Latches, Flip-flops, and Registers

**Sequential logic:** fundamental elements to store values
Output depends on inputs *and stored values.*

(vs. combinational logic: output depends only on inputs)
Processor: Data Path Components

Instruction Fetch and Decode → Registers #2 → ALU #1 → Memory

Diagram showing the data path components of a processor, including instruction fetch and decoding, registers, ALU, and memory.
Bistable latches

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

<table>
<thead>
<tr>
<th></th>
<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>0</td>
<td>0</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>?</td>
<td>?</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>?</td>
<td>?</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

---

**Set**

**Reset**
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!
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

Can still read $Q_{now}$

$Q_{next}$ becomes $Q_{now}$

leader stores D as E

follower stores E as Q

Time
Time matters!
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)
Register file

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

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.
Write port (data in)

Write control → decoder
Clock
Register number

$n$-to-$2^n$ decoder

0
1

$n-2$

$n-1$

Incoming data

Register 0
$C$
$D$

Register 1
$C$
$D$

Register $n-2$
$C$
$D$

Register $n-1$
$C$
$D$

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 \( \#n \) 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.

C-56
Appendix C The Basics of Logic Design
RAM (Random Access Memory)

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

4-bit address: 1101
4 to 16 decoder

Data out