

```
typedef struct {
   uint32_t A;
   uint32_t B;
} edge_t;

typedef struct {
   edge_t *edges;
   int len;
} edgelist_t;
```

```
edgelist_t *build_edgelist(uint32_t *nodes, int N) {
   edgelist_t *L = (edgelist_t *) malloc (sizeof(edgelist_t));
   L->len   = 0;


   L->edges = (edge_t *) malloc (N * N * sizeof(edge_t));

   for (int i = 0; i < N; i++) {
       for (int j = 0; j < N; j++) {
           uint32_t tmp = nodes[i] ^ nodes[j];
           if ((nodes[i] < nodes[j]) && !(tmp & (tmp-1))) {
               L->edges[L->len].A = nodes[i];
               L->edges[L->len].B = nodes[j];
               L->len++;
           }
       }
   }
   L->edges = (edge_t *) realloc(L->edges, sizeof(edge_t) * L->len);
   return L;
}
```

Q4) Felix Unger must have written this RISC-V code! **(30 pts = 3*10)**

```
1. mystery:
2.    la t6, loop
3. loop: addi x0, x0, 0        ### nop
4.    lw   t5, 0(t6)
5.    addi t5, t5, 0x80
6.    sw   t5, 0(t6)
7.    addi a0, a0, -1
8.    bnez a0, loop
9.    ret
```

What this function does (courtesy of Albert Zhan on piazza with minor edits):

1. mystery(x) entry point => where a0 = x, this is the label to jump to mystery.
2. la t6, loop => t6 register now stores the address of the "loop" (in this case, it points to an instruction in the text section of memory)
3. addi x0 x0 0 => instruction is executed. Note that addi instruction looks like
   _ _ _ _ _ _ _ _ _ _ _ _ + _ _ _ _ _ + 000 + _ _ _ _ _ + 0010011
   which are imm[11:0], rs1, and rd
4. lw t5, 0(t6) => loads the 32 bits of instruction into t5 register
5. addi t5, t5, 0x80 => adds
   0000 0000 0000 0000 0000 0000 0100 0000
   to the instruction at "loop" -> counting the bits, it adds 1 to rd (so the instruction on the first iteration becomes "addi x1, x0, 0")
6. sw t5, 0(t6) => stores the updated instruction into memory, so it modifies the instruction at "loop" -- self modifying the code.
7. addi a0, a0, -1 => decreases x by 1
8. bnez a0, loop => if a0 == 0, continue. Otherwise, go back to the instruction at "loop"
9. ret => jump to ra...

So it modifies the registers by modifying the code at "loop", which in turn modifies the registers x0, x1, etc.

You are given the code above, and told that you can read and write to any word of memory without error. The function **`mystery`** lives somewhere in memory, but *not* at address **`0x0`**. Your system has no caches.

a) At a functional level, <u>in seven words or fewer</u>, what does **`mystery(x)`** do when **`x < 10`**?

Resets the first x registers.
Resets register number 0 through x-1.

b) One by one, what are the values of **a0** that **bnez** sees with **mystery(13)** at every iteration? We've done the first few for you. List no more than 13; if it sees fewer than 13, write N/A for the rest.

12, 11, 10, 9, 8, 7, 6, 5, 4, 3, -1, -2, -3

We're merrily rolling along, resetting all the registers, when we reset x10 = a0! But then "addi a0,a0,-1" makes it -1 so it actually never hits the stopping "branch equal to zero" case then! So the bnez sees -1, then -2, then -3 as the resetter continues along its merry way.

Reset register # on "nop" line	a0 before addi line	bnez sees a0 value
0	13	12
1	12	11
2	11	10
3	10	9
4	9	8
5	8	7
6	7	6
7	6	5
8	5	4
9	4	3
10	0	-1 (or 2^{32} - 1)
11	-1	-2 (or 2^{32} - 2)
12	-2	-3 (or 2^{32} - 3)

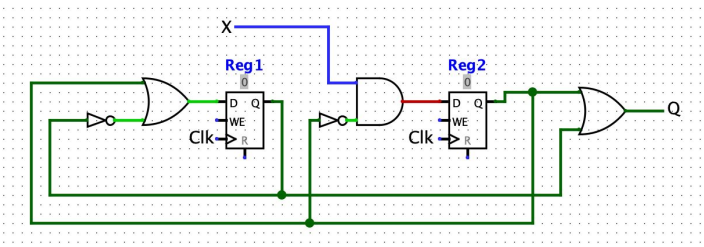
c) How many times is the **bnez** instruction seen when **mystery(33)** is called before it reaches **ret** (if it ever does)? If it's infinity, write ∞. **2^{32} + 10**

d) Briefly (two sentences max) explain your answer for part (c) above.

After we get through a0 resetting (and then skipping 0, the stopping condition), we continue resetting all the registers until we get to t5 (x30). Resetting it doesn't do anything since we clobber it anyway with the lw command. The next iteration, the "nop" line will reset t6. So when we lw t5 0(t6=0) we are loading the first word of memory. We are told this does not cause an error. Then we change it and write it back. We're no longer modifying our own program! So we continue to do this merrily until a0 runs down, which is 2^{32} total iterations (seems like forever, I know). So the total iterations is 2^{32} (after it was -1) and 10 more before that for 2^{32} + 10 iterations.

Q5) Watch the clock and don't delay! **(30 pts = 2*5 + 10 + 10)**

Consider the following circuit:



You are given the following information:

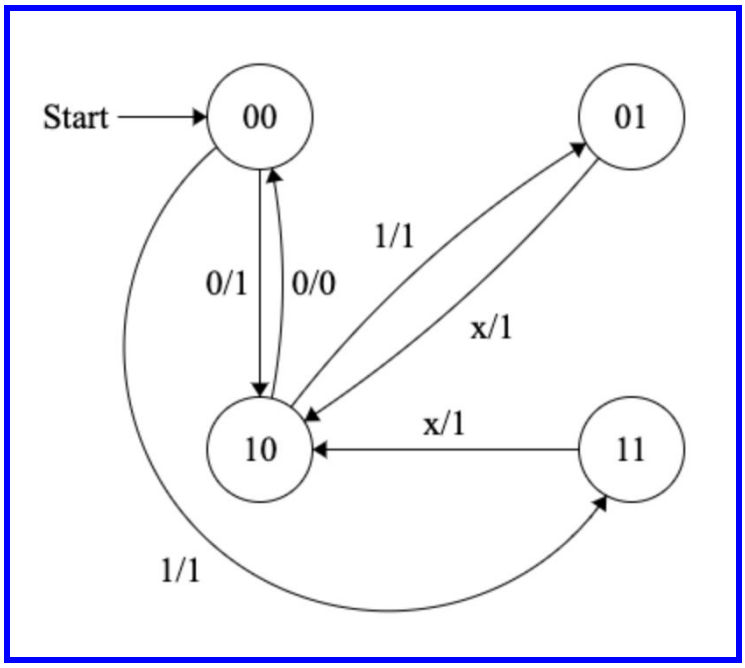
- Clk has a frequency of 50 MHz
- AND gates have a propagation delay of 2 ns
- NOT gates have a propagation delay of 4 ns
- OR gates have a propagation delay of 10 ns
- X changes 10ns after the rising edge of Clk
- Reg1 and Reg2 have a clock-to-Q delay of 2 ns

The clock period is 1/(50 * 10^6) s = 20 ns. This means that if X changes, it changes 10 ns after the clock positive edge.

SHOW YOUR WORK BELOW

a) What is the **longest possible setup time** such that there are no setup time violations? **4 ns**	Reg 1 longest possible setup time: the path is output of Reg1 -> NOT -> OR, with a delay of 2 ns + 4 ns + 10 ns = 16 ns. So 20 - 16 = **4 ns.** Reg 2 longest possible setup time: the path is X changes -> AND, with a delay of 10 ns + 2 ns = 12 ns. So 20 - 12 = **8 ns.** So longest setup time: min(4ns, 8ns) = **4ns**
b) What is the **longest possible hold time** such that there are no hold time violations? **8 ns**	Reg 1 longest possible hold time: the path is output of Reg2 -> OR, with a delay of 2 ns + 10 ns = 12 ns. Reg 2 longest possible hold time: the path is output of Reg2 -> NOT -> AND, with a delay of 2 ns + 4 ns + 2 ns = 8 ns. So longest hold time: min(12ns, 8ns) = **8ns**

c) Represent the circuit above using an equivalent FSM, where X is the input and Q is the output, with the state labels encoding Reg1Reg2 (e.g., "01" means Reg1=0 and Reg2=1). We did one transition already.



d) Draw the **FULLY SIMPLIFIED** (*fewest* primitive gates) circuit for the equation below into the diagram on the lower right. You may use the following primitive gates: AND, NAND, OR, NOR, XOR, XNOR, and NOT.

$out = \overline{(C + AB\overline{C} + \overline{B}\,\overline{C}D)} + \overline{(C + \overline{B + D})}$

SHOW YOUR WORK IN THIS BOX

out = $\overline{C}\ \overline{(AB\overline{C})}\ \overline{(\overline{B}\overline{C}D)} + \overline{C}(B + D)$ **(Demorgan's)**

out = $\overline{C}(\overline{A} + \overline{B} + C)(B + C + \overline{D}) + \overline{C}(B + D)$ **(Demorgan's)**

out = $(\overline{A}\,\overline{C} + \overline{B}\overline{C} + \overline{C}C)(B + C + \overline{D}) + B\overline{C} + \overline{C}D$ **(Distributive)**

out = $(\overline{A}\,\overline{C} + \overline{B}\overline{C})(B + C + \overline{D}) + B\overline{C} + \overline{C}D$ **(Inverse)**

out = $\overline{A}B\overline{C} + \overline{A}C\overline{C} + \overline{A}\,\overline{C}\,\overline{D} + B\overline{B}\overline{C} + BC\overline{C} + \overline{B}\,\overline{C}\,\overline{D} + B\overline{C} + \overline{C}D$ **(Distributive)**

out = $\overline{A}B\overline{C} + \overline{A}\,\overline{C}\,\overline{D} + \overline{B}\,\overline{C}\,\overline{D} + B\overline{C} + \overline{C}D$ **(Inverse)**

out = $B\overline{C}(A + 1) + \overline{A}\,\overline{C}\,\overline{D} + \overline{B}\,\overline{C}\,\overline{D} + \overline{C}D$ **(Distributive)**

out = $\overline{C}(B + \overline{A}\,\overline{D} + \overline{B}\overline{D} + D)$ **(Distributive)**

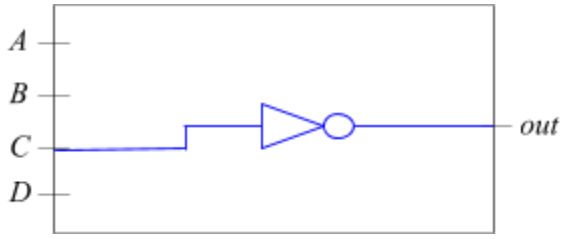
out = $\overline{C}((B + \overline{B}\,\overline{D}) + (\overline{A}\,\overline{D} + D))$ **(Associative)**

out = $\overline{C}(B + \overline{D} + \overline{A} + D)$

out = $\overline{C}(\overline{A} + B + (D + \overline{D}))$ **(Associative)**

out = $\overline{C}(\overline{A} + B + 1)$ **(Inverse)**

out = $\overline{C}$ **(Identity)**



Q6) RISCV Exam-isim Debug – Single Cycle **(12 pts = 2 x 6)**

For your CPU project, you followed the datapath diagram we gave you exactly and built a single-cycle CPU. However, something is not working correctly. *All instructions besides some of the I-types and SB-types are working.* You start by testing with an **`addi a0 a0 -3`** instruction. The **`a0`** register initially holds a value of **7** and all other registers initially hold **0**. This instruction is stored in IMEM at address **`0x00000004`**. DMEM reflects the initial IMEM. Undefined ImmSel outputs an I-type imm. You put a probe at the data read from IMEM and find the instruction is correct. You next put a probe at **wb**, and see the output is `0b0000_0000_0000_0000_0001_0000_0000_0100` (**`0x00001004`**).

a) Since the output is incorrect, what errors alone would cause the erroneous behavior? (select all that apply)

- ☐ The RegWEn is set to false.
- ■ The Immediate Generator is not sign extending.
- ☐ Read Reg1 and Read Reg2 are flipped.
- ☐ The writeback MUX is selecting DMEM.
- ☐ PCSel is set to PC + 4.
- ☐ The writeback MUX is selecting PC + 4.
- ☐ MemRW is set to write.
- ☐ BSel is selecting rs2 and not imm.

a) RegWEn does not change the state of wb since this is the only inst.
b) This is the issue!
c) If the two registers were flipped, we would get a different value since we would find we would be using a register with 0 in it thus would be getting back 0xFFC thus these are actually correct.
d) This is not correct since we would be getting the machine code of the current instruction since the correct address would be 0x4 which is this instruction. Thus this is not a root issue.
e) We would not be able to detect an issue here by looking at wb since this does not affect the wb of the current program.
f) If the write back mux was set to PC + 4, we would receive an output of 0x0000000c thus this is not correct either.
g) MemRW will not change the output of wb. Even if we were outputting the read of the address, it would be incorrect.
h) If BSEL selected RS2, we would have selected t6 which we know has a value of 0 and would have had 0 + 5 which is not the result we got.

You fix that issue. You then test **`beq x0 a0 label`** but something is still not working. This instruction is at address **`0xbfffff00`** and **`label`** is at address **`0xbfffff40`**. The register **`a0`** holds **0** and all other registers hold **1**. Assume that we get the correct instruction machine code for **`beq x0 a0 label`** when we probe it. You put a probe before the PC register and see this incorrect output: **`0xbfffff20`.**
Note: All other instruction types are working.

b) What errors alone would cause the erroneous behavior? (select all that apply)

- ☐ WBSel is incorrect.
- ☐ ImmSel is Incorrect.
- ☐ PCSel is set to PC + 4.
- ☐ There is an error in the Read Data1/rs1 wire.
- ■ The ImmGen is not correctly padding w/ extra 0.
- ☐ The inputs to the ASel MUX are flipped.
- ☐ The inputs to the BSel MUX are flipped.
- ☐ Read Reg1 and Read Reg2 are flipped.

a) We do not care about the WBsel since it does not change the input to the pc.
b) The immediate select seems to be correct as the other incorrect immediates are not equal to 0x20. (If you use the same bits of this instruction and interpret an immediate of other types, you will see none of them are 0x20).
c) We see the PC is not set to 0xbfffff04 thus PC + 4 was not taken.
d) The branch seems to be taken as we are not getting 0xbfffff04
e) Seems like the address is shifted left by one so this is an error.
f) Since we know all registers are 1, the output of the ALU is not possible if these are flipped.
g) Since we know the output of the ALU and that all registers are 1, we know BSel is correct.
h) Even if these were flipped, the branch should not be affected as we are doing an equality of the values.

Q7) *RISCV Exam-isim Debug – Pipelined* (18 pts = 3 + 6 + 3 + 6)

After solving your datapath bug, you decide to introduce the traditional five-stage pipeline into your processor. You find that your unit tests with single commands work for all instructions, and write some test patterns with multiple instructions. After running the test suite, the following cases fail. You should assume registers are initialized to 0, the error condition is calculated in the fetch stage, and no forwarding is currently implemented.

Case 1: Assume the address of an array with all different values is stored in `s0`.

```
addi  t0 x0 1
slli  t1 t0 2
add   t1 s0 t1
lw    t2 0(t1)
```

Each time you run this test, there is the same incorrect output for `t2`. All the commands work individually on the single-stage pipeline.

Pro tip: you shouldn't even need to understand what the code does to answer this.

a) What caused the failure? (select ONE)	**b)** How could you fix it? (select all that apply)
○ Control Hazard	■ Insert a nop 3 times if you detect this specific error condition
○ Structural Hazard	☐ Forward execute to write back if you detect this specific error condition
● Data Hazard	☐ Forward execute to memory if you detect this specific error condition
○ None of the above	■ Forward execute to execute if you detect this specific error condition
	☐ Flush the pipeline if you detect this specific error condition

The issue with the above code is the use of a register (aka we get the value during the decode phase) before we have written back the value of the previous instruction. This is a data hazard as the data which we want is not restored to the regfile. This means that we would have the current instruction in the execute phase while we have the previous in decode. This means that the next cycle, we would have to forward the execute output to the execute input to make sure the value is the correct, updated one. Inserting a nop when you realize this error happens will allow the system to do the write back. The other forwards in this problem are necessary for the given code above. Flushing the pipeline does not work as it means that we will no longer execute the instructions which were flushed. This means we would just drop instructions which would not get the correct value instead of just waiting till they can get the correct value.

Case 2: *After fixing that hazard,* the following case fails:

```
      addi  s0 x0 4
      slli  t1 s0 2
      bge   s0 x0 greater
      xori  t1 t1 -1
      addi  t1 t1 1
greater:
      mul t0 t1 s0
```

When this test case is run, `t0` contains `0xFFFFFFC0`, which is not what it should have been.
Pro tip: you shouldn't even need to understand what the code does to answer this.

c) What caused the failure? (select ONE)	**d)** How could you fix it? (select all that apply)
● Control Hazard	■ Insert a nop 3 times if you detect this specific error condition
○ Structural Hazard	☐ Forward execute to write back if you detect this specific error condition
○ Data Hazard	☐ Forward execute to memory if you detect this specific error condition
○ None of the above	☐ Forward execute to execute if you detect this specific error condition
	■ Flush the pipeline if you detect this specific error condition

The issue with the code above is we do not clear/flush the instructions if the branch determines it is taken. Remember that we are running on a five stage pipeline CPU which just assumes PC + 4 unless an instruction says otherwise. This means that we will not determine if the branch is taken until the branch is in the execute phase. This means that we will have the next two instructions already in the pipeline (one in instruction fetch, the other in instruction decode). So we have a Control Hazard as we are not executing the correct instructions. Some ways how to fix it: insert nops if you detect a branch instruction in the instruction fetch stage OR flush the pipeline if the branch is in the opposite direction of what was predicted. Forwarding data in this case will not help at all.

Some other notes about this. The [incorrect] value we got was -64. How did we get this? Well if we look at the code, we see that s0 will hold 4 and t1 will hold 16. Due to the control hazard, this means we will execute and write back the following two instructions before we execute the mul. This means that we will flip the bits (xori, note that -1 means all bits are 1 due to twos complement encoding) and add one (addi). This is just twos complement inversion! This is what causes us to get -16 which multiplied by 4 will give us -64. In a correct implementation, we would not have executed the xori and addi thus have gotten 64.

Q8) This is for all the money! (15 pts = 3 + 7 + 5)

Assume we have a single-level, 1 KiB direct-mapped L1 cache with 16-byte blocks. We have 4 GiB of memory. An integer is 4 bytes. The array is block-aligned.

```
#define LEN 2048

int ARRAY[LEN];
int main() {
    for (int i = 0; i < LEN - 256; i+=256) {
       ARRAY[i] = ARRAY[i] + ARRAY[i+1] + ARRAY[i+256];
       ARRAY[i] += 10;
    }
}
```

a) Calculate the number of tag, index, and offset **bits** in the L1 cache.

T:22	I:6	O:4

SHOW YOUR WORK

Offset: log2(block size) = log2(16) = 4
Index: log2(cache size / block size) = log2(1 KiB / 16) = log2(64) = 6
Tag:
First find total address bits log2(4 GiB) = log2(4 * 2^30) = log2(2^32) = 32
Then 32 - Index - Offset = 32 - 6 - 4 = 22

b) What is the hit rate for the code above? Assume C processes expressions left-to-right.

50%

SHOW YOUR WORK

Every iteration it's
ARRAY[i] read MISS
ARRAY[i+1] read HIT
ARRAY[i+256] read CONFLICT→ MISS
ARRAY[i] write CONFLICT→ MISS
ARRAY[i] read HIT
ARRAY[i] write HIT
3 MISSES, 3 HITS. 50% hit rate.

c) You decide to add an L2 cache to your system! You shrink your L1 cache, so it now takes 3 cycles to access L1. Since you have a larger L2 cache, it takes 50 cycles to access L2. The L1 cache has a hit rate of 25% while the L2 cache has a hit rate of 90%. It takes 500 cycles to access physical memory. What is the average memory access time in cycles?

78

SHOW YOUR WORK

AMAT = 3 + ¾ (50 + 1/10 500)
AMAT = 3 + ¾ (50 + 50)
AMAT = 3 + ¾ (100)
AMAT = 3 + 75
AMAT = 78

Q9) We've got VM! Where? **(15 pts = 2 + 3 + 5 + 5*1)**

Your system has a 32 TiB virtual address space with a single level page table. Each page is 256 KiB. On average, the probability of a TLB miss is 0.2 and the probability of a page fault is 0.002. The time to access the TLB is 5 cycles and the time to transfer a page to/from disk is 1,000,000 cycles. The physical address space is 4 GiB and it takes 500 cycles to access it. The system has an L1 physically indexed and tagged cache which takes 5 cycles to access and a hit rate of 50%. On a TLB miss, the MMU checks physical memory next.

a) How many bits is the Virtual Page Number?

27 bits

SHOW YOUR WORK

Number of reachable virtual addresses: log2(32 TiB) = 45
Bits needed to reach all addresses in a page: log2(256 KiB) = 18
So the virtual page number bits are: 45 - 18 = 27

b) What is the total size of the page table (in **bits**), assuming we have **no** permission bits or any other metadata in a page table entry, just the translation?

14×2^{27} bits

SHOW YOUR WORK

We need to figure out the number of bits in the physical page number. It is the same method except we use the physical address space:
Number of reachable physical addresses: log2(4 GiB) = 32
So PPN size is 32 - 18 = 14. We do not have any metadata bits so the total number of bits in a PTE is 14. To figure out how many entries we need, we need to look at the total number of virtual page numbers we have = 27. This means we need 2^{27} entries in the page table. This means we need a total of 14×2^{27} bits in our page table.

c) What is the average memory access time (in cycles) for a single memory access for the current process? Assume the page table is resident in DRAM.

760 cycles

SHOW YOUR WORK

Translation AMAT = 5 + ⅛(500 + 2/1000(1M))
= 5 + ⅛(500 + 2000)
= 5 + ⅛(2500)
= 5 + 500
= 505
plus
Data access AMAT = 5 + 50% (500)
= 5 + 250
= 255
AMAT (overall) = 505 + 255 = 760

d) Which of the following, if any, **must be done** when we switch to a different process? Do **not** select any option that is unnecessary.

		Yes	No
1)	Update page table address register	●	○
2)	Evict pages for the previous process from RAM	○	●
3)	Clear TLB dirty bits	○	●
4)	Clear cache valid bits	○	●
5)	Clear TLB valid bits	●	○

Q10) Parallelism and Potpourri (30 pts = 6 + 4 + 3 + 3 + 3 + 3 + 4 + 4)

a) What are all the possible values of `x` and `y` after execution has completed if the code were run on two cores concurrently?

`x`: ☐ 0 ☐ 1 ■ 2 ■ 3 ☐ 4
☐ 5 ☐ 6 ☐ 7 ☐ 8 ☐ 9

`y`: ☐ 0 ☐ 1 ☐ 2 ■ 3 ■ 4
■ 5 ■ 6 ■ 7 ☐ 8 ☐ 9

```
int x = 1;
int y = 1;
#pragma omp parallel
{
   x += 1;
   y = x + y;
}
```

SHOW YOUR WORK

x: Both cores could see x = 1 for the first line -> x will end up as 2; otherwise, one core will set x = 2 first, and the other core will set x = 3

y: The key point here is that the two cores are updating shared, global variables -- these updates can occur throughout execution! Let's call core i's execution of line j "ij" -> this results in 4 lines executed (11, 12, 21, 22). Execution order (core 1 and 2 could be flipped for equivalent results):
(11, 21 together), (12, 22 together): y -> 3
(11, 21 together), 12, 22: y -> 5
11, 21, (12, 22 together): y -> 4
11, 12, 21, 22: y -> 6
11, 21, 12, 22: y -> 7

Another way to look at this is to realize for each thread when it executes the line "y = x + y", x equals either 2 or 3. Since each thread adds either +2 or +3 to y, both threads together add +4, +5, or +6.

Additionally, we know that if both threads read y before writing back to it, only one thread will actually properly increment it, giving us +2 or +3. .

b) A Job involves four Tasks, and the % of time spent in each Task is shown in the table. If we buy accelerators that speed up `f` by 2x and `k` by 8x what's your *total speedup*?

Task	%
f	10% → 2x
g	4%
h	6%
k	80% → 8x

4x

SHOW YOUR WORK

Old: 10+4+6+80
New: 5+4+6+10
Old/New = 100/25 = 4x

c) Which of the following were discussed in the MapReduce lecture? (select all that apply)

☐ Workers specialize: A "map" worker that finishes their map task early are *only* given a new map task.
Map workers can be reassigned map or reduce tasks

■ The system automatically reassigns tasks if a worker "dies", providing automatic fault-tolerance.

☐ MapReduce was specifically designed for custom high-end machines and custom high-end networks.
Commodity hardware and networks

☐ Hadoop was better than Spark, since Hadoop offered better performance, lazy evaluation, and interactivity.
vice-versa

d) Virtual memory allows us to: (select all that apply)

■ Pretend that programs do not have to share the address space with other programs.

■ Have more stable and secure computer systems.

☐ Divide the entire address space into 4 sections specifically for static, code, heap, and stack.
Nope, that can happen with or without VM

☐ Provide the illusion that the computer has access to storage the size of DRAM but at the speed of disk.
vice-versa

e) Which of the following were discussed in Prof. David Patterson's lecture? (select all that apply)

■ The power demands of machine learning are growing 10x per year; Moore's Law is only 10x in 5 years.

☐ The Tensor Processing Unit has a similar performance per watt to a CPU on production applications.
TPU blows away CPU on performance/watt

☐ The marketplace has decided: Open ISAs (e.g., RISC-V) are better than proprietary ones (e.g., ARM).
Marketplace /will/ decide, it hasn't yet (as it has for RISC vs CISC)

☐ Domain Specific Architectures achieve incredible performance but just for one application, like ASICs.
For one /domain/, there are many applications in one domain

f) Which of the following were discussed in James Percy's GPU lecture? (select all that apply)

☐ A square is the base shape used when rendering scenes.
Triangle (since it's planar)

☐ The GPU achieves its speed because all of the threads run different programs on the same data.
The same program on different data (like map square over a list, or 3D project all these points)

☐ A GPU has many more cores than a CPU and operates at a higher frequency.
More cores, but lower frequency (otherwise we couldn't cool it!)

■ When pixels along polygon edges are different between new generations of GPUs, the team investigates it.
Yep, what was it that changed?

g) You have an SSD which can transfer data in 32-byte chunks at a rate of 64 MB/second. No transfer can be missed. If we have a 4GHz processor, which takes 200 cycles for a polling operation, what fraction of time does the processor spend polling the SSD drive for data? Leave your answer in the box provided as a percentage.

10%	**SHOW YOUR WORK** 64 MB/sec / 32 B/poll = 2M polls/sec 2M polls/sec * 200 cycles/poll = 400M cycles/sec 4 GHz clock ⇒ 4000M cycles/sec; 400M c/s / 4000M c/s = 10%

h) You are designing a 64-bit ISA for a simplified CPU with 3 bit-fields: `immediate | register | opcode`. You reserve enough of the rightmost bits to handle 1,500 opcodes, and enough of the leftmost bits to encode unsigned numbers up to 500 trillion. What's the greatest number of registers can you have?

16	**SHOW YOUR WORK** 1500 opcodes requires 11 bits (2048) 500 trillion requires 49 bits (512 Tebi) 64 - (11 + 49) = 4 bits for registers 2^4=16