CS 61C Spring 2026 — Yan Midterm

Solutions last updated: Monday, March 30, 2026

PRINT Your Name: _____

PRINT Your Student ID: _____

PRINT the Name and Student ID of the person to your left: _____

PRINT the Name and Student ID of the person to your right: _____

PRINT the Name and Student ID of the person in front of you: _____

PRINT the Name and Student ID of the person behind you: _____

You have 110 minutes. There are 8 questions of varying credit. (100 points total)

Question:	1	2	3	4	5	6	7	8	Total
Points:	13	10	21	16	20	10	10	0	100

For questions with **circular bubbles**, you may select only one choice.

- ○ Unselected option (Completely unfilled)
- ◎ Don't do this (it will be graded as incorrect)
- ● Only one selected option (completely filled)

For questions with **square checkboxes**, you may select one or more choices.

- ☐ You can select
- ☐ multiple squares
- ☑ (Don't do this)

Anything you write outside the answer boxes or you ~~cross out~~ will not be graded. If you write multiple answers, your answer is ambiguous, or the bubble/checkbox is not entirely filled in, we will grade the worst interpretation. For coding questions with blanks, you may write at most one statement per blank and you may not use more blanks than provided.

If an answer requires hex input, you must only use capitalized letters (`0xB0BACAFE` instead of `0xb0bacafe`). For hex and binary, please include prefixes in your answers unless otherwise specified, and do not truncate any leading 0's. For all other bases, do not add any prefixes or suffixes.

As a member of the UC Berkeley community, I act with honesty, integrity, and respect for others. I will follow the rules of this exam.

Acknowledge that you have read and agree to the honor code above and sign your name below:

_____

Q1 *61Crazy Potpourri* 🤯 **(13 points)**

Q1.1 (3 points) Translate the RISC-V instruction in Line 1 to 32-bit hexadecimal machine code.

```
1  bge a6 t5 fun
2  addi a6 x0 1
3  fun:
4     lw a6 0(s5)
5     sw a6 0(s4)
```

0x01E85463

Q1.2 (3 points) For this question, suppose J-type instructions use a `5`-bit **unsigned** immediate stored as `imm[6:2]` in machine code, with an implied `0` as the `0`th and `1`st bits. What is the range of target addresses for jump instructions, inclusive? You must express your answers as integers.

[PC − 0 , PC + 124]

Q1.3 (2 points) For this question, you may assume we are using the standard RISC-V 32I ISA from lecture. Consider the following RISC-V code below.

```
1  # CODE OMITTED
2  beq x0 x0 label
3  # CODE OMITTED
```

`label` can be either above or below the `beq` instruction. Which of the following is a possible PC-relative offset to branch from the beq instruction to `label`?

- ○ `-5286`
- ○ `-205`
- ● `1404`
- ○ `6002`
- ○ `-2342`
- ○ `602`
- ○ `4217`
- ○ None of the above

Q1.4 (3 points) Suppose we execute `lui a0 imm`, where `imm` is some arbitrary immediate. What is the largest possible value `a0` can hold after this instruction is run, interpreted as a signed integer? You may express your answer in terms of powers of two (e.g. $2^{10} - 2^{17}$).

$2^{31} - 2^{12}$

For each of the following questions, select the step of CALL that performs the translation.

Q1.5 (1 point) Responsible for translating the `li a0 0xB0BACAFE` instructions.

- ○ Compiler
- ● Assembler
- ○ Linker
- ○ Loader
- ○ None of the above

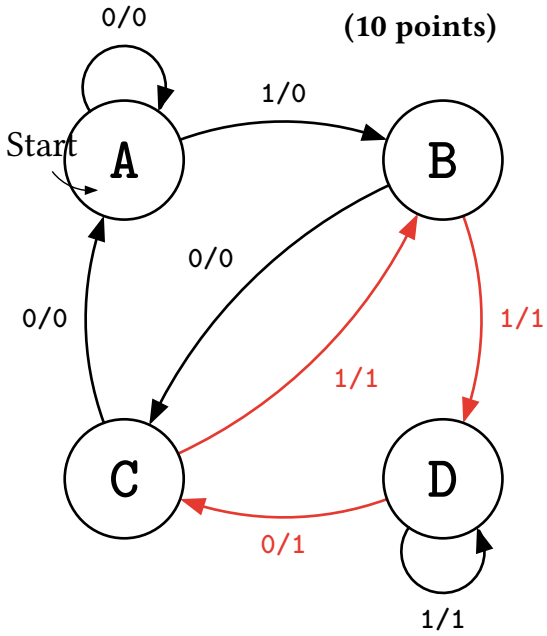
Q1.6 (1 point) What steps of CALL does the program `gcc` perform? Select all that apply.

- ■ Compiler
- ■ Assembler
- ■ Linker
- □ Loader
- ○ None of the above

Q2 *61Camera* 📸 (10 points)

Felicity has been stealing deliveries from Yogurt Park! In order to catch her in the act, Chloe has set up a surveillance camera that **records** if motion has occurred in at least 2 of the last 3 seconds.

Chloe has **partially** implemented the camera FSM, shown right. The table below describes the input and output signals. One transition occurs every second.



		Behavior
In	0	No motion detected
	1	Motion detected
Out	0	Camera does not record
	1	Camera records

Q2.1 (2 points) What is the minimum number of cycles for the camera to start recording and stop recording twice (start→stop→start→stop)? Give your answer as an integer.

5

Q2.2 (7 points) Fill in the remaining state transitions describing this FSM.

Curr State	Input	Next State	Output
A	● 0 ○ 1	● A ○ B ○ C ○ D	● 0 ○ 1
A	○ 0 ● 1	○ A ● B ○ C ○ D	● 0 ○ 1
B	● 0 ○ 1	○ A ○ B ● C ○ D	● 0 ○ 1
B	○ 0 ● 1	○ A ○ B ○ C ● D	○ 0 ● 1
C	● 0 ○ 1	● A ○ B ○ C ○ D	● 0 ○ 1
C	○ 0 ● 1	○ A ● B ○ C ○ D	○ 0 ● 1
D	○ 0 ● 1	○ A ○ B ○ C ● D	○ 0 ● 1
D	● 0 ○ 1	○ A ○ B ● C ○ D	○ 0 ● 1

Q2.3 (1 point) In 10 words or less, what does State C represent?

Motion last detected exactly two cycles ago.

Q3 *61C(Q)uilt* (21 points)

Carolann is designing a Cal quilt using C! A quilt pattern is a rectangular grid stored in the pattern struct `pattern_t`, shown right. Each string in the array `patches` is a row of the quilt.

- Each string in `patches` is null-terminated and composed only of the characters: `'B'` (blue), `'G'` (gold), and `'W'` (white);
- `num_cols` is the number of patches per row;
- `num_rows` is the number of rows in the quilt.

```
1 typedef struct {
2   char **patches;
3   uint32_t num_cols, num_rows;
4 } pattern_t;
```

BWBBWG	BWBBW	BWBW
BGWBWB	GBGWB	GBGWB
	WBWWW	WB
Valid	*Valid*	*Invalid*

Figure 1: Valid quilt patterns are rectangular.

Q3.1 (2 points) What is `sizeof(pattern_t)` on a 64-bit system, where all pointers are word-aligned?

16 bytes

`pattern_t` is a valid quilt pattern if all strings in `patches` have length `num_cols` (Figure 1). Implement the `is_valid` function, which returns `true` if the pattern pointed to by `pat` is valid, and `false` otherwise.

Hint: See the reference provided for the C standard library function `strchr`.

```
bool is_valid(pattern_t *pat) {
    for (int i = 0; i < pat->num_rows; i++) {
        char *null_term = strchr(pat->patches[i], '\0');
        if ( ____[Q2.2]____ ) { return ____[Q2.3]____; }
    }
    return ____[Q2.4]____ ;
}
```

Q3.2 (2 points) Fill in Blank 2.2. **Your expression cannot include any C standard library functions.**

null_term - pat->patches[i] != pat->num_cols

Q3.3 (1 point) Fill in Blank 2.3. ○ `true` ● `false` ○ `NULL`

Q3.4 (1 point) Fill in Blank 2.4. ● `true` ○ `false` ○ `NULL`

The `init_pattern` function below is used on the next page.

`init_pattern`: Return a pointer to a heap-allocated `pattern_t` struct. Sets `num_rows` and `num_cols` appropriately and initializes `patches` to a pointer to an array of `num_rows` heap-allocated strings, each of which is `num_cols + 1` bytes.		
Arguments	`uint32_t num_rows`	Number of quilt rows, i.e., number of elements in `patches`
	`uint32_t num_cols`	Number of patches in each row.
Return value	`pattern_t *`	A pointer to heap-allocated `pattern_t`.

(15 points) Implement the `make_pattern` function, which creates a new `pattern_t` on the heap from the provided string `line`. The resulting quilt pattern should be valid, meaning all strings in `patches` should have length `num_cols` and be null-terminated. The last line should be padded with white (`'W'`) as needed.

```
BWBBWG
BGWBWB
```

```
BWBBW
GBGWB
WBWWW
```

Figure 2: `line` string `"BWBBWGBGWBWB"`
Left: `make_pattern(line, 6)`
Right: `make_pattern(line, 5)`

Instructions: Complete the implementation of `make_pattern` below. You should use `init_pattern`, shown on the previous page, and the C standard library functions on the reference card. If you don't use a line, leave it blank.

```
1  pattern_t *make_pattern(char *line, uint32_t num_cols) {
2      uint32_t num_patches = strlen(line);
3      if (!num_patches) { return NULL; }
4      pattern_t *pat = init_pattern(
5          (num_patches + num_cols - 1) / num_cols, num_cols);
                                  Q3.5
6      if (!pat) { return NULL; }
7      char *cur = line;
8      for(int i = 0; i < pat->num_rows - 1; i++) {
9          strncpy(pat->patches[i], cur , num_cols);
                        Q3.6          Q3.7     Q3.8
10         *(pat->patches[i] + num_cols) = '\0';
                             Q3.9
11         cur += num_cols;
                    Q3.10
12     }
13     uint32_t last_row = pat->num_rows - 1;
14     size_t remainder = strlen(cur);
                               Q3.11
15     strncpy(pat->patches[last_row], cur , remainder);
                        Q3.12              Q3.13     Q3.14
16     if (remainder < num_cols) {
17         memset(pat->patches[last_row] + remainder,
                                    Q3.15
18                'W' ,
                   Q3.16
19                num_cols - remainder);
                          Q3.17
20     }
21     *(pat->patches[last_row] + num_cols) = '\0';;
                             Q3.18
22     return pat;
23 }
```

Q4 *61Captivating* ✨ **(16 points)**

Woo! You've become a TA for CS61C, and it's time for your first office hours. Your student, Alysa Liu, shows you their code for a RISC-V procedure, `mystery`, which they claim does not work properly.

`mystery`: May not follow proper calling convention.		
Arguments	`a0`	???
	`a1`	???
Return value	`a0`	???

The RISC-V procedure `malloc` takes in `a0` the number of bytes to allocate, and returns in `a0` a pointer to the allocated memory.

```
1  mystery:
2      mv s0 a0
3      mv s1 a1
4      li s2 0
5
6      slli a0 s1 2
7      jal malloc
8      mv s3 a0
9
10 loop:
11     blt s1 s2 done
12
13     slli t0 s2 2
14     add t1 s0 t0
15     add t2 s3 t0
16
17     lw a0 0(t1)
18     jal square
19     sw a0 0(t2)
20
21     addi s2 s2 1
22     j loop
23
24 done:
25     mv a0 s3
26     jr ra
27
28 square:
29     mul a0 a0 a0
30     jr ra
```

For Q4.1-Q4.4, match each s-type register to its corresponding value in `mystery`.

Q4.1 (1.5 points) `s0`

- ○ number of array elements
- ○ current array index
- ○ output array address
- ○ number of bytes allocated
- ● input array address
- ○ return address

Q4.2 (1.5 points) `s1`

- ● number of array elements
- ○ current array index
- ○ output array address
- ○ number of bytes allocated
- ○ input array address
- ○ return address

Q4.3 (1.5 points) `s2`

- ○ number of array elements
- ● current array index
- ○ output array address
- ○ number of bytes allocated
- ○ input array address
- ○ return address

Q4.4 (1.5 points) `s3`

- ○ number of array elements
- ○ current array index
- ● output array address
- ○ number of bytes allocated
- ○ input array address
- ○ return address

Solution: The `mystery` function takes as input an array of values in `a0`, and the number of values in `a1`. The function loops to compute the square of each of these values while storing them in a new array returned in `a0`.

Q4.5 (2 points) When attempting to test the code with Venus, you notice the program enters an infinite loop. Which register most likely causes this error?

○ `s1` ○ `s2` ● `ra` ○ `t2`

Why? In the box below, give your answer in at most 10 words.

Calling convention code for ra is not present

Solution: The issue here primarily arises from calling convention not being properly followed for the `ra` register. It is modified when calling `jal malloc`, and later `jal square` but is not saved/restored from the stack with proper calling convention. As a result, when done calls `jr ra`, the code jumps back to `loop` rather than the code that initially called `mystery`.

Many students indicated that the issue may be related to `s1` and `s2` because the condition is incorrect or `s2` should be decremented. While the condition is in fact an issue as we see in Q4.9, it is not the cause of an infinite loop. By incrementing `s2` (our counter), it will eventually be larger than `s1` (the number of elements in the array) and cause the branch to occur.

To receive credit for the 'Why?' box, answers must have explicitly mentioned that `ra` is not saved or calling convention is not followed. Answers mentioning `ra` being changed inside of `square` are also marked incorrect, as the change happens upon execution of the `jal` instruction.

Q4.6 (2 points) Alysa's project partner rewrites `square`, and now calls to `mystery` error on line 19. Which register causes this error?

○ `s1` ○ `s2` ○ `ra` ● `t2`

Why? In the box below, give your answer in at most 10 words.

The new square implementation changes the value of t2

Solution: Since we have no knowledge of what the new implementation of `square` looks like, we must assume that it can change any of our temporary registers. In this case, we use `t2` immediately following a call to `square` without having saved/restored it from the stack. As a result, we may be attempting to save to an inaccessible/garbage address.

After fixing this error, the code seems to work! Just in case, the student enables Memcheck. Upon execution with a test input, the following warning is printed.

(Question 4 continued...)

```
[memcheck] Invalid memory access of size 4. Address 0x10000A5C is 0 bytes
after a block of size 28 in static.
   Program Counter: 0x00000060
   File: mystery.S:17
   Instruction: lw a0 0(t1)
```

Q4.7 (1.5 points) How many elements are present in the test array?

7

Q4.8 (1.5 points) Which address is this test array (i.e. the 0th element) located at?

0x10000A40

Q4.9 (3 points) Which **one** line would you replace to fix the Memcheck warning? Provide the line number and full instruction to replace the original. *Hint: When the warning occurs, registers s1 and s2 contain the same value.*

Line: 11

Instruction: beq s1 s2 done

Q5 *61Counting* 🔢 **(20 points)**

The **binary population count ("popcount")** is the number of 1 bits in the binary representation of a number. A number has an odd popcount if the number of 1 bits is odd.

Examples:

- `1 → 0b1 → one 1 → odd`
- `2 → 0b10 → one 1 → odd`
- `3 → 0b11 → two 1s → even`
- `7 → 0b111 → three 1s → odd`

(8 points) Implement the `odd_popcount` function.

`odd_popcount`: Determines whether the number in `a0` has an odd number of 1 bits.		
Arguments	`a0`	A non-negative integer
Return value	`a0`	1 if the popcount is odd, otherwise 0

```
1  odd_popcount:
2      li t0 0
3  loop:
4      andi t1  a0   1
                Q5.1 Q5.2
5      xor  t0 t0 t1
        Q5.3
6      srli a0 a0 1
            Q5.4
7      bne x0  a0  loop
               Q5.5 Q5.6
8      mv  a0   t0
          Q5.7 Q5.8
9      jr ra
```

The rest of this page is left intentionally blank.

The **parity depth** of a non-negative integer `n` is the total number of integers from `0` to `n` (inclusive) that have an odd popcount. For example, the parity depth of 3 is two because the popcounts of `0`, `1`, `2`, and `3` are even, odd, odd, and even, respectively.

(10 points) Implement the `parity_depth` function. For this question, you may assume that `odd_popcount` has been correctly implemented.

`parity_depth`: Computes the parity depth of a number.		
Arguments	`a0`	A non-negative integer
Return value	`a0`	Parity depth of the input value

```
1  parity_depth:
2      # prologue omitted
3      beq  a0 x0 base_case
                Q5.9   Q5.10
4      mv  s0  a0
5      jal odd_popcount
                Q5.11
6      mv s1 a0
7      addi a0 s0 -1
            Q5.12
8      jal parity_depth
                Q5.13
9      add  a0  a0  s1
10 done:
11     # epilogue omitted
12     jr  ra
13 base_case:
14     li a0 0
         Q5.14
15     j done
          Q5.15
16
```

Q5.16 (2 points) Which registers need to be saved in the prologue and restored in the epilogue for `parity_depth` to satisfy RISC-V register calling conventions? Select all that apply.

■ `s0` ■ `s1` ☐ `a0` ☐ `t0` ☐ `t1` ■ `ra` ○ None of the above

Grading: Correct registers: `ra`, `s0`, `s1`

`ra` must be saved because the function makes function calls using `jal`.

`s0` and `s1` are saved registers that are modified in the function and must therefore be preserved across calls.

Temporary registers `t0` and `t1` do not need to be saved.

Q6 *61Circuit* (10 points)

Q6.1 (2 points) Fill in the blanks for the Boolean expression for the below table in terms of `A`, `B`, and `C`. Your answer may consist only of the following operators and symbols:

Operators			Symbols		
NOT	AND	OR	Inputs	Constants	Parentheses
`!A`	`A * B`	`A + B`	`A, B, C`	`0, 1`	`()`

A	B	C	Output
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

The optimal number of operators is 3. Partial credit will be granted for fully equivalent solutions using at most 4 operators.

Output = `(A * !B) + C`

Solution: Observe that the first, third, and seventh rows are the only rows with an output of 0. We can express the output by taking the complement of the sum of these specific cases:

$$\overline{\overline{A}\overline{B}\overline{C} + \overline{A}B\overline{C} + AB\overline{C}}$$

We can simplify this expression step-by-step:

$$\begin{aligned}
\overline{\overline{A}\overline{B}\overline{C} + \left(\overline{A} + A\right)B\overline{C} + B\overline{C}} &= \overline{\overline{A}\overline{B}\overline{C} + B\overline{C}} \\
&= \overline{\left(\overline{A}\overline{B} + B\right)\overline{C}} \\
&= \overline{\overline{A} + B} + C \\
&= \left(A * \overline{B}\right) + C
\end{aligned}$$

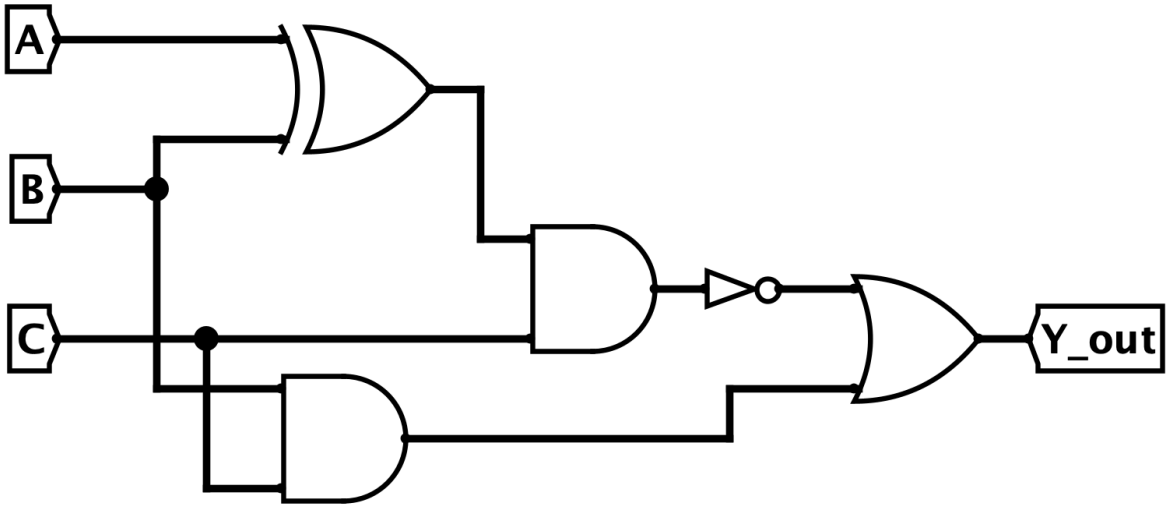
Q6.2 (2 points) Simplify the following expression to use at most 1 operator. Partial credit will be granted for fully equivalent solutions using at most 2 operators.

```
!(A * !(B + !A)) * (A + B) + !(!A + B) * !(A * B)
```

Output = `A + B`

The rest of this page left intentionally (mostly) blank.

For Q6.3 — Q6.5, consider the following circuit and timings, given that `A`, `B`, `C`, and `Y_out` are all directly connected to registers.



$t_{\texttt{setup}} = 6$ ns $\quad t_{\texttt{clk-to-q}} = 8$ ns $\quad t_{\texttt{NOT}} = 10$ ns $\quad t_{\texttt{AND}} = 35$ ns $\quad t_{\texttt{OR}} = 20$ ns $\quad t_{\texttt{XOR}} = 25$ ns

Q6.3 (2 points) What is the longest combinational logic delay?

90ns

Solution: $R_{\texttt{A}} + R_{\texttt{B}} \longrightarrow$ XOR $\longrightarrow$ AND $\longrightarrow$ NOT $\longrightarrow$ OR = 25 + 35 + 10 + 20 = 90ns

Q6.4 (2 points) What is the minimum allowable clock period?

104ns

Solution: $T_{\texttt{min}} = t_{\texttt{clk-to-q}} + t_{\texttt{CL,max}} + t_{\texttt{setup}} = 8 + 90 + 6 = 104$ ns

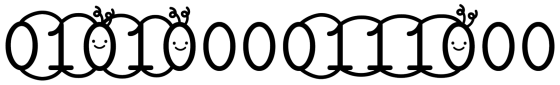
Q6.5 (2 points) What is the maximum allowable hold time?

63ns

Solution: $T_{\texttt{hold}} = t_{\texttt{clk-to-q}} + t_{\texttt{CL,min}} = 8 + 55 = 63$ ns

Q7 *61Caterpillars* **(10 points)**

A 61Caterpillar is a bitstring, beginning with a `0`, followed by one or more `1`s, and ending with a `0`. Zeroes can be shared by two 61Caterpillars. For example, `01010000111000` is three caterpillars:

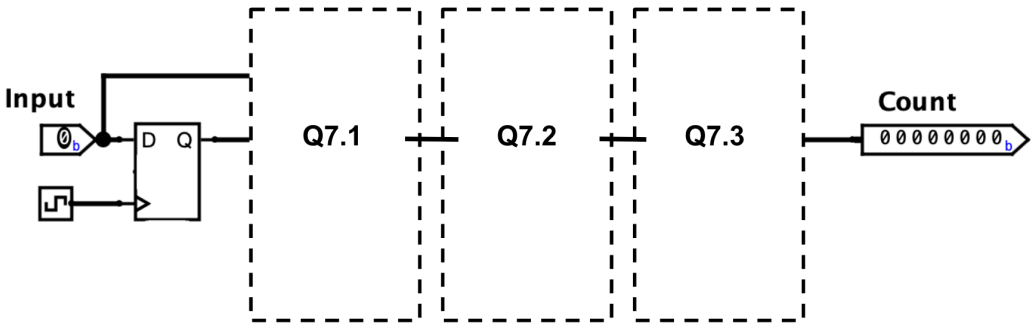


The `count_caterpillars` circuit takes in a zero-prefixed bitstring and maintains a `Count` of the number of 61Caterpillars in the bitstring. Increment `Count` the cycle immediately after a 61Caterpillar is seen.

Example `Input`	`01010000111000`
Example `Count`	`00011222222233`

Design `count_caterpillars` with the following specification:

- Your circuits may consist of some or all of the following: wire, constant, logic gate (`NOT`, `AND`, `OR`, `XOR`), mux, adder, clock, register.
- The input string is guaranteed to begin with 0
- All wires, registers, and/or flip-flops used will hold 0 on initialization.



Please read through all parts before drawing your circuit on the next page.

Q7.1 (3 points) Implement subcircuit Q7.1 that outputs a one bit signal holding `1` if we see the head of caterpillar, and `0` otherwise. In the circuit, `Input` is the current bit, `Q` is the previous bit, and `sig` is the head-detection signal.

Q7.2 (3 points) We want to add 1 to **Count** if a caterpillar is found, and add 0 to **Count** otherwise. What component allows us to choose between two inputs based on **sig**?

You must represent all constants in binary form. Implement subcircuit Q7.2 that outputs an 8-bit 1 or a 0 based on the signal given by Q7.1.

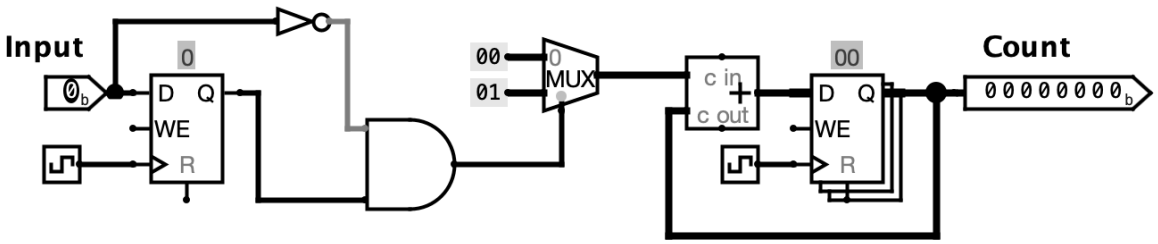
In the circuit, **sig** is the 1-bit control input, and **val** is the 8-bit output value to add to **Count**.

Q7.3 (4 points) Finally, in order to maintain a count of our caterpillars, we need an adder circuit. Draw an adder circuit that adds the value from Q7.2 to **Count** on the rising edge of the clock. You may assume that the value provided by Q7.2 matches the bit-width of **Count**.
In the circuit, **val** is the 8-bit value being added, and **Count** is the updated 8-bit count output.

90ns

Solution: $R_A + R_B \rightarrow$ XOR $\rightarrow$ AND $\rightarrow$ NOT $\rightarrow$ OR = 25 + 35 + 10 + 20 = 90ns

Solution:



The solution shows the entire circuit, broken up according to the question parts defined on the previous page. Note that while the question asks for boolean constants, Logisim (where the solution was drawn) defaults to hex, so the constants `0x00` and `0x01` should have been drawn as `0b00000000` and `0b00000001`.

For Q7.1: Logically equivalent solutions were given full credit

For Q7.2: Solutions correctly using splitters or bit extenders were given full credit.

Q8 *61Cool* 😎 **(0 points)**

These questions will not be assigned credit. Feel free to leave them blank.

Q8.1 What cool stuff are you doing over spring break?

Q8.2 If there's anything else you want us to know, or you feel like there was an ambiguity in the exam, please put it in the box below.

For ambiguities, you must qualify your answer and provide an answer for both interpretations. For example, "if the question is asking about A, then my answer is X, but if the question is asking about B, then my answer is Y". You will only receive credit if it is a genuine ambiguity and both of your answers are correct. We will only look at ambiguities if you request a regrade.