Sequential Logic and State

Output depends on inputs \textit{and stored values}.

(vs. combinational logic: output depends only on inputs)

Elements to store values: latches, flip-flops, registers, memory

https://cs.wellesley.edu/~cs240/s20/
Processor: Data Path Components

Instruction Fetch and Decode → Registers → ALU → Memory

Sequential Logic
Bistable latches

Suppose we somehow get a 1 (or a 0?) on here.
**SR latch**

<table>
<thead>
<tr>
<th>S</th>
<th>R</th>
<th>Q</th>
<th>Q'</th>
<th>Q (stable)</th>
<th>Q' (stable)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>?</td>
<td>?</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>?</td>
<td>?</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Sequential Logic
SR latch
if $C = 0$, then SR latch stores current value of $Q$.

if $C = 1$, then $D$ flows to $Q$:

  if $D = 0$, then $R = 1$ and $S = 0$, $Q = 0$

  if $D = 1$, then $R = 0$ and $S = 1$, $Q = 1$
Time matters!

Assume Q has an initial state of 0
**Clocks**

**Clock:** free-running signal with fixed cycle time = clock period = $T$. **Clock frequency** = $1 / \text{clock period}$

A clock controls when to update a sequential logic element's state.
Synchronous systems

Inputs to state elements must be valid on active clock edge.
D flip-flop with falling-edge trigger

Clock

Can still read $Q_{now}$

follower stores $E$ as $Q$

leader stores $D$ as $E$

Time

Sequential Logic
Time matters!

Assume Q and E have an initial state of 0
Reading and writing in the same cycle

Assume $Q$ is initially 0.
D flip-flop = one bit of storage
A 1-nybble* register
(a 4-bit hardware storage cell)

*Half a byte!
Register file

Array of registers, with register selectors, write/read control, input port for writing data, output ports for reading data.

- Read register selector 1
- Read register selector 2
- Write register selector
- Write data
- Read data 1
- Read data 2

Write port

0 = read
1 = write

r = \log_2 \text{number of registers}
w = \text{bits in word}
Read ports
(data out)

Read register number 1

Read register number 2

Register 0
Register 1
\ldots
Register \( n-2 \)
Register \( n-1 \)

Read data 1

Read data 2
Write port (data in)

The write port for a register file is implemented with a decoder that is used with the write signal to generate the \( C \) input to the registers. All three inputs (the register number, the data, and the write signal) will have setup and hold-time constraints that ensure that the correct data is written into the register file.

Write port diagram:
- **Write control**
- **Clock**
- **Register number**
- **Incoming data**

Decoder:
- \( n \)-to-\( 2^n \)

Registers:
- Register 0
- Register 1
- Register \( n-2 \)
- Register \( n-1 \)

Chapter 4 makes extensive use of such logic.

Specifying Sequential Logic in Verilog

To specify sequential logic in Verilog, we must understand how to generate a clock, how to describe when a value is written into a register, and how to specify sequential control. Let us start by specifying a clock. A clock is not a predefined object in Verilog; instead, we generate a clock by using the Verilog notation
\[
\#n \text{ before a statement; this causes a delay of } n \text{ simulation time steps before the execution of the statement.}
\]

In most Verilog simulators, it is also possible to generate a clock as an external input, allowing the user to specify at simulation time the number of clock cycles during which to run a simulation.

The code in Figure C.8.10 implements a simple clock that is high or low for one simulation unit and then switches state. We use the delay capability and blocking assignment to implement the clock.
RAM (Random Access Memory)

Similar to register file, except...
16 x 4 RAM

4-bit address
1101
4 to 16 decoder

data out