Sequential Logic Circuits, Clocks

CS 350: Computer Organization & Assembler Language Programming

Sequential Logic Circuits

- A Sequential Logic Circuit combines combinatorial circuits and storage elements. The combinatorial circuit calculations take input values and memory values and produce output values and updated memory values.

- It turns out that we also need a way to control when the calculations occur relative to when memory gets updated. Consider the following block diagram for a sequential logic circuit; ignore the clock for now. There are two storage elements, with inputs $D[1:0]$ and outputs $S[1:0]$.

- If we think of calculating $D$ and setting $S$ as four operations (calculate $D_1$ and $D_0$ and set $S_1$ and $S_0$), then different orders of operations might produce different results. As a concrete example, say we have input $X$ and we calculate $D_1 = \overline{S}_1 * S_0 = 1 * 0 = 0$ and $D_0 = S_1 + X * S_0 = 1 + 0 * 0 = 1$.
  - If we calculate $D_1$ and $D_0$ before setting $S_1$ and $S_0$, then $S$ becomes 01.
    
    $D_1 = \overline{S}_1 * S_0 = 1 * 0 = 0$
    $D_0 = S_1 + X * S_0 = 1 + 0 * 0 = 1$
    $S_1 \leftarrow D_1 = 0$
    $S_0 \leftarrow D_0 = 1$
  - But if we calculate $D_1$ and set $S_1$ before calculating $D_0$ and setting $S_0$, then $S$ becomes 00.
    
    $D_1 = \overline{S}_1 * S_0 = 1 * 0 = 0$
    $S_1 \leftarrow D_1 = 0$
    $D_0 = S_1 + X * S_0 = 0 + 0 * 0 = 0$
    $S_0 \leftarrow D_0 = 0$

- For this example, the times taken to calculate $D_1$ and $D_0$ don’t depend on the values of $S_1$ and $S_0$, so we’d expect to consistently see the same sequence of events. (E.g., calculate $D_1$ and $D_0$ before setting $S_1$ and $S_0$.) So at least, the behavior of the overall circuit would be consistent.

- In a more general circuit, the times taken to calculate $D_1$ and $D_0$ might depend on the values of $S_1$ and $S_0$, so we might see different sequences of events, which means that the circuit behavior could be harder to predict.
Clocks and Flip-Flops

- To solve the problem caused by allowing calculations and memory updates to intermingle, we will alternate them: Do all the calculations, then do all the memory updates, and repeat.

- To signal which alternation we’re in, we’ll use a Clock, a device that cycles its output (the Clock Signal) from 0 to 1 and back, at a regular speed.

- During one half of the clock cycle, we’ll use the memory state to do our calculations, but we won’t allow the results of those calculations to change the memory state. During the other half cycle, we’ll update the memory state while ignoring any changes to the calculation results caused by the update.

- We’ll use a Master-Slave Flip-Flop to store a memory bit using a clock to do this alternation.
  - Internally, the flip-flop holds two bits, one to hold the current external state of the flip-flop, and one to hold the next external state of the flip-flop.
  - The clock is used to alternate between two modes.
  - In the first mode, it reads in the value of the next external state of the flip-flop but maintains the current external state unchanged.
  - In the second mode, it updates the external state by copying the next external state value into it but ignores requests to change that next external state value.

- In terms of circuitry, a master-slave flip-flop has two D-latches and a clock connection.
  - The output of the left-hand D-latch (labeled B in the figure below) is the output for the flip-flop, so B holds the current external state of the flip-flop.
  - The right-hand D-latch (labeled A below) holds the next external state. Its input is the input to the flip-flop, and its output is the input to B.
  - The flip-flop output is input to the combinatorial circuit; the combinatorial circuit output is the input to the flip-flop.
  - The write enable line of B is connected to the clock; the write enable line of A is the negation of the clock. So, when the clock is 0, we update A but not B; when the clock is 1, we update B but not A.
- In detail, when the clock signal is 0
  - The output of $B$ doesn’t change, so the combinatorial circuit continues to use that value as it does its calculation. The output of the calculation is copied into $A$.
  - The calculation needs to finish before $A$ stops writing into its state.
- When the clock signal is 1
  - The state of $A$ is copied into $B$ and becomes the new output of the flip-flop, which becomes the new input to the combinatorial circuit, so the combinatorial circuit may start recalculating its output.
  - Since different parts of the combinatorial circuit may change at different times, the output of the combinatorial circuit as a whole may be incomplete and unstable, but $A$ ignores its input, so this doesn’t matter.
- For fullest safety, the calculation should complete while the clock signal is 1 so that when the clock changes to 0, latch $A$ will see the same input throughout that half cycle. This requires the clock cycle to be at least twice the time taken by the slowest part of the combinatorial circuit.
- For an $m$-bit register, we have $m$ flip-flops all connected to the same clock. Either $m$ bits of data are being read into the register (without changing the register’s $m$-bit output), or $m$ bits of data are being output from the register (while ignoring the $m$ bits of data entering the register).

### Finite-State Machines
- In a finite state machine, we have a sequential logic circuit with an input, traditionally viewed as a character like ‘0’ or ‘a’. The input can change with each clock cycle, so it’s actually a sequence, like “a30Qz” etc.
- The storage element is traditionally viewed as containing an unsigned integer state.
  - Since the storage element is of fixed length, there are only so many possible states the machine can be in, hence the name, finite-state machine.
- The combinatorial circuit may or may not have an output external to the machine, but it does use the input and current state to calculate a new state.
- Some finite-state machines have an external output; in some machines the output depends only purely on the state number, and in others the output depends on the input and state together. The output can vary with the
clock, so it’s actually a sequence of outputs, and the machine can be viewed as an input-to-output string transformer.

- A machine that flashes one or more lights under a set pattern uses the state (and possibly an input) to decide which lights to turn on and off with each clock cycle. An example is a road safety sign.

- Some finite-state machines don’t have an output sequence but simply give an output at the end of the input sequence. The output is a single bit whose value depends on the state; this bit says whether or not the input sequence was “accepted” or “rejected”. Such machines are called “recognizers”.

- A text search based on a regular expression transforms the expression into a finite-state machine that reads a sequence of characters and accepts it iff the character string matches the pattern given by the expression.

- To specify a recognizer FSM you specify
  - The state set (the set of all possible states).
  - The start state (the one the FSM begins execution in).
  - The set of accepting states (when the FSM ends computation, it accepts the input iff it ends in an accepting state).
  - A transition function that describes how the FSM executes. It takes an input character and the current state and produces the next state for the machine (which may or may not be the same as the current state).

- FSM Execution:
  
  Initialize state ← start state;

  while there exists more input {
    read character of input
    set state to new state
    i.e., state ← transition_function(input_char, state)
  }

  if state ∈ Accepting States then
    output “Accept”
  else
    output “Reject”

- FSM Implementation using a Sequential Logic Circuit
  - The FSM state is represented as an unsigned bitstring and is stored in the sequential logic circuit’s memory. The FSM input is modeled as a sequence of characters over time, with a special “end of input” symbol attached to the end.

  - With each clock cycle, the FSM receives one symbol’s worth of input and sends it along with the current FSM state to the combinatorial logic circuit. The circuit calculates the new FSM state and sends it to memory; it also calculates the one bit of external output (saying whether or not we’d accept if the input ended here).
- Note the FSM is always in a state (which is why we needed a start state), and there are a fixed number of possible states.

**FSM Example:**
- Machine $M_1$ reads strings of a's and b's and accepts a string iff it contains exactly two occurrences of b. We have four states $S_0$, $S_1$, $S_2$, and $S_3$.
- For the transitions between states, here's state transition diagram. The transitions are indicated by arrows labeled with the character that causes the transition. The headless arrow indicates the start state. Double-circled states (there's just one, $S_2$) are accepting states.
- It's easy to follow the diagram and convince yourself that starting at state $S_0$, you can get to $S_2$ and accept a string exactly when it has two b's, with any number of a's interspersed before/behind/between them.

![](image)

- Note if you have a string with three or more b's, you end up in state $S_3$ with no way to get to another state (and certainly no way to get to an accepting state). The arrow labeled a, b means that in $S_3$, seeing an a or a b takes you back to $S_3$.
- One can also specify the transitions using a transition table:

<table>
<thead>
<tr>
<th>State</th>
<th>Input</th>
<th>New State</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_0$</td>
<td>a</td>
<td>$S_0$</td>
</tr>
<tr>
<td>$S_0$</td>
<td>b</td>
<td>$S_1$</td>
</tr>
<tr>
<td>$S_1$</td>
<td>a</td>
<td>$S_1$</td>
</tr>
<tr>
<td>$S_1$</td>
<td>b</td>
<td>$S_2$</td>
</tr>
<tr>
<td>$S_2$</td>
<td>a</td>
<td>$S_2$</td>
</tr>
<tr>
<td>$S_2$</td>
<td>b</td>
<td>$S_3$</td>
</tr>
<tr>
<td>$S_3$</td>
<td>a</td>
<td>$S_3$</td>
</tr>
<tr>
<td>$S_3$</td>
<td>b</td>
<td>$S_3$</td>
</tr>
</tbody>
</table>

*Initial state: $S_0$; Accepting state: $S_2$*

- If you're familiar with regular expressions, for any basic regular expression (one built with symbols, concatenation, alternation, and Kleene star), it's possible to design a recognizer FSM that accepts exactly the strings generated by the regular expression.
- Machine $M_1$ recognizes strings that meet the pattern a* b a* b a*.
- In fact, the basic algorithm for using a regular expression for pattern matching is to convert the expression to an equivalent FSM and then execute the FSM on any given input string.
- Less obvious is that every FSM has an equivalent regular expression. This makes FSMs and regular expressions equivalent as computation tools.
Activity Questions
1. How are combinatorial and sequential logic circuits different? Which uses state information?
2. How and why do sequential logic circuits alternate calculations and memory updates?
3. How many $D$-latches go into a master-slave flip-flop? What do the latch states mean? How is the clock connected? How does the flip-flop work, as the clock ticks? How does a sequential logic circuit use the clock and flip-flop? How fast can the clock be relative to the calculation part of a sequential logic circuit?
4. What is the connection between finite state machines and regular expressions?

Activity Solution
1. A combinatorial logic circuit has no loops; the boolean value of an output can be described using a boolean expression of its input values. A sequential logic circuit combines a combinatorial logic circuit and memory into a loop (in addition to having inputs and outputs). A sequential logic circuit has a state that it can update; a combinatorial circuit simply performs operations.
2. We separate the calculation of the new state from the update of the state because if the two are intermingled, some of the calculations may be incorrect. (The calculations that use already-updated parts of the state might produce values that differ from the values that would have been produced if the state hadn’t been partially updated.)
3. Two $D$-latches, one holding the current external state of the flip-flop and one holding the next external state. For the rest, see the description in the section on clocks and flip-flops.
4. For each regular expression, we can build a recognizer FSM that accepts the strings generated by the pattern. In the other direction, every FSM has a regular expression that generates all the strings accepted by the FSM. I.e., FSMs and regular expressions are equivalent in expressive power.