Combinational Logic and FFs in Verilog

Hardware is made of combinatorial logic and Flip Flops. **Verilog can describe both combinational logic and Flip-Flops.**

Combinatorial Logic	**Flip-Flops (registers)**
• control logic for FSMs • multiplexers • adders, multipliers, dividers	• used as memory devices • indicates current state for FMs • used for pipelining
Implementation	**Implementation**

using **assign**:
`wire c;`
`assign c = a+b;`
 	
using **always@(*)**:
`reg c; //Not a register!`
`always @(*) begin`
`  c = a+b;`
`end`
	use **always@(posedge clk)**:
	`reg c_reg; //A register!`
	`always @(posedge clk) begin`
	`  c_reg <= c;`
	`end`

Always Blocks in Verilog

```
always @(<sensitivity list>) begin
    // code here
end
```

The **sensitivity list** is a list of events that will trigger an always block.

- when provided synchronous signal, will synthesize into a FF
- when provided asynchronous signal, synthesizes into combinational/implicit latch

The + sign is synthesized into an adder.

Combinational Logic

- c changes whenever a or b changes

Executed when a or b change

```
always @(a or b) begin
   c = a+b;
end
```

Synchronous Logic

- c is instantiated as a Flip Flop
- c only changes on a new positive clock edge

Synchronous Design

```
always @(posedge clk) begin
   c <= a+b;
end
```

Implicit Latch, avoid this!!

- recall: latches are memory devices that have an asynchronous enable to update the CS
- synthesized into combinatorial logic and a latch, where b acts as the enable
- a state element is required to implement this. a + b forwarded to a latch.
- a + b is fed into the latch as input
- the b is fed into the latch as enable

Executed when b changes

```
always @(b) begin
   c = a+b;
end
```

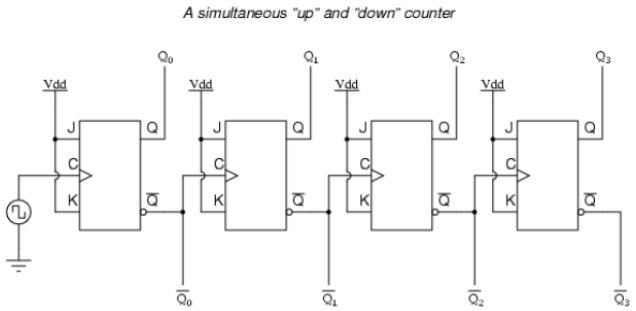
IMPLICIT LATCH, AVOID THIS!

use the wildcard (*) most of the time for combinational logic
use posedge clk for most synchronous designs.

Flip-Flop versus Latch

Why do we like synchronous designs?
Synchronous designs are king because they alleviate timing. Combinatorial logic latency is less than the clock period. Synthesis tools are not capable of generating designs that figure out timing asynchronously.

How would an asynchronous design look like?
Sequential chain of synchronously clocked Flip-Flops:



Most developers used a limited subset of Verilog.
For the most part, always clock blocks are used to generate Flip-Flops.

Synchronous logic blocks

```
always @(posedge clk) begin
   c <= a+b;
end
```

Combinatorial logic blocks

```
assign c = a+b;
```

Combinatorial logic blocks: use when you need to describe complex comb. logic (if/else/for)

```
always @(*) begin
   c = a+b;
end
```

Rules in Verilog to remember

1. Always use **reg** as LHS assignments for always blocks
2. always use **wires** for assign statements
3. **always blocks** can produce both combinatorial and sequential logic
    a. if not all inputs are in the sensitivity list then a latch is synthesized
    b. if no inputs are in the sensitivity list but clk is, a FF is synthesized
    c. always@(clk) without posedge or negedge would be triggered at both edges

There are 3 types of signals in Verilog:

1. arbitrary wire/reg signals
2. clock signal
3. reset signal

Clock and reset signals are 2 special signals that you can specify in Vivado/Xilinx settings.

Assign Statements in Verilog

assign statements physically connect hardware components together.
The two examples below do the exact same thing.

```
assign a = b;
```

```
reg a;
always @(*) begin
   a = b;
end
```

Multiple assign statements are bad!
you can only assign multiple wires onto a single wire if it is a tristate wire that goes through a tristate buffer. We don't wan to fry our FPGAs!

ERROR

```
assign a = b;

always @(*) begin
   a = c;
end
```

BAD

ERROR

```
assign a = b;

assign a = c;
```

BAD

Only Behavioural

```
always @(*) begin
   a = b;
end

always @(*) begin
   a = c;
end
```

BAD

```
always @(*) begin
   a = b;
   a = c;
end
```

No Error but still BAD

Synthesis tool will detect and throw and error because there are multiple assignments.

ERROR

```
assign a = b;

assign a = c;
```

BAD

Only Behavioural

```
always @(*) begin
   a = b;
end

always @(*) begin
   a = c;
end
```

BAD

a would have to be declared as both a wire and a reg. Also, a cannot be assigned 2 different values.

ERROR

```
assign a = b;

always @(*) begin
   a = c;
end
```

BAD

The first assignment is ignored because it is overwritten by the second. Bad!

```
always @(*) begin
   a = b;
   a = c;
end
```

No Error but still BAD

Only Behavioural

```
always @(*) begin
   a = b;
end

always @(*) begin
   a = c;
end
```

BAD

Blocking vs. non-blocking assignments

There are 2 assignment types in Verilog. Assignment inputs are from the current cycle, assignment outputs are for the next cycle.

b <= b; //assign b with the value of b in the last cycle

Blocking (=)	Non-Blocking (<=)

• multiple = are executed iteratively
• can be used outside or inside **always/initial** blocks
• value is assigned immediately
 	
▪ a = b & c & d
vs.
▪ tmp = c & d
▪ a = b & tmp
 	
When to use blocking (=)
• use in combinatorial blocks
• used to describe sequential hardware
• use to break complex combinatorial logic operations into multiple lines (sequential logic chain)
	• multiple <= are executed concurrently
	• can only be used in an **always** or **initial** block. LHS must be a **reg**
	
	The following does the same thing:
	a <= b; vs. b <= a;
	b <= a; a <= b;
	the a and b on the RHS are from the previous clock cycle.
	
	When to use non-blocking (<=)
	• use in synchronous blocks
	• use when order is irrelevant, b/c they happen in parallel
	• try not to use non-blocking assignment for combinational logic

How to structure your modules:

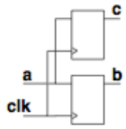
→ Use different assignments/always@(*) blocks for each signal, one for each LHS value.
→ Use an always@(clk) to assign values to registers.

- assign each variable only 1 time in your code.
- keep combinatorial logic and sequential logic in separate always blocks
- **Do not mix blocking and non-blocking assignments in always blocks**. Bad coding style!

[Example: DON'T DO THIS]

a goes directly into b b/c the code implies c and b are updated at the same clock cycle. might as well assign a to both c and b.

```
always @ (posedge clk)
  begin
    c = a;
    b = c;
  end
```



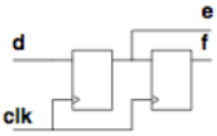
If the intention is to have parallel FFs, use the following:

```
always @(posedge clk) begin
    c<= a;
    b<= a;
end
```

[Example: DO THIS]

2 FFs are synthesized in series/sequence. The current state of e is the last state of d. the current state of f is the last state of e. It takes 2 clock cycles for the value of d to be assigned to f. Acts as a shift register. Note that the block is only evaluated on the positive clock edge.

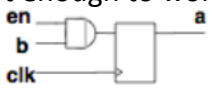
```
always @ (posedge clk)
  begin
    e <= d;
    f <= e;
  end
```



[Example: DO THIS ONLY IF YOU HAVE TO]

Synthesizes as a synchronous block. When if evaluates to true, a is assigned twice. Icky coding! Synthesis tools are smart enough to work around it.

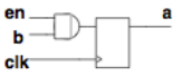
```
always @ (posedge clk)
  begin
    a <= 1'b0;
    if (en == 1'b1)
      a <= b;
  end
```



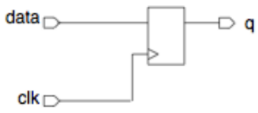
[Example: DO THIS]

Here there is no dual assignment to a. Maybe consider splitting it into 2 always blocks.

```
always @ (posedge clk)
  begin
    if (en == 1'b1)
      a <= b;
    else
      a <= 1'b0;
  end
```



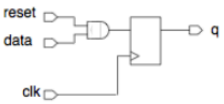
Implementing a D-Flop




```
always @(posedge clk) begin
   q <= data;
end
```

Implementing a D-Flop with Synchronous Reset

reset signal is often implemented as active low. This design adds gate delay and is less power efficient.



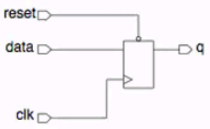
Assume: reset is active low, better name: res_n

```
always @(posedge clk) begin
   if (!res_n)
      q <= 0;
   else
      q <= data;
end
```

```
always @(posedge clk) begin
   if (!res_n) begin
      q <= 0;
   end else begin
      q <= data;
   end
end
```

Implementing a D-Flop with Asynchronous Reset

Better code style for declaring FFs. The asynchronous design means there isn't gate delay added between FFs.




```
always @(posedge clk or negedge res_n) begin
   if (!res_n)
      q <= 0;
   else
      q <= data;
end
```

Reminder: use non-blocking assignment for FFs. The implementations below lead to different hardware.

```
always @(posedge clk) begin
   a = b;
   c = a;
end
```

Bad

```
always @(posedge clk) begin
   a <= b;
   c <= a;
end
```

Good

Ring Counter in Verilog

2 FFs storing a and B that form a loop.

```
always @(posedge clk) begin
   b <= a;
   a <= b;
end
```

If/else in Verilog

- control flow element in Verilog
- begin and end act as curly braces, must be included for if/else more than 1 line
- if/else statements must be in an always or initial block.

[Example: dealing with fanout burden]

- If the data1 signal drives many logic blocks assigning it to multiple FFs to maintain driving strength
  - timing issues can arise from 1 signal fanned out too much
  - signal driving strength comes from Vdd that drives the FF output
  - more combinational logic driven by a signal means less strong signal.

```
always @(posedge clk or negedge res_n) begin
   if (!res_n)
   begin
      q1 <= 0;
      q2 <= 0;
   end
   else
   begin
      q1 <= data1;
      q2 <= data1;
   end
end
```

Multiplexer in Verilog:

another use of if/else statement. Below a 2-x multiplexer is implemented. n-1 multiplexers are built by cascading multiple if/else statements

```
always @(*) begin
   if (sel)
      z = in1;
   else
      z = in0;
end
```

FPGAs have custom hardware for multiplexors.