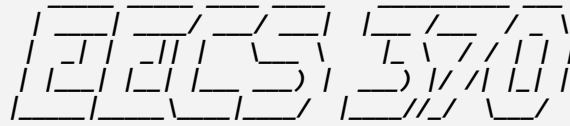


Midterm Exam



EECS 370 Winter 2024: Introduction to Computer Organization

You are to abide by the University of Michigan College of Engineering Honor Code. Please sign below to signify that you have kept the honor code pledge:

***I have neither given nor received aid on this exam,
nor have I concealed any violations of the Honor Code.***

Signature: _____

Name: _____ *Answer Key* _____

Uniquename: _____

Uniquename of person sitting to your **Right**
(Write ⊥ if you are at the end of the row) _____

Uniquename of person sitting to your **Left**
(Write ⊥ if you are at the end of the row) _____

Exam Directions:

- You have **120 minutes** to complete the exam. There are 7 questions in the exam on 18 pages (double-sided). **Please flip through your exam to ensure you have all pages.**
- Write legibly and dark enough for the scanners to read your answers.
- **Write your uniquename on the line provided at the top of each page.**

Exam Materials:

- You are allotted **one 8.5 x 11 double-sided** note sheet to bring into the exam room.
- You are allowed to use calculators that do not have an internet connection. All other electronic devices, such as cell phones or anything or calculators with an internet connection, are strictly forbidden.

Problem	1	2	3	4	5	6	7
Point Value	10	13	10	14	14	16	23

Problem 1: Multiple Choice

10 points

Completely shade in the boxes with the best answer. Select only 1 answer unless specified otherwise. [1 point each] **Checked boxes indicate the correct answer(s) (ignore cross outs)**

1. Which of the following processor components can be implemented without the need for sequential logic (as defined in lecture)? **(FILL IN ALL THAT APPLY)**

- ☒ Mux
- ☐ Program counter
- ☒ Full-adder
- ☒ Decoder
- ☒ Control ROM
- ☒ AND-gate
- ☐ Register

2. An error about an unresolved global symbol would most likely occur during the _____ phase.

- ☐ Compiling
- ☐ Assembling
- ☒ Linking
- ☐ Loading

3. What does ISA stand for?

- ☐ Instruction Set Analysis
- ☒ Instruction Set Architecture
- ☐ Integrated System Architecture
- ☐ Input/Output System Architecture

4. Writing assembly programs in a _____-type ISA typically involves writing more, simpler instructions, rather than a smaller number of sophisticated instructions.

- ☐ CISC
- ☒ RISC
- ☐ Multi-cycle
- ☐ Single-cycle

5. Which of the following is the best description of Dennard Scaling (regardless of whether Dennard Scaling is holding or not)?

- ☒ As an individual transistor gets smaller, it consumes a smaller amount of power
- ☐ As an individual transistor gets smaller, it consumes a constant amount of power
- ☐ As an individual transistor gets smaller, it consumes a greater amount of power

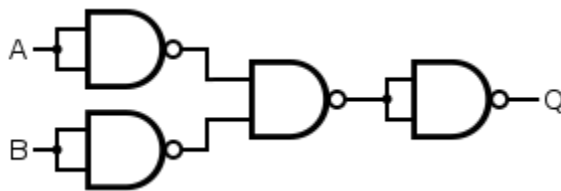
6. Which of the following represents -4 as a 5-bit 2's complement binary number?

- ☐ 00100
- ☐ 00101
- ☐ 11011
- ☒ 11100
- ☐ 11101
- ☐ None of the above

7. ARM has what kind of addressability?

- ☒ Byte
- ☐ Word
- ☐ Big endian
- ☐ Little endian
- ☐ It can be configured by the programmer

8. What inputs will cause the circuit below to output a 0? (FILL IN ALL THAT APPLY)



- ☐ A=0, B=0
- ☒ A=0, B=1
- ☒ A=1, B=0
- ☒ A=1, B=1

9. If the program counter in an LC2K machine points to a line corresponding to a ".fill X" directive, the program will:

- ☐ immediately crash
- ☒ ~~execute whatever LC2K instruction assembles to the value of X, and have undefined behavior for any value that does not correspond to a valid instruction~~
- ☐ write the value of X into memory
- ☐ perform an add instruction

10. Moving from single-cycle to a multi-cycle implementation is expected to significantly: (FILL IN ALL THAT APPLY):

- ☒ ~~Decrease the clock period~~
- ☐ Decrease the latency to execute the slowest instruction
- ☐ Decrease the CPI
- ☐ Increase the total number of instructions executed to completion for a given program

Problem 2: Short Answers

13 points

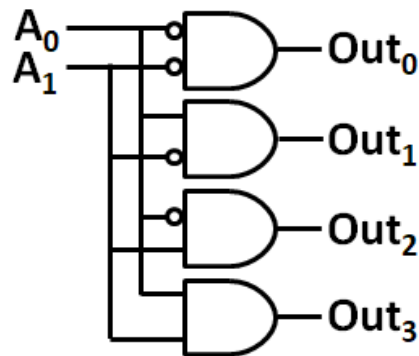
1. Convert each of the 8-bit hexadecimal numbers into its binary form and decimal form (both for treating the original number as a signed (two's-complement) and an unsigned value). [3 points] *Each cell is graded independently, so getting the binary form wrong may result in multiple lost points.*

Hexadecimal	Binary	Decimal (signed)	Decimal (unsigned)
0x8C	10001100	-116	140
0xB2	10110010	-78	178

2. Place an 'X' in **each** cell to indicate that the LC2K instruction **ALWAYS** performs the corresponding action during some point during its cycle on our single-cycle processor implementation (assume the program has fewer than 1000 instructions). [4 points]

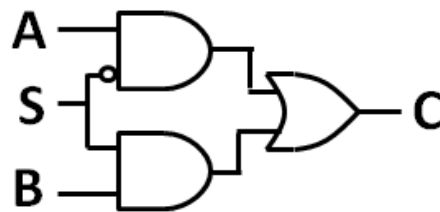
Inst / Action	Reads memory	Calculates an address using an immediate	Reads from register file	Increments PC to a larger value
add / nor	X		X	X
lw	X	X	X	X
sw	X	X	X	X
beq	X	X	X	
jalr	X		X	

3. What combinational logic element discussed in class does the following circuit implement?
Briefly (in 20 words or fewer) explain its behavior. [3 points]



Decoder - the circuit selects a single output line corresponding to whichever binary number is passed in.

4. What combinational logic element discussed in class does the following circuit implement?
Briefly (in 20 words or fewer) explain its behavior. [3 points]



Mux - S chooses where A or B is passed to C .

Problem 3: Time to Run**10 points**

Eldon Tyrell is tasked with building a processor that implements the new BC3K (“Blade Computer 3000”) ISA. This processor will be used to run a BC3K program called **multitude**, which does numerical computation on integers in loops. Eldon decides to build both a single-cycle and a multi-cycle implementation of BC3K and see which one of them runs **multitude** better. The clock period and latency of individual instruction types for Eldon’s two implementations is shown below.

Single cycle design:

- Clock period of 10ns
- Cycles per instruction:

Instruction	Cycles
All instructions	1 cycle

Multi-cycle design:

- Clock period of 2ns
- Cycles per instruction:

Instruction	Cycles
Add	3 cycles
Branch	4 cycles
Load and store	5 cycles
Multiply	8 cycles

The **multitude** program consists of 10,000 instructions with the following distribution:

Instruction	Percentage
Add	20%
Branch	10%
Load and store	20%
Multiply	50%

1. What is the execution time of the **multitude** program on the single-cycle design? [2 points]

$$10,000 * 10 \text{ ns} = 100,000 \text{ ns (100 us)}$$

2. What is the execution time of the **multitude** program on the multi-cycle design? [3 points]

$$3*.2 + 4*.1 + 5*.2 + 8*.5) * 10000 * 2 = 120,000 \text{ ns (120 us)}$$

3. Eldon believes that he can make the execution time of the **multitude** program on the multi-cycle design equivalent to the execution time of **multitude** on the single-cycle design by modifying the **multitude** program. Specifically, many of the multiplications are very simple and can be replaced with a single add instruction (i.e. $\text{var} * 2$ can be replaced with $\text{var} + \text{var}$). Assume that it is possible to replace multiply instructions in **multitude** with add instructions and still have **multitude** compute the correct result. Also assume that the percentage of load, store, and branch instructions in the overall program does not change.

Compute the percentage of multiply instructions that the revised **multitude** program will need to have for the two designs to have the same execution time. [5 points]

$$10 = (.1*4 + .2*5 + x*8 + (.7-x)*3) * 2 = (.4 + 1 + 2.1 + 5x)*2 = 7 + 10x$$

$$10x = 3$$

$$x = .3 \text{ or } 30\%$$

$$\text{or } .3/.5 = 60\% \text{ if you interpreted it as "\% of original multiplications"}$$

Problem 4: Caller/Callee Save

14 points

Count caller/callee saves for the code shown when foo() is called once. Assume the compiler checks for liveness across function calls **but does not make any other optimizations**.

<pre> void foo(void){ int a=2,b=4,c=8; bar(); a = c + b; b = c - b; for(b=0;b<4;b++){ bar(); a += b; } a = 100; bar(); c = a + b; bar(); return; } </pre>	<pre> void bar(void){ int j=0,k=1,l=4; int m=2; printf("Binary Fun\n"); j = 2 * k; while(j<4){ printf("Hello %d\n",j); j++; } j = k - l; l = j + m; return; } </pre>
----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Determine the number of load/store pairs required for each variable in each case in the table below.

Variables	Caller loads/stores	Callee loads/stores
a	5	1
b	6	1
c	1	1
j	7*2=14	7
k	7*3=21	7
l	7*3=21	7
m	7*3=21	7

Problem 5: Assembling and Linking

14 points

1. Fill in the rest of the symbol and relocation tables for the following two C functions (you may not need all lines). Assume all variable reads involve a load, and all variable writes involve a load and a store. [8 points]

foo.c

```

1. extern int glop();
2. int y = 10;
3. int x = 5;
4.
5. int main() {
6.     int tmp = glop() + y;
7.     return tmp;
8. }
```

bar.c

```

1. extern int x;
2.
3. int glop() {
4.     x = x + 1;
5.     return x;
6. }
```

T: Text D: Data U: Undefined**Instructions: ldur (load), stur (store), bl (branch)**

foo.c symbol table

Symbol	Type (T/D/U)
glop	U
y	D
x	D
main	T

bar.c symbol table

Symbol	Type (T/D/U)
x	U
glop	T

foo.c relocation table

Line #	Symbol	Instruction
6	glop	bl
6	y	ldur

bar.c relocation table

Line #	Symbol	Instruction
4	x	ldur
4	x	stur
5	x	ldur

Let us say the “`int x = 5;`” was moved inside of `main()` in `foo.c`.

2. Would `foo.c`'s and/or `bar.c`'s symbol and/or relocation tables change? If so, how? *[3 points]*

Yes - `foo.c`'s symbol table no longer has `x`. Relocation table still does not have anything with `X`. `bar.c` stays the same.

3. Would the linker report an error after the change in the previous question, assuming `foo.c` and `bar.c` are the only code files? If yes, what is the error? If no, why not? *[3 points]*

Yes, unresolved symbol `X`.

Problem 6: ARM Assembly

16 points

Fill in the blanks to complete the following C-to-LEGV8 assembly conversion. You may not define any other labels. Assume we are following the ARM ABI discussed in class (see reference material), and assume that the call stack **grows downwards** in memory as items are placed on the stack (i.e. memory addresses get smaller as more items are added to the stack). Remember that LEGV8 is a 64-bit system. **Each entry is graded independently. Some errors might compound into multiple penalties applied.**

```
#include <stdlib.h>
void other_func();

struct my_struct {
    short small[2];
    int medium;
};

int my_func(struct my_struct
            *input) {
    int temp = input[1].medium;
    temp++;
    if(temp < 10)
        other_func();
    return temp;
}
```

```
my_func:
    stur    x30, [x28, -8]
    stur    x19, [x28, -16]
    subi    x28, x28, #__16__
    ldursw  x19, [__x0__, #12]
    addi    x19, x19, #1
    __cmp (or cmpi) x19, #10__
    b.lt    .if
    b (or b.ge)__ .endif

.if:
    bl____ other_func

.endif:
    mov     __x0__, x19
    addi    x28, x28, #__16__
    ldur    __x19__, [x28, -16]
    ldur    __x30__, [x28, -8]
    __br x30_____
```

Problem 7: New ISA

23 points

A simple alternative to the “register-set architecture” we have discussed in class is the “accumulator” architecture, in which a single register called an “accumulator” is used to hold operands. Because there is only one accumulator, it can be referenced implicitly without the need for index bits and thus instructions can be kept very small. Let’s consider an 8-bit version of LC2K which makes use of an accumulator: “LC-ACC”. A subset of the instructions (all of which are “I-type” instructions with 5-bit signed immediates specified in decimal and encoded in two’s-complement using bits 0-4) are below:

Instr	Opcode [bits 7-5]	Functionality
LOAD	000	Loads single byte from memory address [#imm] and writes it to the accumulator
STORE	001	Stores single byte from the accumulator and writes it to memory address [#imm]
NAND	010	Computes the bitwise NAND of the data in the accumulator and data in memory address [#imm] and writes the result into the accumulator
BZ	011	Sets PC to PC+[#imm] if and only if accumulator holds the value 0

All instructions should be written in the form "[opcode] [#imm]" (without brackets).

Additional notes:

- The accumulator, PC, memory addresses, and memory values are each 8-bits wide.
- Memory is byte-addressable
- Negative addresses are undefined (you don’t need to worry about what happens with these)

1. What is the inclusive-range of values that can be specified in the immediate values of these instructions? Write your answer in decimal. *[1 point]*

-16 to 15

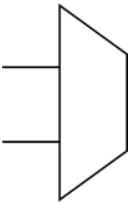
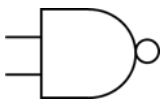
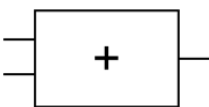
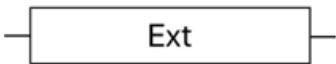
2. Implement the following code in LC-ACC assembly using 5 instructions or fewer. You can assume a, b, and c are already initialized at their respective addresses. You should assume all other memory addresses are in use and should not be overwritten. *[4 points]*

C-code	Assembly
<pre>uint8_t a; // address 8 uint8_t b; // address 9 uint8_t c; // address 10 c = a & b;</pre>	<pre>LOAD 8 NAND 9 STORE 10 NAND 10 STORE 10</pre>

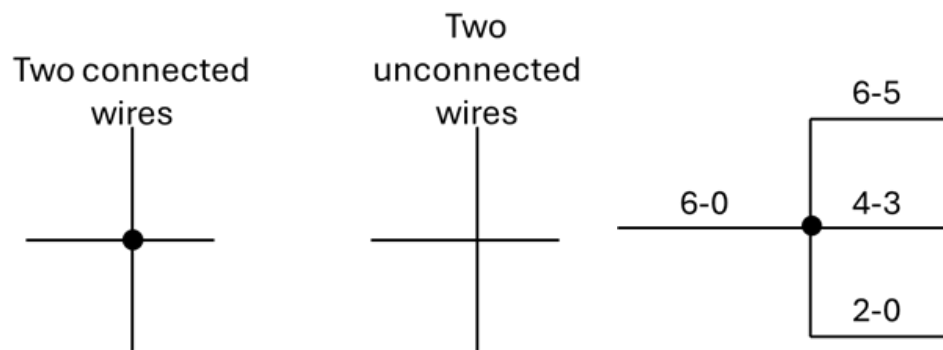
3. With the same assumptions, implement the below in 4 instructions or fewer. You may not use any labels. *[3 points]*

C-code	Assembly
<pre>uint8_t a; // address 8 uint8_t b; // address 9 uint8_t c; // address 10 if(a != 0) b = a c = a</pre>	<pre>LOAD 8 BZ 2 STORE 9 STORE 10 Alternate: LOAD 8 STORE 10 BZ 2 STORE 9</pre>

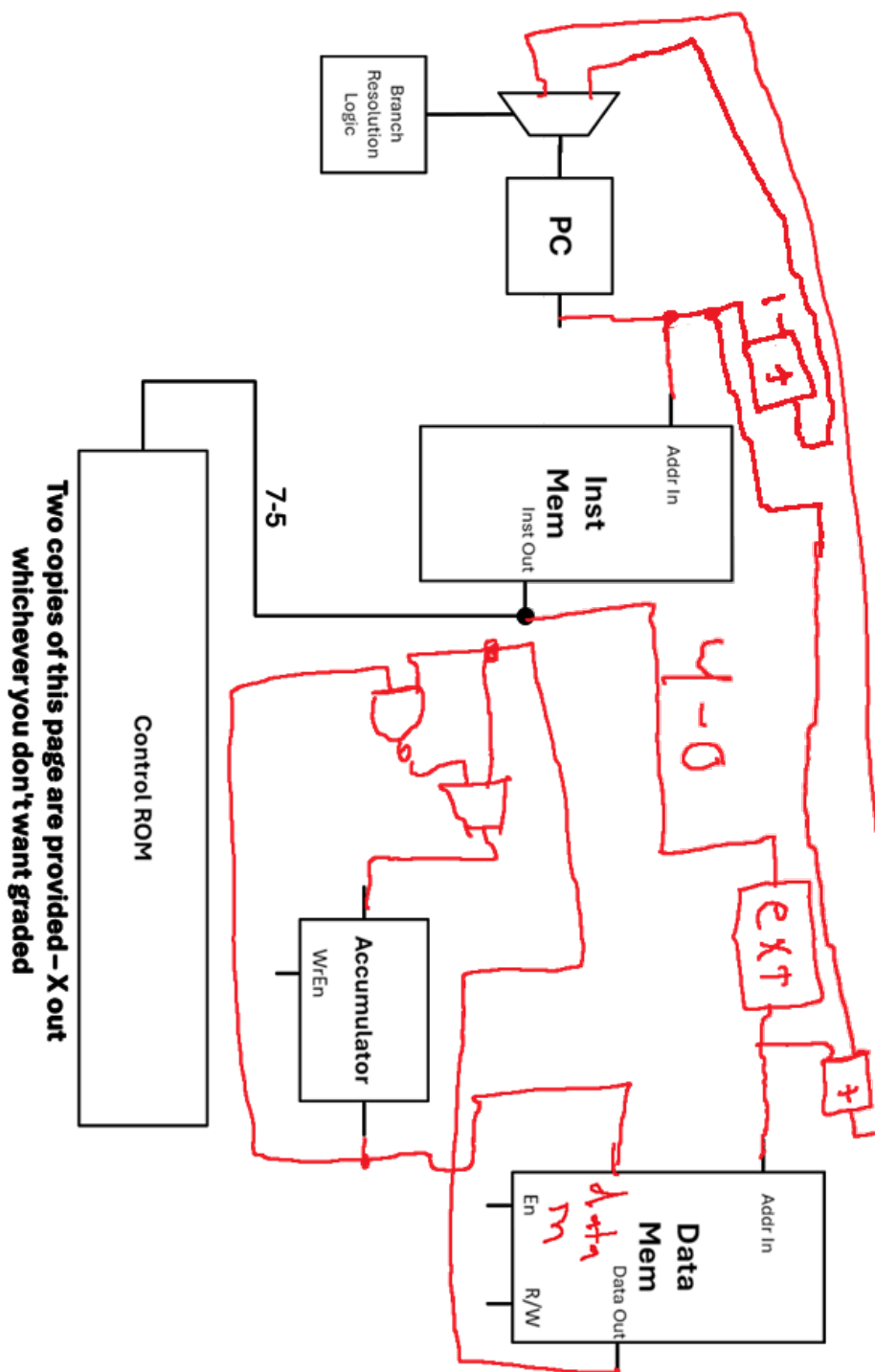
4. On the next page, fill in the rest of the diagram for the datapath of a single-cycle implementation of LC-ACC. You do not need to account for any other possible instructions in this ISA. You may use any number of the logic elements below as well as hardcoded values, but nothing else. [10 points]

2-to-1 Mux	2-input NAND Gates	8-bit full Adder	Sign-Extension Unit
			

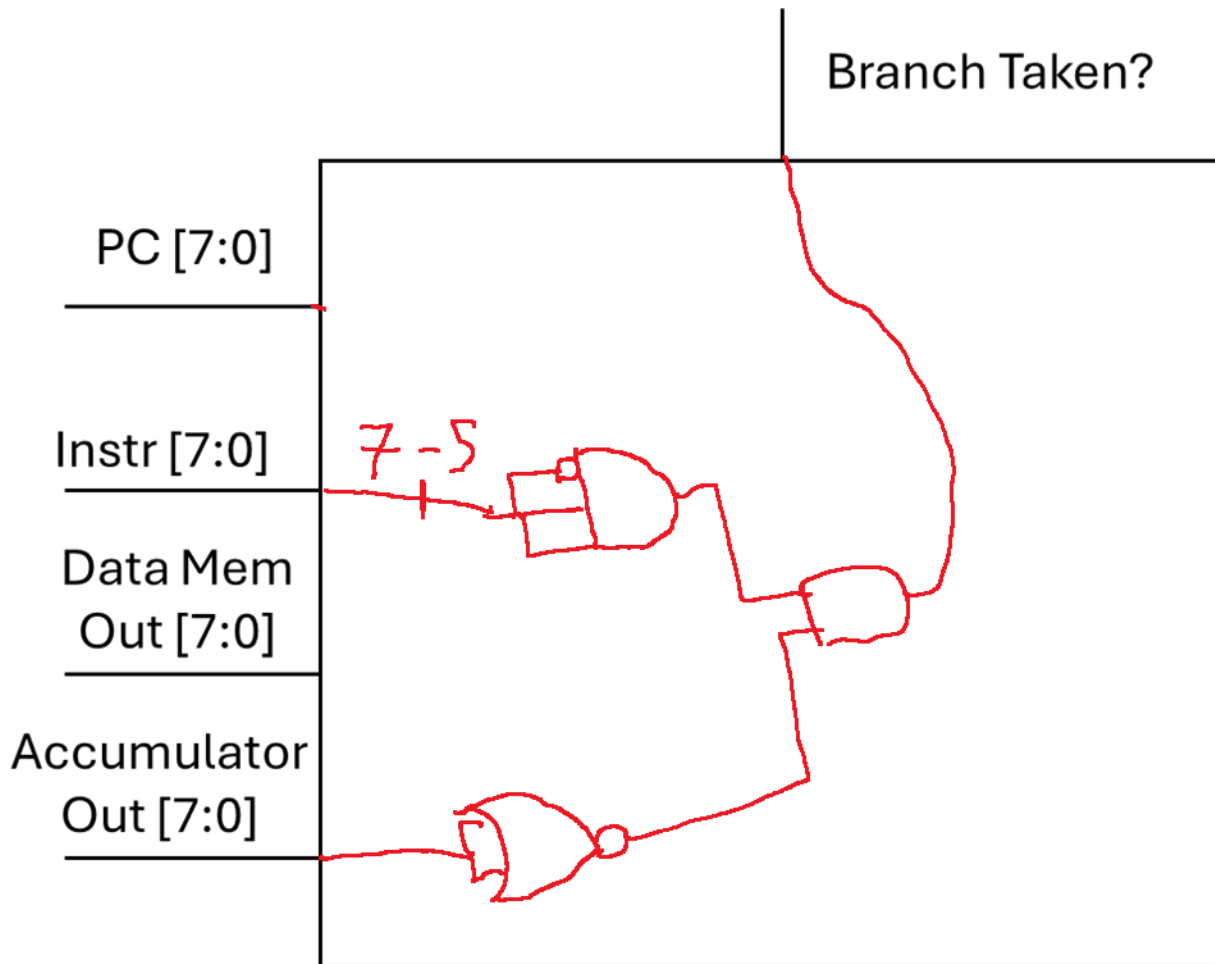
- Place a clear dot on any crossing lines which should be connected. Crossing lines without a dot will be assumed to be unconnected. In any instance where a multi-bit signal is split into different groups of signals, you **must label which bits are which** by placing a number range above each line. E.g.



- You may assume any signals which can be controlled by the control ROM are connected without drawing them and that they work correctly (i.e. you do not need to design the control ROM). The ROM only takes the current opcode as input
- The branch resolution logic, which controls the select input(s) for the PC input, has been added for you. You will be asked to implement this in the next part, but for here **you can assume it works correctly**. It outputs a 1 if and only if a branch is taken. You do not need to show the inputs to this unit for this part.



5. Implement the **Branch Resolution Logic** using only AND, OR, (each of which can have any number of inputs), NOT gates, and hardcoded 0s and 1s. The output should be 1 if and only if a branch is to be taken. A set of available input signals has been provided but you may not need them all. Leave any unused inputs unconnected, and label any subset of bits used. [5 points]



As a reminder, here is a description of the branch instruction

Instr	Opcode [bits 7-5]	Functionality
BZ	011	Sets PC to PC+[#imm] if and only if accumulator holds the value 0

Uniqname: _____

Page 17 of 18

THIS PAGE INTENTIONALLY LEFT BLANK