COSC-175: Systems-I
Fall 2025
Mid-term exam — Solutions

1. **Provide short answers the following questions:**

   (a) Question: What does it mean for NAND to be a *universal logic operator*?

   Answer: NAND's universality implies than any possible logic function can be expressed and implemented using only NAND recombined with itself.

   Observations: The most common (and minor) mistake was to note only that NAND can be used to implement AND/OR/NOT. While that is true, it is, of itself, insufficient; a complete answer must then observe that AND/OR/NOT are, as a trio, universal *because they can be used to express any logic function* (as established by the existence of *disjunctive normal form*.

   (b) Question: What is the *critical path* of a circuit?

   Answer: It is the longest path (not necessarily unique) from any original input to a combinational circuit to a final output, thus dictating how much time one must wait between presenting input values and then being able to trust the correctness of the output values.

   Observations: The one pitfall in many answers was vagueness. Stating that the critical path was (for example) the "biggest number of gates" is unclear for its lack of statement about the relationship between all those gates. (Also, grammar matters, and "biggest number" is not correct.)

   (c) Question: What is the difference between *combinational* and *sequential logic*?

   Answer: *Combinational logic* is only the composition of logic operations with one another; the input values alone determine the output. *Sequential logic* employs *memory elements*, with external inputs *and* the values in the memory elements determining the output together. That is, the output of a sequential circuit depends on its *state*, which is provided by those memory elements.

   Observations: Again, vagueness was the problem for some, especially in relation to *sequential logic*. However, any mention of *memory* or retained values from previous inputs was sufficient.

2. QUETION: **Subtract** 10 **from** 4 using 5-bit, *twos complement* binary numbers. Show your work, and show that your answer is correctly -6.

   ANSWER:

   - Subtraction is addition of a negated second value: $4 - 10 = 4 + -10$
   - Negate 10: $01010 \rightarrow 10101 + 1 = 10110$
   - Add $4 + -10$: $00100 + 10110 = 11010$
   - Negate 11010 (which is negative) to show that it is 6: $11010 \rightarrow 00101 + 1 = 00110$

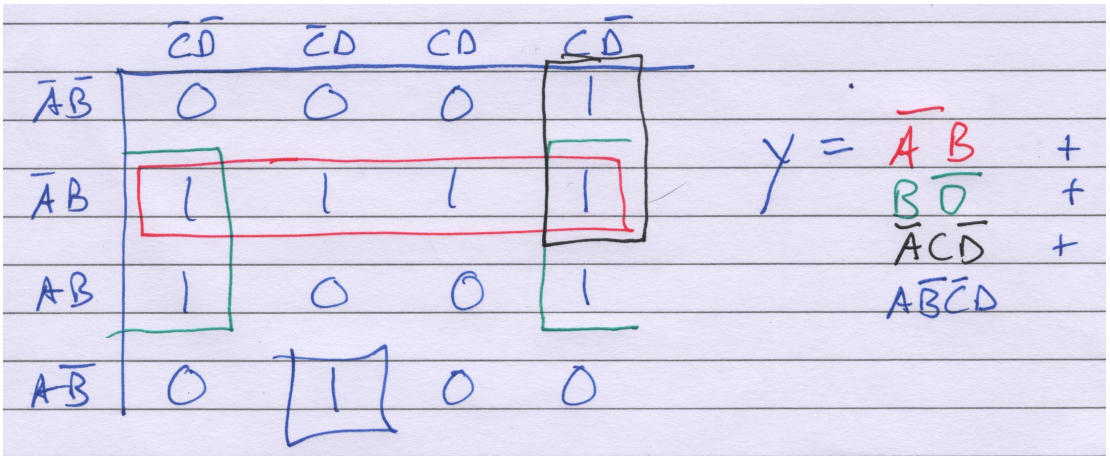
   OBSERVATIONS: This question went well overall, with few points lost. The printed question indicated the use of *4-bit* (instead of the proper *5-bit*) values, and while those who proceeded with 4-bit values should have noticed that -10 is not representable in a 4-bit twos-complement value, I didn't hold it against anyone (since the typo in the question was my fault!).

3. QUESTION: Consider the logic function described by the following truth table:

A	B	C	D	Y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	0
1	1	1	0	1
1	1	1	1	0

Produce a simplified version of this logic function by using a *Karnaugh map*.

ANSWER:



OBSERVATIONS: Far and away the most common problem here was the failure to see the 2x2 rectangle that provided the $B\bar{D}$ term. This is an understandable mistake because the adjacent 1x4 term for $\bar{A}B$ contained two of the four values to be captured. Nonetheless, the need for the largest valid rectangles led to the greatest simplification.

Some tried to work around the issue by taking what the map's rectangles provided them and then applying further algebraic transformations. It is important to note

that, even with such algebraic steps, if the result is expressed using only AND/OR/NOT (as it should be), then the number of logic operations (and thus gates) is **not smaller** for having performed these steps. A Karnaugh map, used properly, yields the greatest possible reduction.

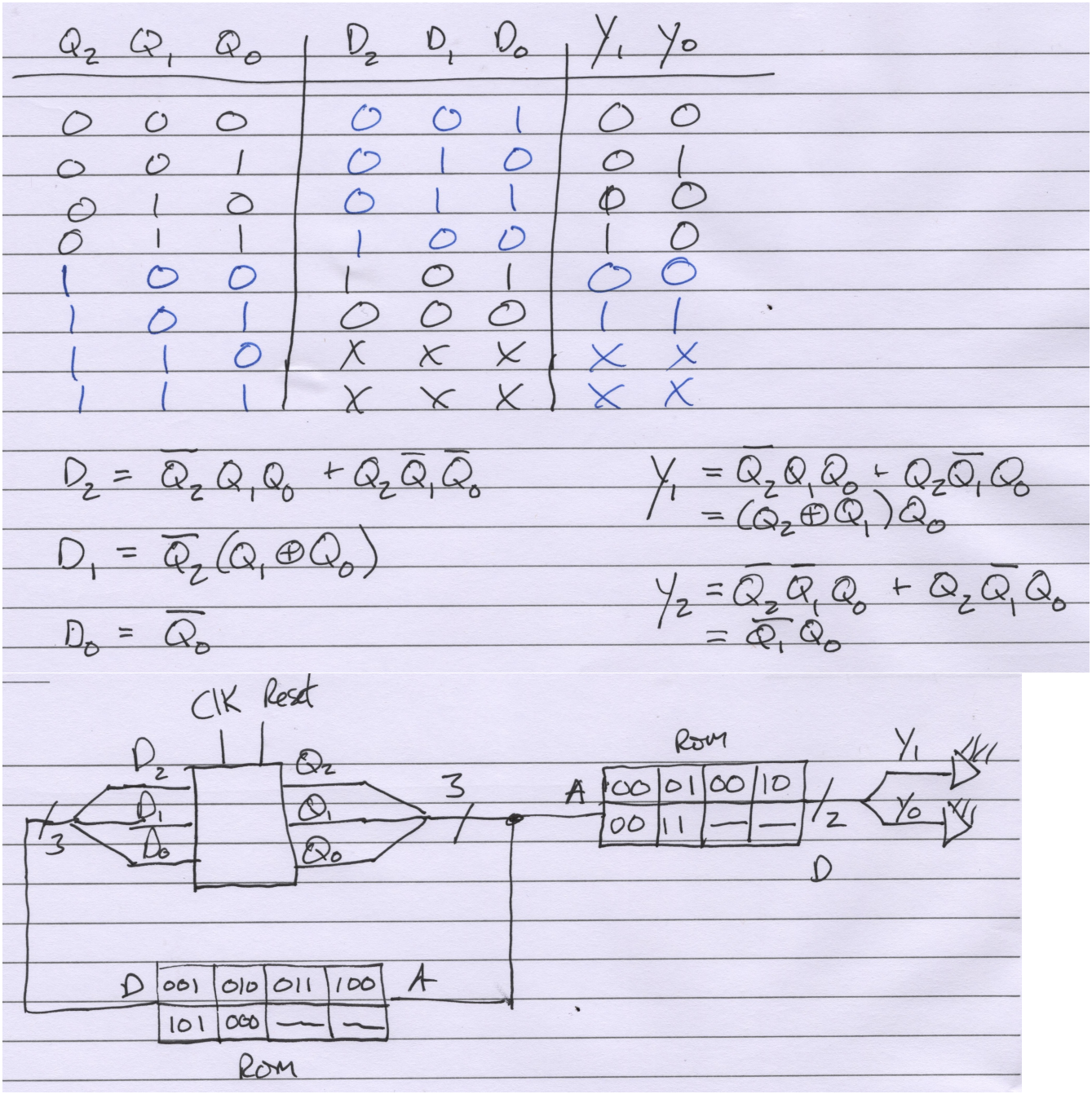
4. QUESTION: **Design and draw a circuit that emits the following repeating sequence**:

$$00, 01, 00, 10, 00, 11, \ldots$$

Your circuit should have 2 bits of output, as well as *clock* and *reset* inputs. Every 6 clock cycles should progress through the sequence; upon the next cycle, the circuit should "wrap around" and begin the sequence again. The *reset* should return the circuit to the beginning of the sequence. You may use basic memory devices such as *flip-flops* (which may be aggregated into *registers*).

ANSWER: Although my solution simply relies on ROM's loaded with the correct values, the outputs can also be expressed as the logic functions shown below.

[Shown on next page…]



OBSERVATIONS: This was a difficult question because, done properly, it required a conceptual separation of the *internal state number* and the *external output*. Internally, the circuit should use 3 bits to count through the numbered sequence of states (e.g., $0, 1, 2, 3, 4, 5, 0, 1, 2, 3, 4, 5, \ldots$). Externally, the state number should map onto the a 2-bit output pattern. The even-numbered states always output 00, **but they are still distinct states**, where each holds a specific position in the sequence. Those who tried to draw a *finite state machine* in which the states were the output values found multiple states into and out of the 00 state. With no additional input value to choose between the output transitions from this 00 state, there's no way to know (from being in that 00 state) which state to transition to next. That is, if your state information

is only 00, then you don't know whether the next state is 01, 10, or 11, because each state does not "remember" which state preceeded it.

Some did solve it in a more ad-hoc was by using a larger number of flip-flops to store the entire sequence in pairs of bits, and then cycle them through the memory elements. This approach can work, but it requires a separate pre-loading cycle.

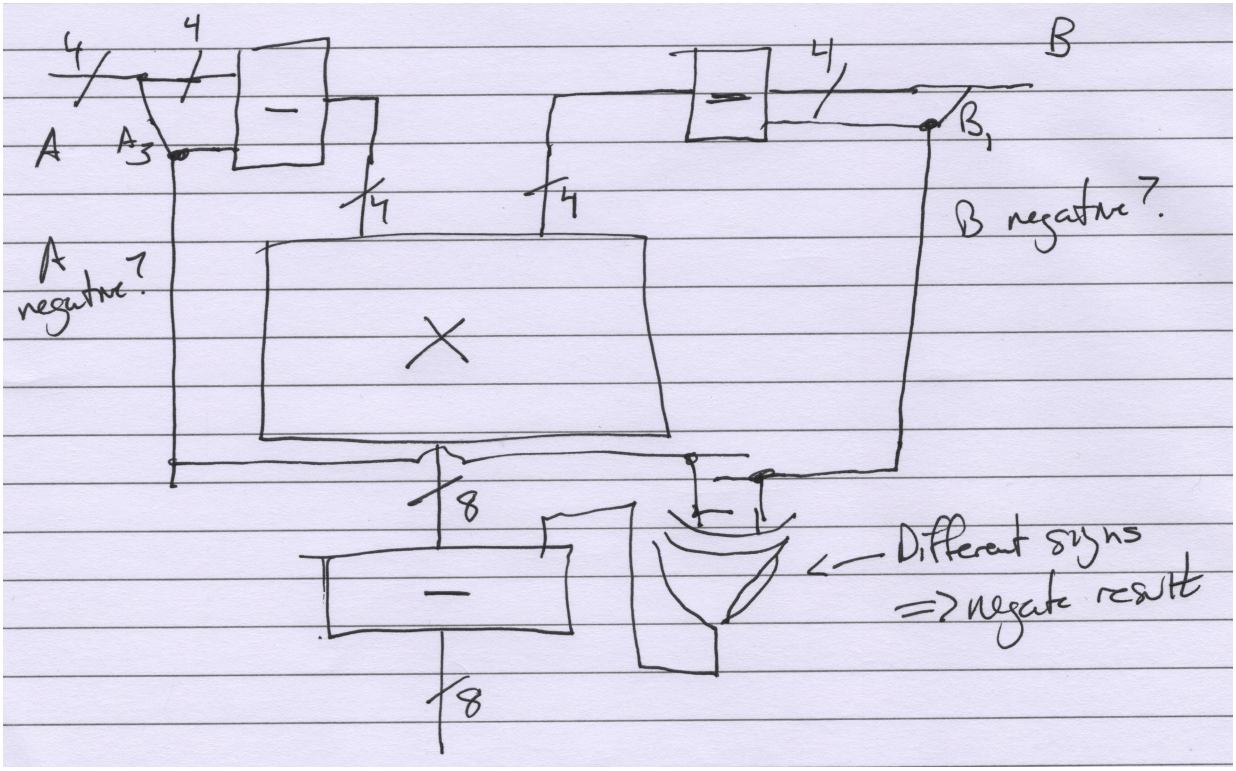
Others tried to ping-pong between the 00 of the even-numbered states and the non-00 of the odd-numbered ones, somehow trying to get the output to emit 00 on one step, and then the stored non-00 value on the next. To advance the non-00 value, they employed adders/incrementers, but often then failed to reckon with the incrementation wrapping around to 00 again, introducing a 00 output in a place where it doesn't belong.

Finally, some tried to leverage the *clock* or *reset* values as being more than they should noramlly be. The *reset* should only jump back to the beginning of the sequence (state zero); the *clock* should only be used to trigger the flip-flops. Using these inputs as part of the logic itself to determine the next value isn't valid. The *reset* input is *asychronous*—it zeros the memory elements immediately, not waiting for a change in the *clock* signal—and thus can't participate in the sequential logic itself. The *clock* cannot be relied upon as a value determining the input because then the *clock* as a trigger and the data value that uses the *clock* as an input are racing to see which completes its part of the circuit first; that's unstable at best, and consistently wrong at worst.

5. QUESTION:Assume that you have developed a *combinational multiplier*, which multiplies two 4-bit inputs to produce an 8-bit output. Assume that, like our sequential multiplier from lab-5, this multiplier only produces correct results for *unsigned binary integers*; if we interpret the inputs as *signed, twos complement integers*, then the results of the multiplication may be incorrect.

   Augment this multiplier to handle *signed, twos complement integers*. You may consider the original multiplier as a "black box" that accepts two unsigned inputs and produces an unsigned output, and then add logic/circuitry that manipulates those inputs and outputs. You may use logic gates as well as high-level components that we have already constructed (e.g., multiplexers, adders/subtractors, etc.).

   ANSWER:



   OBSERVATIONS: It is interesting that so many provided perfectly correct answers on Question 2 then seemed to forget entirely how twos complement worked in this question. The most common mistake was simply to assume that manipulating the most significant bit would be sufficient; of course, twos complement requires proper negation of a value, and that process will alter the sign bit as well as the rest of the bits.

   A number of answers negated the inputs and the output, but would do so irrespective of the signs of the input values. Similarly, some answers negated only the output (the product), but failed to selectively negate the inputs (to obtain the absolute value of each) before passing those inputs to the multiplier itself.

There were many novel structures for the selective negation. For example, rather than invert the bits selectively by using XOR gates (where the input was the inversion selection as one input and the original bit value as the other), many split off a copy of each bit, inverted it with a NOT gate, and then used a *multiplexer* to selectively choose between the two. This approach is fully correct and got all the credit, but is bulkier than necessary.