Sequential Logic and State

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

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

Motivation

Now that we have ALUs to perform computations, how do we store the results?

How do we calculate different results over time?

Answer: we need circuits that depend not just on inputs, but also on prior state = Sequential Logic

Example from previous lab

Can you think of an example from lab of a sequential circuit you used?

Hint: previous button pushes are past state.

Nobody has responded yet.

Hang tight! Responses are coming in.
**Goal for this section**

Design a circuit state that holds a state over time

- We should be able to set the value to 0 or 1
- We should be able to read the value off the circuit

---

**First attempt: Unstable circuit**

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

**Second attempt: stable circuit?**

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

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

Now stable, but how do we set the value?
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.

Change to a 2-input gates so that we can set updated values to be stored.

SR latch

Set

Reset

<table>
<thead>
<tr>
<th>S</th>
<th>R</th>
<th>Q_{prev}</th>
<th>Q'_{prev}</th>
<th>Q_{next (stable)}</th>
<th>Q'_{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>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>any</td>
<td>any</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>any</td>
<td>any</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>any</td>
<td>any</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Violates invariant that $Q$ and $Q'$ are inverses!

SR latch

Move from the circuit we built to the canonical form.

Meets our goals:

- Able to set the value to 0 or 1
- Able to read the value off the circuit
SR latch

Meets our goals:
- Able to set the value to 0 or 1
- Able to read the value off the circuit

But:
- Ambiguous when \( S = 1 \) and \( R = 1 \)
- No distinction between new value and timing
D latch

Goals:
- Only 1 bit for data
- Control over timing

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 flow to outputs

In general: 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.

Time matters!

Assume Q has an initial state of 0

Aside: “Clock frequency”

Clock frequency
= 1 / period = 1 / s = Hz

Typical CPU: 3-4 GHz
Synchronous systems

Inputs to state elements must be **valid** on active clock edge.

Time matters! **D flip-flop** with falling-edge trigger

Assume $Q$ and $E$ have an initial state of 0

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**

D Flip-Flop

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

---

**Registers**

Assembly code (later this semester): `addq $rdi, $rsi`

---

**A 1-nybble* register**

*Half a byte!*

Write value

Clock line may be indicated

---

**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)

Register address #1 (log₂ k bits)

Register 0
Register 1
...
Register k - 2
Register k - 1

Data read from register address #1 (n bits)

Read ports (data out)

Register address #2 (log₂ k bits)

Register 0
Register 1
...
Register k - 2
Register k - 1

Data read from register address #2 (n bits)

Write port (data in)

Address of register to write to (log₂ k bits)

Write control clock

Data to write (n bits)

Write port (data in)

Address of register to write to (log₂ k bits)

Write control clock

Data to write (n bits)
Registers summary

For our purposes: implemented with flip-flops
Very fast access
Limited in size:
  - Need an m-to-$2^m$ decoder
  - CPUs typically have ~10s of words of register storage

We'll think about at a higher level of abstraction
Designed to handle a much larger amount of data
CPUs can have millions-billions of words of memory storage

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)

16 x 4 RAM

- 4-bit address
- 4 to 16 decoder
- Data out
- 4-bit address 1101