Basics of Logic Design:
Storage Elements and the Register File
(Sequential Logic)

Tyler Bletsch
Duke University

Slides are derived from work by
Daniel J. Sorin (Duke), Alvy Lebeck (Duke), and Drew Hilton (Duke)
So far…

- We can make logic to compute “math”
  - Add, subtract ... and you can do mul/div in 350
    - Assume for now that mul/div can be built
  - Bitwise: AND, OR, NOT,...
  - Shifts (left or right)
  - Selection (MUX)
    - ...pretty much anything
- But processors need state (hold value)
  - Registers
  - ...
Storage

• All the circuits we looked at so far are **combinational circuits**: the output is a Boolean function of the inputs.

• We need circuits that can remember values (registers, memory)

• The output of the circuit is a function of the input **and** a function of a stored value (state)

• Circuits with storage are called **sequential circuits**

• Key to storage: feedback loops from outputs to inputs
Ideal Storage – Where We’re Headed

• Ultimately, we want something that can hold 1 bit and we want to control when it is re-written

• However, instead of just giving it to you as a magic black box, we’re going to first dig a bit into the box
  • I will not test you on the insides of the “flip flop”
Building up to the D Flip-Flop and beyond

SR Latch
(too awkward)

D Latch
(bad timing)

D Flip-Flop
(okay but only one bit)

Register
(nice!)
FF Step #1: NOR-based Set-Reset (SR) Latch

Don’t set both S & R to 1. Seriously, don’t do it.
Set-Reset Latch (Continued)
Set-Reset Latch (Continued)

Set Signal Goes High: 
- $S$ goes from 0 to 1.

Output Signal Goes High: 
- $Q$ goes from 0 to 1.

Time: 
- The timing diagram shows the transition of signals over time.
Set-Reset Latch (Continued)

Set Signal Goes Low

Output Signal Stays High

Time
Set-Reset Latch (Continued)

Until Reset Signal Goes High

Then Output Signal Goes Low
SR Latch

- Downside: S and R at once = chaos
- Downside: Bad interface

- So let’s build on it to do better
Building up to the D Flip-Flop and beyond

SR Latch
(too awkward)

D Latch
(bad timing)

D Flip-Flop
(okay but only one bit)

Register
(nice!)
FF Step #2: Data Latch ("D Latch")

Starting with SR Latch
Data Latch (D Latch)

Starting with SR Latch

Change interface to
Data + Enable (D + E)

If E=0, then R=S=0.
If E=1, then S=D and R=!D
Data Latch (D Latch)

<table>
<thead>
<tr>
<th>D</th>
<th>E</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>Q</td>
</tr>
</tbody>
</table>

D goes high
E stays high
E goes low
D "latched"
Q stays as output
Data Latch (D Latch)

<table>
<thead>
<tr>
<th>D</th>
<th>E</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>Q</td>
</tr>
</tbody>
</table>

Diagram:

- **Enable (E)**
- **Data (D)**
- **Q**
- **Q̅**

**Time Diagram**

- **E goes low**
- **Output unchanged**
- **By changes to D**

Does not affect Output
Data Latch (D Latch)

<table>
<thead>
<tr>
<th>D</th>
<th>E</th>
<th>Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>Q</td>
</tr>
</tbody>
</table>

- **E** goes high
- **D“latched”**
- Becomes new output
Data Latch (D Latch)

Enable

Data

Q

\[
\begin{array}{ccc}
D & E & Q \\
0 & 1 & 0 \\
1 & 1 & 1 \\
- & 0 & Q \\
\end{array}
\]

Slight Delay

(Logic gates take time)
Logic Takes Time

• Logic takes time:
  • Gate delays: delay to switch each gate
  • Wire delays: delay for signal to travel down wire
  • Other factors (not going into them here)

• Need to make sure that signals timing is right
  • Don’t want to have races or wacky conditions..
Clocks

- Processors have a clock:
  - Alternates 0 1 0 1
  - Like the processor’s internal metronome
  - Latch $\rightarrow$ logic $\rightarrow$ latch in one clock cycle

- One clock cycle

- 3.4 GHz processor = 3.4 Billion clock cycles/sec
• First thoughts: Level Triggered
  • Latch enabled when clock is high
  • Hold value when clock is low
Strawman: Level Triggered

- How we’d like this to work
  - Clock is low, all values stable
Strawman: Level Triggered

- How we’d like this to work
  - Clock goes high, latches capture and xmit new val

![Diagram](image_url)
Strawman: Level Triggered

- How we’d like this to work
  - Signals work their way through logic w/ high clk
Strawman: Level Triggered

- How we’d like this to work
  - Clock goes low before signals reach next latch
Strawman: Level Triggered

• How we’d like this to work
  • Clock goes low before signals reach next latch
Strawman: Level Triggered

- How we’d like this to work
  - Everything stable before clk goes high

```
Clk
```

```
010
```

```
Logic
```

```
000
```

```
100
```

```
111
```

```
D latch
```

```
0
```

```
D latch
```

```
3
```

```
3
```

```
D
```

```
E
```

```
Q
```

```
Q
```

```
Q
```

```
Q
```

```
E
```

```
Clk
```

```
Clk
```
Strawman: Level Triggered

- How we’d like this to work
  - Clk goes high again, repeat
Strawman: Level Triggered

- Problem: What if signal reaches latch too early?
  - I.e., while clk is still high
Strawman: Level Triggered

- Problem: What if signal reaches latch too early?
  - Signal goes right through latch, into next stage.

![Logic Diagram]
That would be bad...

- Getting into a stage too early is bad
  - Something else is going on there $\rightarrow$ corrupted
  - Also may be a loop with one latch

- Consider incrementing counter (or PC)
  - Too fast: increment twice? Eeek...

```
D  Q
D latch E Q

+1

001 010

3
```
Building up to the D Flip-Flop and beyond

SR Latch
(too awkward)

D Latch
(bad timing)

D Flip-Flop
(okay but only one bit)

Register
(nice!)
FF Step #4: Edge Triggered

- Instead of level triggered
  - Latch a new value at a clock level (high or low)
- We use edge triggered
  - Latch a value at a clock edge (rising or falling)

**Falling Edges**

**Rising Edges**
Our Ultimate Goal: D Flip-Flop

- Rising edge triggered D Flip-flop
  - Two D Latches w/ opposite clking of enables
D Flip-Flop

- Rising edge triggered D Flip-flop
  - Two D Latches w/ opposite clocking of enables
  - On Low Clk, first latch enabled (propagates value)
    - Second not enabled, maintains value
D Flip-Flop

- Rising edge triggered D Flip-flop
  - Two D Latches w/ opposite clking of enables
  - On Low Clk, first latch enabled (propagates value)
    - Second not enabled, maintains value
  - On High Clk, second latch enabled
    - First latch not enabled, maintains value
No possibility of “races” anymore

- Even if I put 2 DFFs back-to-back...
- By the time signal gets through 2\textsuperscript{nd} latch of 1\textsuperscript{st} DFF
  1\textsuperscript{st} latch of 2\textsuperscript{nd} DFF is disabled

Still must ensure signals reach DFF before clk rises

- Important concern in logic design “making timing”
D Flip-flops (continued…)

• Could also do falling edge triggered
  • Switch which latch has NOT on clk

• D Flip-flop is ubiquitous
  • Typically people just say “latch” and mean DFF
  • Which edge: doesn’t matter
    • As long as consistent in entire design
    • We’ll use rising edge
D flip flops

- Generally don’t draw clk input
  - Have one global clk, assume it goes there
  - Often see > as symbol meaning clk

- Maybe have explicit enable
  - Might not want to write every cycle
  - If no enable signal shown, implies always enabled
  - Inside DFF, E signal is ANDed with Clk:
    if E is off, Clk is ignored (so we don’t commit changes)

- Get output and NOT(output) for “free”
More Storage Than A D FF: Register File

- A MIPS register can be made with 32 flip flops
- One register can store one 32-bit value
- So do we just replicate this 32 times to get the 32 registers for a MIPS processor?
  - Not exactly

- **Register File** (the physical storage for the regs)
  - MIPS register file has 32 32-bit registers
- How do we build a Register File using D Flip-Flops?
- What other components do we need?
Building up to the D Flip-Flop and beyond

- **SR Latch**: (too awkward)
- **D Latch**: (bad timing)
- **D Flip-Flop**: (okay but only one bit)
- **Register**: *(nice!)*
Stick a bunch of DFFs together to make a register

DQ
DFF
E
Q

DQ
DFF
E
Q

DQ
DFF
E
Q

DQ
DFF
E
Q

DQ
DFF
E
Q

DQ
DFF
E
Q

DQ
DFF
E
Q

enable

in0

out0

in1

out1

in2

out2

in31

out31

32 bit reg

D
Q

E
Q
Next evolution: multiple registers

Register

(nice!)

Register File

(Tremendous!)
Register File Design

- Two problems: write and read

  **Writing** the registers
  - Need to pick which reg
  - Have reg num (e.g., 19)
  - Need to make En19=1
    - En0, En1,... = 0

  **Read**: Use a mux to pick?
  - 32-input mux = slow
  - Need a better method...

- Let’s talk about writing first.
First: A Decoder

- First task: convert binary number to “one hot”
  - $N$ bits in
  - $2^N$ bits out
  - $2^N-1$ bits are 0, 1 bit (matching the input) is 1

![Diagram of a decoder](image-url)
Decoder Logic

- Decoder basically AND gates for each output:
  - $\text{Out}_0$ only on if input 000

3-input gates are fine.
In theory, gates can have any # of inputs
In practice >4 converted to multiple gates
Decoder Logic

- Decoder basically AND gates for each output:
  - $\text{Out}_1$ only on if input 001

Repeat for all outputs: AND together right bits (gets messy fast on a slide)
Register File

• Now we know how to **write**:
  • Use decoder to convert reg # to one hot
  • Send write data to all regs
  • Use one hot encoding of reg # to enable right reg

• Still need to fix **read** side
  • 32 input mux (the way we’ve made it) not realistic
  • To do this: expand our world from \{1,0\} to \{1, 0, Z\}
Kind of like water in a pipe...

- To understand Z, let’s make an analogy
  - Think of a wire as a pipe
    - Has water = 1
    - Has water = 0
  - This wire is 0 (it has no water)
To understand Z, let’s make an analogy
  • Think of a wire as a pipe
    • Has water = 1
    • Has water = 0
  • This wire is 1 (it is full of water)
Kind of like water in a pipe...

- To understand Z, let’s make an analogy
  - Think of a wire as a pipe
    - Has water = 1
    - Has water = 0
  - Suppose a gate drives a 0 onto this wire
    - Think of it as sucking the water out
To understand Z, let’s make an analogy

- Think of a wire as a pipe
  - Has water = 1
  - Has water = 0

Suppose the gate now drives a 1
- Think of it as pumping water in
Remember this rule?

- Remember I told you not to connect two outputs?

- If one gate tries to drive a 1 and the other drives a 0
  - One pumps water in.. The other sucks it out
  - Except it’s electric charge, not water
  - “Short circuit” → lots of current → lots of heat
So this third option: Z

- There is a third possibility: Z ("high impedance")
  - Neither pushing water in, nor sucking it out
  - Just closed off/blocked
  - Prevents electricity from flowing through

- Gate that gives us Z: Tri-state
We’ve had this rule one day... and you break it

It’s ok to connect multiple outputs together
Under one circumstance:

**All but one** must be outputting Z at any time

![Diagram showing the rule with variables and an AND gate connected to multiple outputs.]
Mux, implemented with tri-states

• We can build effectively a mux from tri-states
  • Much more efficient for large #s of inputs (e.g., 32)
Now we can **write** and **read** in one clock cycle!
• What we just saw: **read** port
  • Ability to do one read / clock cycle
  • May want more: read 2 source registers per instr
    • Maybe even more if we do many instrs at once
  • This design: can just replicate port
    • Another decoder
    • Another set of tri-states
    • Another output bus (wire connecting the tri-states)

• Earlier: **write** port
  • Ability to do one write/cycle
  • Could add more: need muxes to pick wr values
FYI: This is not how a modern register file is implemented
- (Though it is how other things are implemented)
- Actually done with SRAM
- We’ll see that later this semester…
Can layout logic to compute things
   Add, subtract,...
Now can store things
   D flip-flops
   Registers
Also understand clocks

Just about ready to make a datapath!