Storage and Clocking

Tyler Bletsch
Duke University

Slides are derived from work by Andrew Hilton (Duke)
VHDL: Behavioral vs Structural

- A few words about VHDL
  - Structural:
    - Spell out at (roughly) gate level
    - Abstract piece into **entities** for abstraction/re-use
    - Very easy to understand what synthesis does to it
  - Behavioral:
    - Spell out at higher level
    - Sequential statements, for loops, process blocks
    - Can be difficult to understand how it synthesizes
      - Difficult to resolve performance issues
Last time…

• Who can remind us what we did last time?
  • Add
  • Subtract
  • Bit shift
  • Floating point
• We can make logic to compute “math”
  • Add, subtract,... (we’ll see multiply/divide later)
  • Bitwise: AND, OR, NOT,...
  • Shifts
  • Selection (MUX)
  • ...pretty much anything

• But processors need state (hold value)
  • Registers
  • ...
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
• Ultimately, we want something that can hold 1 bit and we want to control when it is re-written

“flip flop” = device that holds one bit (0 or 1)

bit to be written ➔ “flip flop” = device that holds one bit (0 or 1) ➔ bit currently being held

bit to control when we write ➔ “flip flop”

• However, instead of just giving it to you as a magic black box, we’re going to first dig a bit into the box
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
Output Signal Goes High
Set-Reset Latch (Continued)

Set Signal Goes Low
Output Signal Stays High
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!)

32 bit reg
FF Step #2: Data Latch ("D Latch")

Starting with SR 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)

- When E goes high, D is "latched".
- Stays as output.

<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>
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>

Does not affect Output

E goes low

Output unchanged

By changes to D
Data Latch (D Latch)

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

E goes high

D "latched"

Becomes new 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>

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
**FF Step #3: Using Level-Triggered D Latches**

- **First thoughts: Level Triggered**
  - Latch enabled when clock is high
  - Hold value when clock is low

This slide describes how D-latches can malfunction because they were level triggered. Real D-flip-flops are *edge-triggered*, and we’re showing you *why* that’s important.
Strawman: Level Triggered

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

This slide describes how D-latches can malfunction because they were level triggered. Real D-flip-flops are *edge-triggered*, and we're showing you *why* that's important.
Strawman: Level Triggered

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

This slide describes how D-latches can malfunction because they were level triggered. Real D-flip-flops are edge-triggered, and we're showing you why that's important.
Strawman: Level Triggered

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

This slide describes how D-latches can malfunction because they were level triggered. Real D-flip-flops are edge-triggered, and we're showing you why that's important.
Strawman: Level Triggered

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

Logic

This slide describes how D-latches can malfunction because they were level triggered. Real D-flip-flops are *edge-triggered*, and we’re showing you *why* that’s important.
Strawman: Level Triggered

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

This slide describes how D-latches can malfunction because they were level triggered. Real D-flip-flops are edge-triggered, and we’re showing you why that’s important.
Strawman: Level Triggered

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

This slide describes how D-latches can malfunction because they were level triggered. Real D-flip-flops are edge-triggered, and we’re showing you why that’s important.
Strawman: Level Triggered

- How we’d like this to work
  - Clk goes high again, repeat

This slide describes how D-latches can malfunction because they were level triggered. Real D-flip-flops are edge-triggered, and we’re showing you why that’s important.
Strawman: Level Triggered

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

This slide describes how D-latches can malfunction because they were level triggered. Real D-flip-flops are edge-triggered, and we're showing you why that's important.
Strawman: Level Triggered

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

This slide describes how D-latches can malfunction because they were level triggered. Real D-flip-flops are *edge-triggered*, and we're showing you *why* that's important.
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...

This slide describes how D-latches can malfunction because they were level triggered. Real D-flip-flops are *edge-triggered*, and we’re showing you why that’s important.
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
• 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
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
D Flip-Flop

- 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”
Making Timing

• Making timing is important in a design
  • If you don’t make timing, your logic won’t compute right

• Synthesis tool (Quartus) tells you what max freq
  • Running above this your logic doesn’t “finish” in time
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

- Get output and NOT(output) for “free”
DFFs in VHDL

x: dffe port map (  
    clk => clk,        --the clock  
    d => someInput,   --the input is d  
    q => theOutput,   --the output is q  
    ena => en,        --the enable  
    clrn => '1',      --clear  
    prn => '1');      --set

Also, comes in “dff” with no enable
A word of advice

```verilog
signal x_q : std_logic;
signal x_d: std_logic;
x : dffe port map (..., d=> x_d, q=>x_q,...);
```

- Use naming convention: x_d, x_q
- Write x_d, read x_q
- Remember new value shows up next cycle
A few words about timing

- Homework 2: VGA Controller
  - Requires certain clock frequency
    - Else won’t control monitor properly
- Quartus will tell you what timing you make
  - Fmax: how fast can this be clocked
  - Tells you your worst timing paths
    - From which dff to which dff
    - Can see in schematic viewer (usually)
- Homework 2
  - Should be plenty of slack
  - But if not...
Fixing timing misses

• Typical approach: reduce logic (gate delays)
  • Better adder?
  • Rethink approach?
  • Change “don’t care” behavior?

• Fix high fanout
  • Duplicate high FO/simple logic

• Also, feel free to ask for help from me/TAs
  • Quartus’s tools to help you fix them aren’t the best
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

D Q
DFF
E \overline{Q}

D Q
DFF
E \overline{Q}

D Q
DFF
E \overline{Q}

D Q
DFF
E \overline{Q}

D Q
DFF
E \overline{Q}

D Q
DFF
E \overline{Q}

D Q
DFF
E \overline{Q}

32 bit reg

E \overline{Q}

in0
D Q
DFF
E \overline{Q}
out0

in1
D Q
DFF
E \overline{Q}
out1

in2
D Q
DFF
E \overline{Q}
out2

in31
D Q
DFF
E \overline{Q}
out31

enable
Next evolution: multiple registers

Register

(nice!)

Register File

(Tremendous!)
Register File

- Can store one value... How about many?

- Register File
  - In processor, holds values it computes on
  - MIPS, 32 32-bit registers

- How do we build a Register File using D Flip-Flops?
- What other components do we need?
Register File: Interface

- **4 inputs**
  - 3 register numbers (5 bit): 2 read, 1 write
  - 1 register write value (32 bits)
- **2 outputs**
  - 2 register values (32 bits)
Register file strawman

- Use a mux to pick read?
  - 32 input mux = slow
  - (other regs not pictured)
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 $En_{19}=1$
      - $En_0, En_1, ... = 0$
  - **Read**: Use a mux to pick?
    - 32-input mux = slow
    - Need a better method...

- Let’s talk about **writing** first.
First task: convert binary number to "one hot"
  - Saw this before
  - Take register number
• 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)
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 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
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 the gate now drives a 1
    - Think of it as pumping water in
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
• 2 inputs: E and D. What does this do?
  • Write truth table for output

<table>
<thead>
<tr>
<th>E</th>
<th>D</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
CMOS: Complementary MOS

2 inputs: E and D. What does this do?

- Write truth table for output
- When E = 1, straightforward

<table>
<thead>
<tr>
<th>E</th>
<th>D</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
CMOS: Complementary MOS

- 2 inputs: E and D. What does this do?
  - Write truth table for output
  - When E = 1, straightforward
  - When E = 0, no connection: Z

<table>
<thead>
<tr>
<th>E</th>
<th>D</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Z</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Z</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
High Impedance: Z

- Z = High Impedance
  - No path to power or ground
  - “Gate” does not produce a 1 or a 0

- Previous slide: tri-state inverter
  - More commonly drawn: tri-state buffer
  - E = enable, D = data

\[
\begin{array}{ccc}
D & E & \text{Out} \\
0 & 1 & 0 \\
1 & 1 & 1 \\
X & 0 & Z \\
\end{array}
\]

X = “Don’t care”
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 its electric charge, not water
  - “Short circuit”—lots of current -> lots of heat
It’s ok to connect multiple outputs together
Under one circumstance:

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

- 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
Minor Detail

- FYI: This is not how a register file is implemented
  - (Though it is how other things are implemented)
  - Actually done with SRAM
  - We’ll see how those work soon
Summary

Can layout logic to compute things
  Add, subtract,...

Now can store things
  D flip-flops
  Registers

Also understand clocks