Problem Set 8
Computer Science 240
Fall 2014
Due: Friday, November 7

Relevant Reading. Patterson & Hennessy §C.7 -- §C.10

Problem 1. Build a 64K×8 SRAM by adding minimal additional logic to an underlying 16K×32 SRAM. Note: these SRAMs have the same capacity, but different dimensions.

- Treat the existing 16K×32 SRAM as an indivisible element with 32 data-in lines, 32 data-out lines, a set of address lines, and a write-enable line.

- Your resulting 64K×8 SRAM should have 8 data-in lines, 8 data-out lines, a set of address lines, and a write-enable line.

- Assume that all registers inside the memory are edge-triggered (just like the registers used in lecture).

- Assume that, for both the existing SRAM and the one you build, memory is always reading unless the write-enable line is asserted (1). (In other words, do not implement a chip-select line.)

- Be careful with writes: when writing an 8-bit value into your memory, you must be sure not to destroy the other 24 bits that share the same underlying 32-bit cell.

- Do not draw each individual line in a bus (a group of lines treated the same way, e.g., 32 data-out lines) unless necessary to distinguish their endpoints.

Problem 2. Construct a 256×8 SRAM using identical 256×4 SRAM components. Carry over the same types of inputs, outputs, and assumptions as in Problem 1, adjusted for these new dimensions. Unlike in Problem 1, the underlying memory component in this problem has a smaller capacity than the final product.

Problem 3. This problem considers issues as related to mechanical input switches and the signals they produce. We will solve a problem with these switches by implementing finite-state machines using combinational and sequential logic. Read P&H section C.10 before trying this exercise.
Context:
The signal derived from a typical switch (e.g., a keyboard key) is not “clean.” Each time the metallic contacts inside the switch make or break connection, the signal changes. Since the contacts are not perfect, they normally “bounce” on each single actuation (e.g., key press). Waveform $x$ shows the result below (ignore (C)LOCK and $z$ for now): Several bounces on the switch’s depression and a few more on its subsequent release.

![Waveform](image)

The time taken before bounces die out varies with the quality of the switch and may be several milliseconds. Since a computer can respond to sub-microsecond signals, the single key activation would be treated as a sequence of many. The purpose of this problem is to design a switch “debouncer,” a sequential circuit that filters out these intermittent “bouncy” signals.

![Finite State Machine](image)

The input $x$ is the switch signal. To read the input, our circuit will check $x$ on every CLOCK pulse. These pulses (the waveform marked LOCK in the figure above) are spaced $t$ seconds apart, where $3t$ is greater than the maximum time it takes for bounces to die out. While $x$ is stable (i.e., not bouncing) at either level (0 or 1), $z = x$. On any switch actuation in either direction, $z$ does not change until the second successive inverted $x$ sample is detected. This is illustrated in the figure above. The (C)LOCK pulses shown are much wider than they have to be. The time scale has been modified so that you may distinguish both edges of each pulse. The $z$ waveform is drawn by assuming that the internal devices are positively edge-triggered.

The finite state machine shown below describes the behavior of the debouncer circuit. Circles represent states. Arrows between states represent transitions: how the state will change when we observe an $x$ value on a clock pulse. OFF and ON are the stable rest states. PRE-OFF and PRE-ON are temporary states used to represent the fact that $x$ has changed (i.e., $x \neq z$), but we have only observed this change on one clock pulse so far. If the new $x$ value is observed again at the next clock pulse, the switch will change to a new stable state (PRE-ON becomes ON, PRE-OFF becomes OFF), otherwise it will revert to its prior stable state (PRE-ON reverts to OFF, PRE-OFF reverts to ON). Each state is
also labeled with the corresponding \( z \) output value that should appear as the output of the debouncer whenever that state is in effect.

For example, suppose that the system is initially in the OFF state. If \( x \) is observed to be 0 on the next clock pulse, then the system will stay in the OFF state, as indicated by the self-transition in the state machine. If \( x \) is observed to be 1 at the next clock pulse, the system will then move to the PRE-ON state. With the system in the PRE-ON state, if \( x \) is 1 again on the next clock pulse, then the system will move to the ON state. Otherwise, if \( x \) is 0 on this clock pulse, then the system will return to the OFF state. Thus, two consecutive observations of \( x = 1 \), will take the system from OFF to ON. The ON-to-OFF transitions work analogously.

**Your task:**
Design a debouncer circuit by following the steps below. You will use a simple register built from flip-flops to store the current state of the debouncer. The output of your debouncer will be derived from this current state. Finally, you will implement logic to derive the next state based on the current state and the current \( x \) value, and store this next state back into the register.

The following steps will guide your implementation. Submit your answers for each.

a. How many bits are required to represent four states?

b. Assign each state an \( n \)-bit encoding (for the \( n \) you determined).

c. Each bit in your state encoding will be stored by one D flip-flop. These \( n \) flip-flops together form an \( n \)-bit register that represents the current state, \( Q \). The individual bits of the current state are the values stored (and output) by each of the individual flip-flops: \( Q_0 \) through \( Q_{n-1} \).

d. The debouncer output, \( z \), is derived strictly from the current state. Produce a truth table for \( z \), given the current state (available as \( Q_0 \) through \( Q_{n-1} \)) as input.

e. Produce a truth table for the next state of the system, \( Q' \) (expressed as bits \( Q'_0 \) through \( Q'_{n-1} \)), given the current state, \( Q \) (expressed as bits \( Q_0 \) through \( Q_{n-1} \)), and
the current value of \( x \) (the current debouncer input). Each row in the table describes a single transition in the finite state machine diagram.

f. Using the truth tables you produced above, write and simplify Boolean expressions for \( z \) (the debouncer output) and the next state encoding. Each bit \( Q'_i \) of the next state’s encoding will be a separate function over the inputs of the truth table.

g. Implement these Boolean expressions for \( z \) and \( Q' \) using combinational logic gates wired to the inputs and outputs of the flip-flops. To finish the state transition, connect the computed value for each \( Q'_i \) to the input \( D_i \) of the corresponding flip-flop \( i \). This will ensure that the next state is stored into the register and the state transition is complete.

-- end of required problems --

**Challenge Problem.** In lecture, we mentioned that some memories use a single bus (a single set of data lines) for both data in and data out. Choose any of the RAMs above or in the lecture slides and add minimal logic to implement shared data in/out lines without interference. Based on the RAM you choose, consider the existing behavior of its data-in and data-out buses carefully to determine what changes will be necessary. Assume that the user of the memory handles the other end of the data bus carefully as well.