Lecture 16 - State Machines

State Machines

- We talked about sequential logic circuits before. These turn out to be very powerful mechanisms for performing computation.
- We can generalize them a bit to the notion of a finite state machine (FSM).
- FSMs are abstract machines that have a finite state space. They transition between these states and output values based on inputs to the machine.
- Simplest finite state machine is called a recognizer. It only produces one bit of output after all the input is seen. It accepts the input if it outputs 1 ("yes") and rejects the input if it outputs 0 ("no").
- For example, we can build a recognizer FSM that accepts all binary strings with at least two 1s in them.

Specifying State Machines

- You need:
  - A finite set of states \( S \)
  - An input alphabet \( \Sigma \) (non-empty, finite set of symbols, e.g. \{0,1\})
  - A start state \( S_0 \in S \)
  - A transition function \( \delta : S \times \Sigma \to S \). This function takes an input symbol and the current state and outputs the new state.
  - A set of accepting states \( A \subseteq S \)

A pseudocode implementation:

```plaintext
Init_state <- start state;
while there is input left {
    input <- read a symbol;
    state <- \delta(state, input);
}
if state \in A then
    output "Accept" (1)
else
    output "Reject" (0)
fi
```

Implementing in a sequential circuit:

- The FSM's current state is represented as a bitstring (a number) and is stored in the sequential circuit’s memory.
- The input is just a stream of symbols. The last one represents "end of input." We can
make that symbol whatever we like, as long as it’s not in \( \Sigma \)
- During each clock cycle, the FSM reads one symbol, and sends that plus the current state to the combinational circuit, which calculates the new state and overwrites the memory representing the current state (remember how we did this with flip flops?)
- It also calculates one bit of output at each state (whether or not we’d accept at this point)
- FSM must always be in a certain state which is well defined. There must be a finite number of states (hence the name)

**FSM Example:**

- M is our machine. \( \Sigma = \{a, b\} \). M outputs 1 when it has seen exactly 2 occurrences of \( b \) in the input.
- We have four states in \( S = \{s0, s1, s2, s3\} \).
- For \( k = \{0, 1, 2\} \) we will be in \( sk \) iff we’ve seen exactly \( k \) \( b \)'s
- We’ll go to state \( s3 \) if we’ve seen three or more \( b \)'s
- State \( s0 \) is the start state, and \( s2 \) is our accepting state
- Transitions: for \( k=\{0,1,2\} \) if we’re in \( sk \) and we see a \( b \), we go to \( s(k+1) \)
- If we’re in \( s3 \) and we see a \( b \), we stay in \( s3 \)
- for all states, if we see a, we stay where we are
- What happens with \( abaabaa \)?
- What happens if we change our accepting state to \( s3 \)?

**The transition function**

- Can be represented as a transition table or transition matrix with 3 columns and \( |S| \times |\Sigma| \) rows (if it is a total function). First column is cur state, second is input, third is next state

<table>
<thead>
<tr>
<th>State</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>a</td>
</tr>
<tr>
<td>S0</td>
<td>b</td>
</tr>
<tr>
<td>S1</td>
<td>a</td>
</tr>
<tr>
<td>S1</td>
<td>b</td>
</tr>
<tr>
<td>S2</td>
<td>a</td>
</tr>
<tr>
<td>S2</td>
<td>b</td>
</tr>
<tr>
<td>S3</td>
<td>a</td>
</tr>
</tbody>
</table>
- We can also use a **state transition diagram**, which is a directed graph. Circular nodes represent states, directed edges (arrows) representing transitions. We label the arrows with input symbols.
- Accepting states are drawn with concentric circles

```
  a  
S0  b   S1  a  b  S2  a  b  S3
  a  
```

- **Another example:** what would a machine look like that only accepts strings that have nothing but 0s?

**Aside**

- What we’ve seen here is actually a model of computation, like TMs which we saw early on
- However, FSMs are more restricted that TMs (they aren’t as powerful computationally speaking)
- The “languages” that an FSM can recognize are much more restricted. In particular, an FSM can only recognize **regular languages** (those that don’t require memory when recognizing them)