

How to code Synthesizable RTL

Lecturer: Prof. Adam Teman

13 December 2021



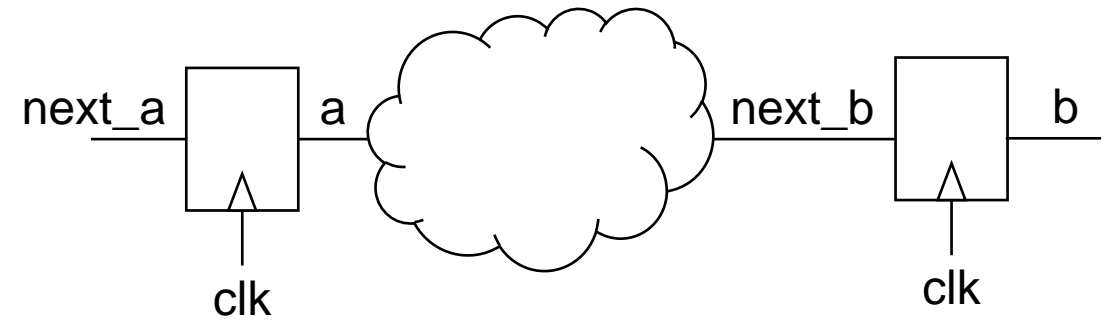
Emerging Nanoscaled
Integrated Circuits and Systems Labs

The Alexander Kofkin
Faculty of Engineering
Bar-Ilan University



Disclaimer: This course was prepared, in its entirety, by Adam Teman. Many materials were copied from sources freely available on the internet. When possible, these sources have been cited; however, some references may have been cited incorrectly or overlooked. If you feel that a picture, graph, or code example has been copied from you and either needs to be cited or removed, please feel free to email adam.teman@biu.ac.il and I will address this as soon as possible.

Reminder: What is RTL?

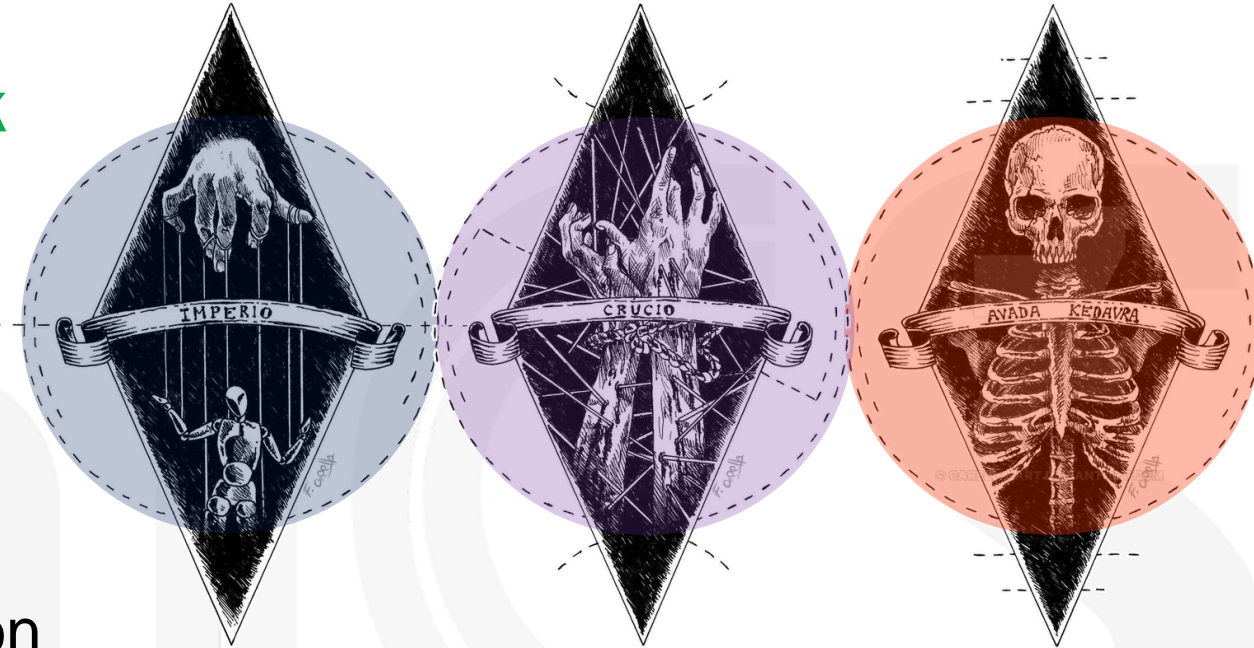


- RTL stands for “**Register Transfer Logic**”
 - In other words, it is a description of a sequential design that passes data between registers, possibly applying some logic to the data on the way.
- RTL is the **synthesizable subset** of a hardware-description language
 - HDLs (e.g., Verilog, VHDL, System Verilog) usually are *Turing Complete* programming languages. In other words, they can “do it all”.
 - But **not everything can be implemented in hardware** (i.e., with logic gates).
 - Therefore, *only a subset* of the HDL is considered “**Synthesizable**”
- RTL coding is not straightforward
 - Veteran logic designers don’t understand why it’s **so complicated**, but noobs **just can’t get it right** until the coin finally drops.
- This lecture is to **try and help you get it right**, despite your intuition!



The Unforgiveable Rules

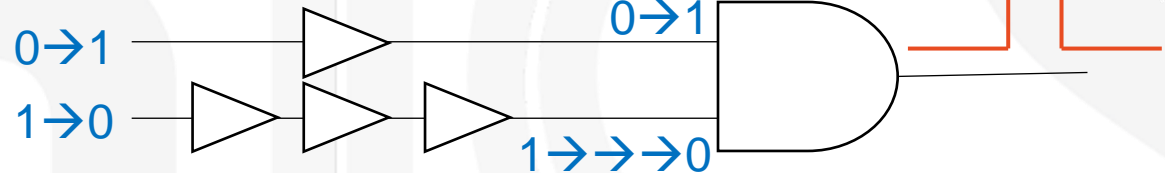
- **Never put logic on reset or on the clock**
 - Never mix reset types
 - Never create clock domain crossings
- **Do Not Infer Latches**
 - Every if has an else
 - Full case statements
 - LHS for every signal for each condition
- **Assignment (always blocks)**
 - Combinational (`always@*`) → blocking (`=`) assignment
 - Sequential (`always@posedge`) → non-blocking (`<=`) assignment
 - **Always separate** sequential and combinational logic
 - **Never assign** a signal (LHS) from **more than one always block**.



Illustrations by Carella Art

No Logic on reset (or clock)

- The **reset** and **clock** signals are not just any signal and they need to be *handled with care*
 - **Clock** signals should only be used to clock registers
 - **Reset** signals should only be used to reset registers
- **Logic glitches** are a characteristic, not a design error
- Any **glitch** on a clock or reset signal is **catastrophic!**
 - A glitch on the clock causes an unwanted (and unexpected) data sampling.
 - A glitch on the reset causes flops to reset accidentally.



- **Therefore, no logic on reset and clock signals!**

```
assign something = a && reset ;  
  
always@*  
  case (state)  
    1'b1011: if (b || reset )  
              next_state = idle;
```

No Logic on Reset – Emphasized Example

- To provide an example of a “**no no**” that can easily be overlooked, let’s assume you have an input that you want to sample at the beginning of a process.
 - The intuitive (**wrong**) way to do it would be something like this:

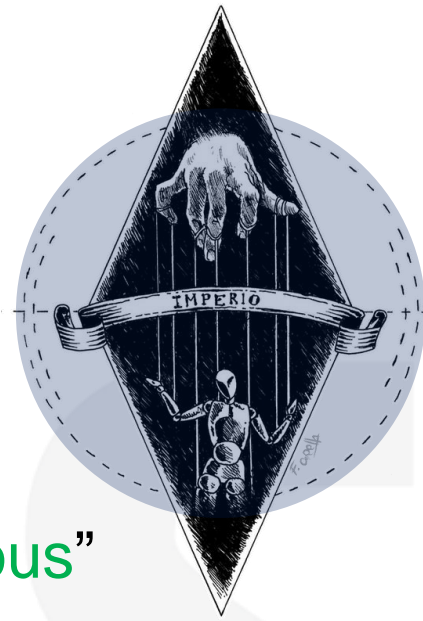
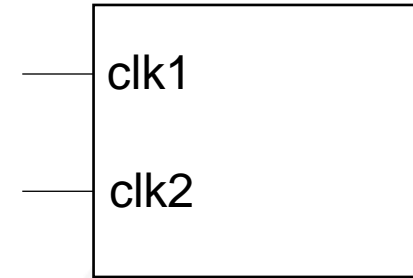
```
input in;
reg in_sampled;
always @(posedge clk or negedge rst_)
    if (!rst_) in_sampled <= in;
    else ...
```
 - But remember that we need to **map** a synchronous block to a flip flop. A flip flop can be reset/set to ‘0’ or ‘1’ and in is a **non-constant** signal.
 - Therefore, the synthesizer would need to decide to create a **set** or **reset** signal during reset, depending on the value of **in**. **This is logic on reset!!!**
- Instead, make an “**initialize**” state and a “**start**” state:

```
always@*
    case state:
        INIT: begin
            next_in = in;
            next_state = START;
            ...
```

```
always @(posedge clk or negedge rst_)
    if (!rst_)
        state <= INIT;
        in_sampled <= 0;
    else
        state <= next_state;
        in_sampled <= next_in;
```


No Clock Domain Crossings

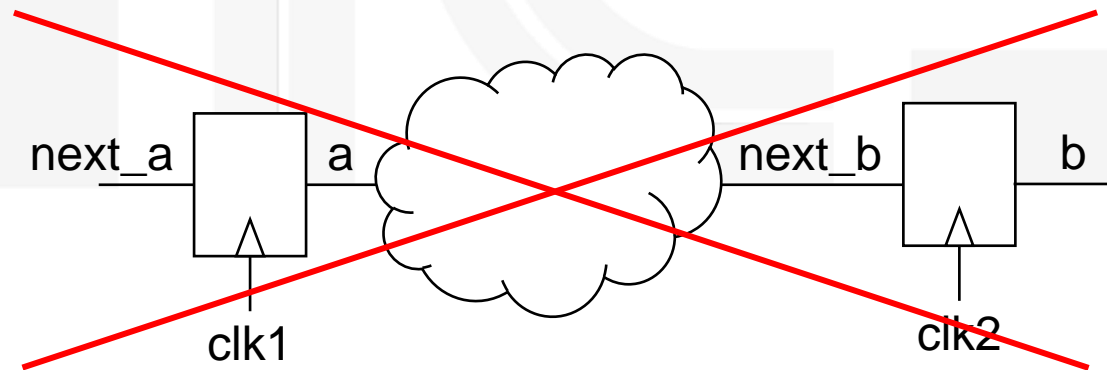
- **Some (most) designs have more than one clock**
 - These clocks may have a different source
 - They may run at different frequencies
 - We **cannot know the phase between them** → they are “**asynchronous**”
- You **cannot have a path** between asynchronous clocks, i.e.:
 - This is known as a **clock domain crossing**



```
always @(posedge clk1)
    a <= next_a;

always @(posedge clk2)
    b <= next_b;

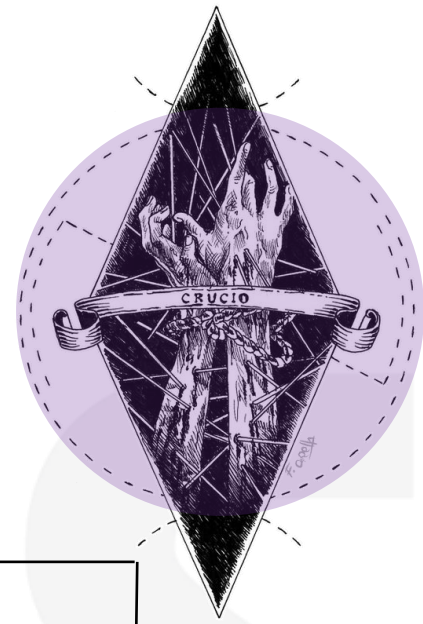
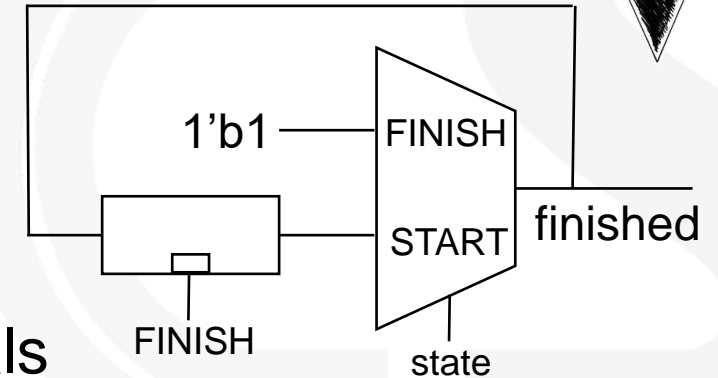
always @*
    next_b = a;
```



No Latch Inference

- The only way to “remember” a value, is with a register (i.e., **flip flop** or **latch**)
 - If we accidentally ask to remember a value, a latch will be **inferred**
- This is due to not assigning (LHS) for all conditions
 - A **combinational if without an else**.
 - A **missing case** option (and no **default**)
 - An **if/else/case without assignment** to all LHS signals
 - For all of the above, we have to keep the value for the **non-defined condition**.
- A similar problem is a missing signal in the sensitivity list
 - Essentially “*don’t change the output when this signal changes*”
 - The **synthesizer** may ignore this, but the **simulator** will adhere to it!
 - Just use **always@*** and this will not happen!

```
always @*
  case (state)
    START: begin
      next_state=FINISH;
    end
    FINISH: begin
      next_state=START;
      finished = 1'b1;
    end
  endcase
```

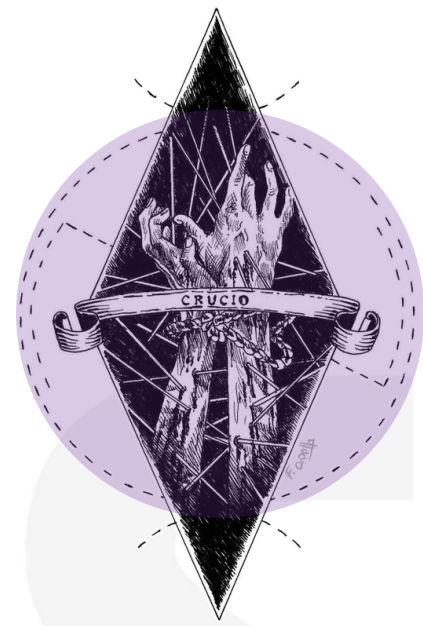


Default values

- To ensure no **latch inference**, just **assign default values** for all combinatorial assignments:
 - At the beginning of an `always@*` block **assign** all **LHS** to a default value
 - **Overwrite** the default value, as necessary, within following `if/else/case` conditions
- Also, it's good practice to provide flip flops with a reset value

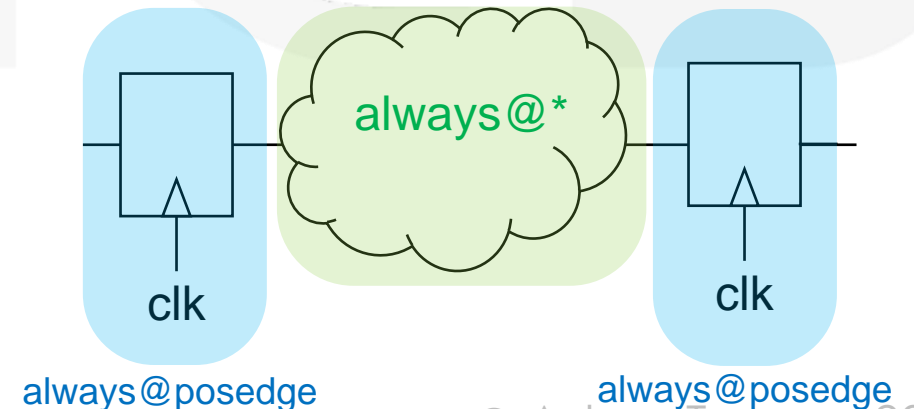
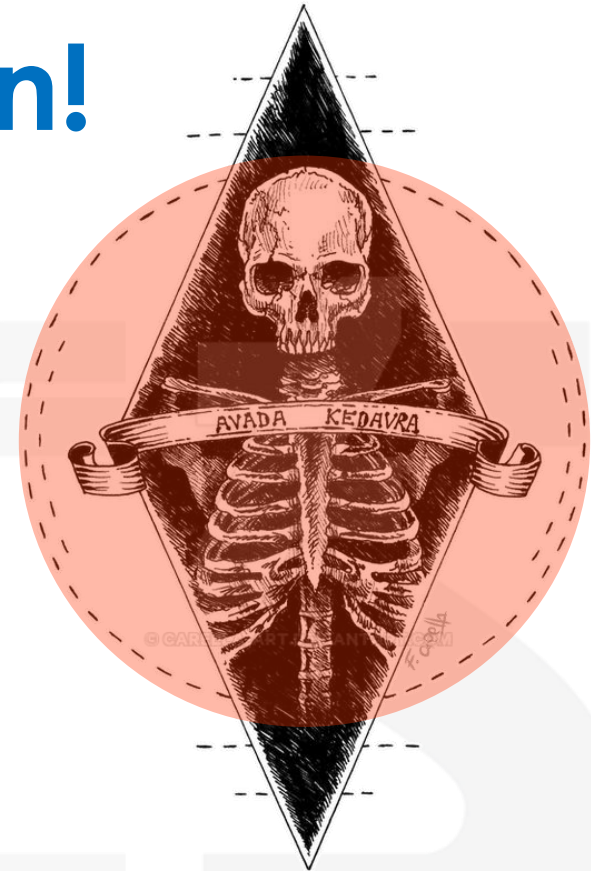
```
always @(posedge clk or negedge rst_)  
    if (~rst_)  
        state <= START;  
    else  
        ...
```

```
always @*  
begin  
    next_a = 1'b0;  
    next_b = 1'b1;  
    next_c = 1'b0;  
    case (state)  
        STATE1: next_a=1'b1;  
        STATE2: next_b=1'b0;  
        STATE3: next_c=1'b1;  
        ...  
        ...
```



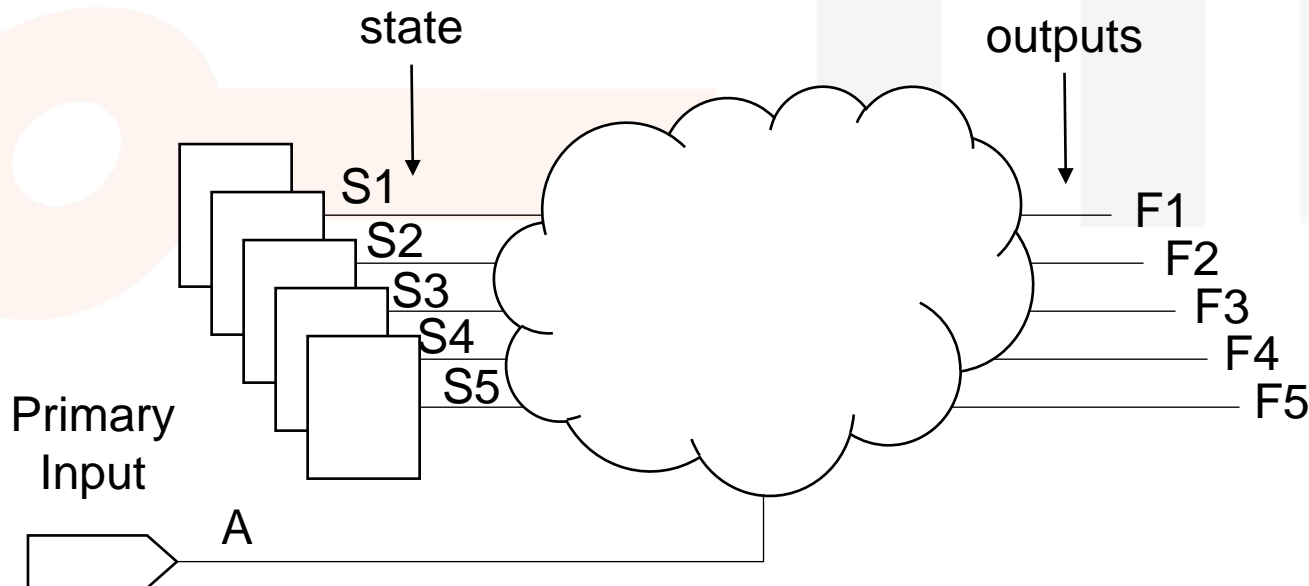
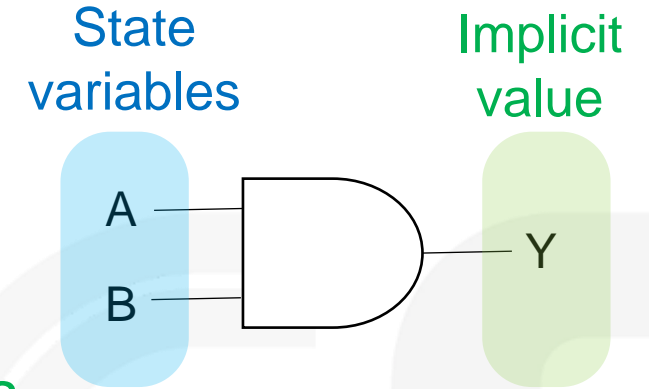
And finally, seq/comb separation!

- I not only told you that this is important, but now I have equated it to the killing curse, **Avada Kedavra**.
- **Google** will give you piles of HDL code that *does not follow this guideline*. But I promise you that if you don't follow it, **you will have to rewrite your code**.
- The guideline is simple:
 - **Sequential logic** is defined within simple `always@posedge` blocks that **map directly** to standard cells from the library.
 - Everything else (i.e., **combinatorial logic**) is defined using `assigns` and `always@*` blocks.
 - In other words, there is a **clear separation** between **sequential** and **combinatorial** logic.



The “State” of a system

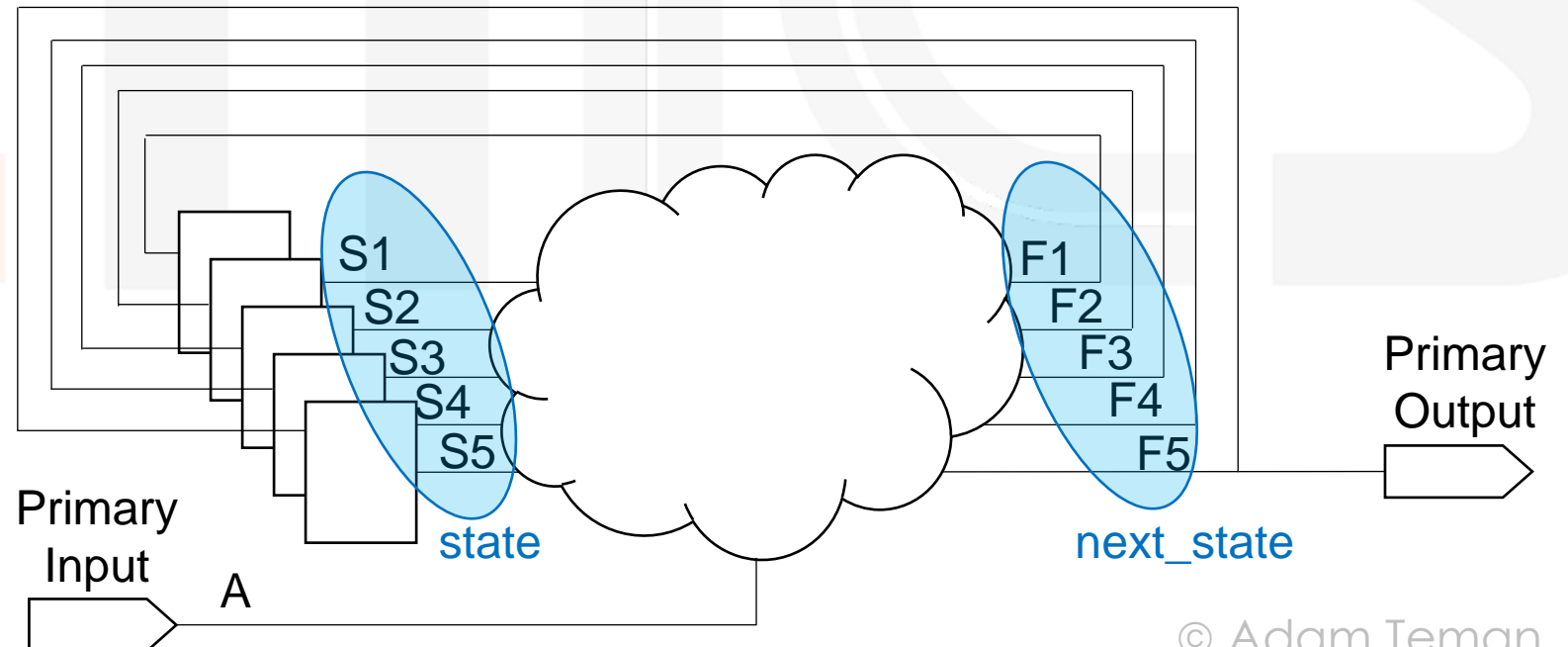
- A sequential system has **state**
 - “**State**” is the **minimum set of variables** required to know what **all values in the system** are.
 - Therefore, the state is the collection of **all the registers** along with the **primary inputs** to the design.
 - Based on the state of the system, we can write a Boolean function (truth table) for **every internal value** in our design



A	S1	S2	S3	S4	S5	F1	F2	F3	F4	F5
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	1	1	1	1	0	0
0	0	0	0	1	0	0	0	0	1	1
1	1	1	1	1	1	0	1	0	1	0

The “State” of a system

- Note that the *entire set of registers* comprises the state
- In other words, the output functions (F1-F5 on the previous slide), can either be:
 - Primary outputs of the design (very few of these)
 - Inputs to state registers
- So the combinatorial logic is *fed back* into the state registers!
- This will be *sampled* on the rising edge of the clock.
 - In other words, the combinatorial logic output is the “next state”

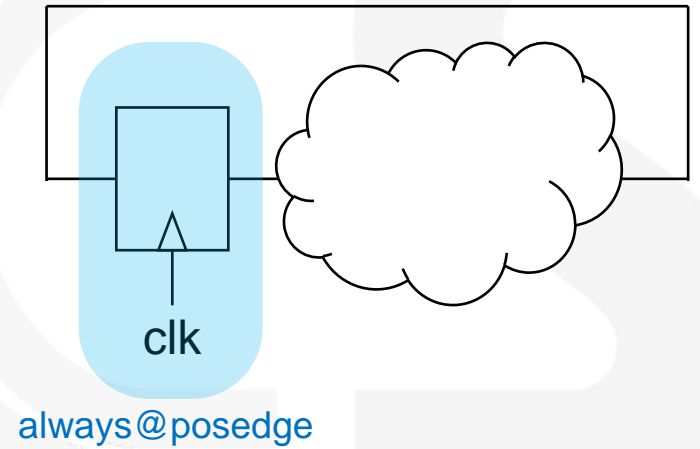


Separating state and next_state



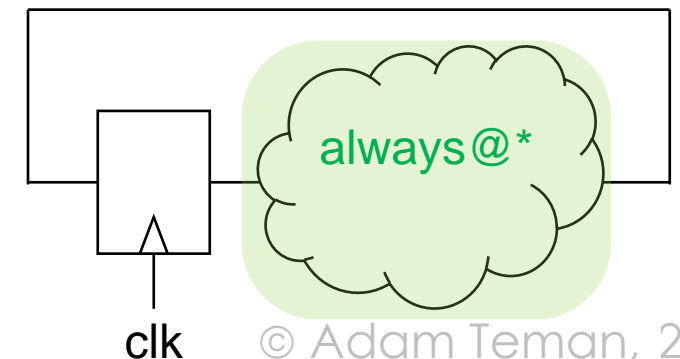
- The **sequential** part of our design is now very clear.
 - It is the set of registers that make up the state of our design.
 - These registers should be written out clearly:

```
always @(posedge clk or negedge rst_n)
  if (!rst_n)
    state <= RESET_STATE;
  else
    state <= next_state;
```



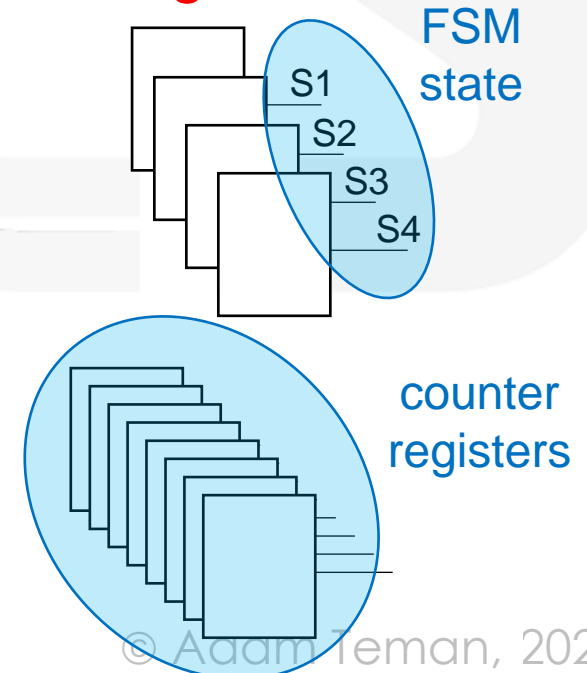
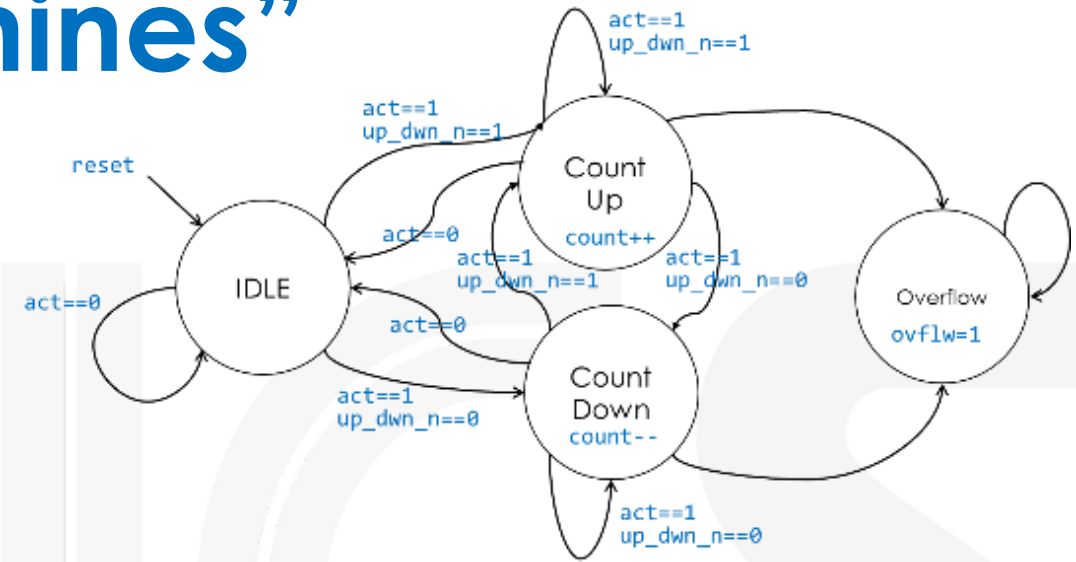
- Everything else calculates the **next_state** and is **combinatorial**

```
always @*
  case (state)
    STATE1: next_state = STATE2;
    ...
    ...
```



Note about “state machines”

- It is important to differentiate between “state” of the design and the states that make up a finite state machine (FSM)
 - The states of an FSM are a good design tool for creating sequential logic.
 - However, the states of the FSM are **not the entire state of the design**.
- For example, looking at the simple counter from the lecture
 - There are **four states** in the state machine
 - This requires **four registers** for one hot encoding
 - But the design also needs registers for storing the **current counter value** (e.g., 8 registers for an 8-bit counter)
 - Therefore, the state of the design includes **four “state” registers** and **eight “count” registers**.



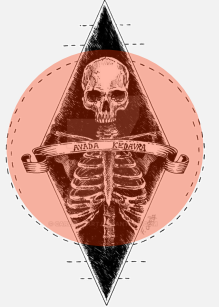
“Fixing” the example from the lecture

- In the example code, I “**cheated**” and put **logic in the count registers’ sequential block**
 - This was really “simple logic”, but *don’t do it* (until you’re a veteran designer)
 - Only write sequential code that **directly maps to a flip flop** from the standard cell library
- I should have written this as such:

```
always @*
begin
    next_count = count;
    if (state==CNTUP)
        next_count <= count+1'b1;
    else if (state==CNTDN)
        next_count <= count-1'b1;
end
assign count_en=(state==(CNTUP|CNTDN)) ? 1'b1 : 1'b0;
```

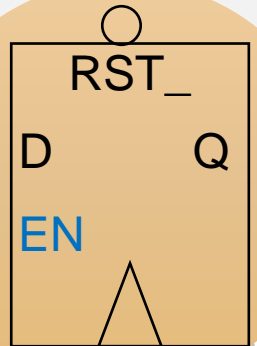
Note that by **initializing next_count**, we don’t need to assign it for every condition

```
always @(posedge clk or negedge rst_n)
if (!rst_n)
    count <='b0;
else
    if (state==CNTUP)
        count <= count+1'b1;
    else if (state==CNTDN)
        count <= count-1'b1;
```



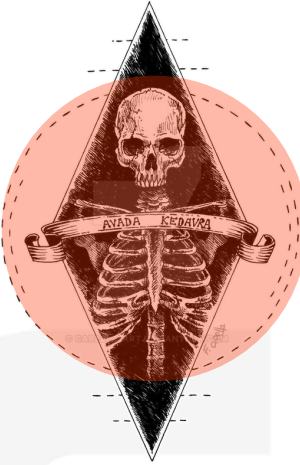
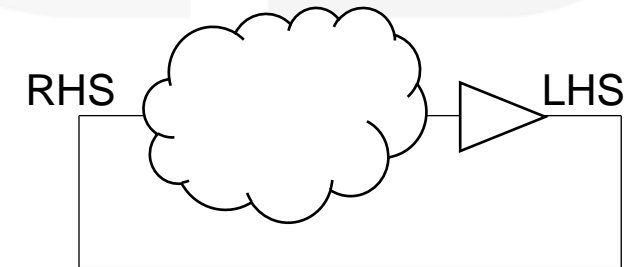
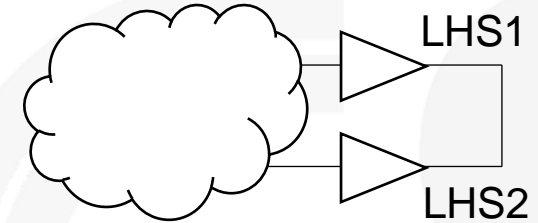
```
always @(posedge clk or negedge rst_n)
if (!rst_n)
    count <='b0;
else if (count_en)
    count <= next_count;
```

Load enable flip flops are part of the standard cell library and you are encouraged to use them.



No multi-driven nets

- **Never assign a signal from two always blocks** (or “assign”s)
 - This results in two logic blocks driving the same net
 - CMOS cannot tolerate multi-driven nets
- **If you follow the guidelines, this is easy to check**
 - Each register should have its own `always@posedge` block
 - Signals can appear in the **LHS** of **only one** `always@*` block
 - Signals can appear in the **RHS** all over the place
- **That being said, watch out for combinatorial loops**
 - A combinatorial signal cannot be assigned (**LHS**) by itself
 - In other words, it **cannot appear on both LHS and RHS** in the same logic path (even upstream several stages)!



Code Verification Checklist

- To summarize, after writing your code, go over this checklist:
 - ❑ Clock and reset signals do not appear on the RHS of any `always@*` or `assigns`
 - ❑ No clock domain crossing occurs.
 - ❑ All combinatorial `ifs` have a corresponding `else`.
 - ❑ All `case` options are covered and/or a `default case` is provided.
 - ❑ All LHS in an `if/else/case` appear are assigned for all conditions.
 - ❑ An LHS in an `assign/always@*` are an LHS only in that block.
 - ❑ All combinational sensitivity lists use the `always@*` notation.
 - ❑ All sequential blocks have (`posedge clk` or `negedge rst_`) notation.
 - ❑ No logic in sequential blocks, i.e., `always@posedge` blocks map directly to a flip flop from the standard cell library
 - ❑ Check for combinatorial loops, i.e., no LHS is affected by itself in RHS

Additional useful tips

- Use **System Verilog logic** type:
 - Instead of screwing around with Verilog's wire and reg types, use SV's logic.
- Use **System Verilog always** blocks:
 - Use `always_comb` instead of `always@*` (also verifies no **multi-driven!**)
 - Use `always_ff` instead of `always@posedge` (also verifies no **accidental latch**)
- Initialize your **LHS** values
 - Provide an **initial value** for each **LHS** in an `always@*` block
 - Always use a **default** case value
- Run **synthesis** or **linter**:
 - Synthesize your design and go over the errors and warnings.
 - Use a linter to make sure your code is well-written.