Sequential Logic and State

Output depends on inputs and stored values. (vs. combinational logic: output depends only on inputs)

Elements to store values: latches, flip-flops, registers, memory
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/
Processor: Data Path Components

Instruction Fetch and Decode

Registers

ALU

Memory

Sequential Logic
Unstable circuit

How can $Q = Q'$?

Have this issue with any odd number of inverters in a loop.
Bistable latches

Things are more sensible with an even number of inverters in a loop.

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

![SR latch diagram]

<table>
<thead>
<tr>
<th>S</th>
<th>R</th>
<th>Q\text{_prev}</th>
<th>Q'\text{_prev}</th>
<th>Q\text{_next} (stable)</th>
<th>Q'\text{_next} (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>\text{any}</td>
<td>\text{any}</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>\text{any}</td>
<td>\text{any}</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>\text{any}</td>
<td>\text{any}</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

violates invariant that Q and Q' are inverses!
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 $S = 1$, $Q = 1$

Notes:
• Data bit $D$ replaces $S$ & $R$: it’s the bit value we want to store when Clock = 1
  • Internally, Data bit $D$ prevents bad case of $S = R = 1$
• This logic is level-triggered; as long as Clock = 1, changes to $D$ have impact
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 / 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

Assume $Q = Q_{\text{now}}$

leader stores $D = Q_{\text{next}}$ as $E$

follower stores $E = Q_{\text{next}}$ as $Q$

Q still = $Q_{\text{now}}$

Q is now $Q_{\text{next}}$
Time matters!

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

Assume Q is initially 0.

Moral: It’s OK to use the current output Q of a flip-flop as part of the next data input D to the same flip-flop.
D flip-flop = one bit of storage

The bit value of D when C has a falling edge is remembered at Q until the next falling edge of C.
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.

Write? 0 = read 1 = write

r = log$_{2}$ number of registers
w = bits in word

Why 2?

Write port

Read ports

Register file

<table>
<thead>
<tr>
<th>Register address #1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read data 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register address #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read data 2</td>
</tr>
</tbody>
</table>

Write register selector
Write data
Memory Elements: Flip-Flops, Latches, and Registers

FIGURE C.8.7 A register file with two read ports and one write port has five inputs and two outputs. The control input Write is shown in color.

FIGURE C.8.8 The implementation of two read ports for a register file with \( n \) registers can be done with a pair of \( n \)-to-1 multiplexors, each 32 bits wide. The register read number signal is used as the multiplexor selector signal. Figure C.8.9 shows how the write port is implemented.

Register file

Read ports (data out)

Register address #1

\((\log_2 k \text{ bits})\)

Register address #2

\((\log_2 k \text{ bits})\)

Data read from register address #1

\((n \text{ bits})\)

Data read from register address #2

\((n \text{ bits})\)
Memory Elements: Flip-Flops, Latches, and Registers

FIGURE C.8.7 A register file with two read ports and one write port has five inputs and two outputs. The control input Write is shown in color.

FIGURE C.8.8 The implementation of two read ports for a register file with \( n \) registers can be done with a pair of \( n \)-to-1 multiplexors, each 32 bits wide. The register read number signal is used as the multiplexor selector signal. Figure C.8.9 shows how the write port is implemented.

Read ports (data out)

Register address #1
\((\log_2 k \text{ bits})\)

Register address #2
\((\log_2 k \text{ bits})\)

Data read from register address #1
\((n \text{ bits})\)

Data read from register address #2
\((n \text{ bits})\)
**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.

![Diagram of write port](image)

### Sequential Logic

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 before a statement; this causes a delay of \( n \) 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.
Write port (data in)

Address of register to write to ($log_2 k$ bits)

Data to write ($n$ bits)

write control

clock

$m$-to-$2^m$ decoder

$0$

$1$

$k - 2$

$k - 1$

Register 0

Register 1

$\vdots$

Register $k - 2$

Register $k - 1$
RAM (Random Access Memory)

- A is number of words in RAM
- Specify the desired word by an address of size $\log_2 A$
- B is the width of each word (in bits)

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

4-bit address

1101

4 to 16 decoder

data out