Lab 6
Introduction to Verilog - 3

Instructor: Chia-Chi, Tsai

Speaker: 李秉軒 William

Outline

1. Syntax for verification
2. Other useful syntax
3. Concept of pipeline
4. Synthesis
5. Reference

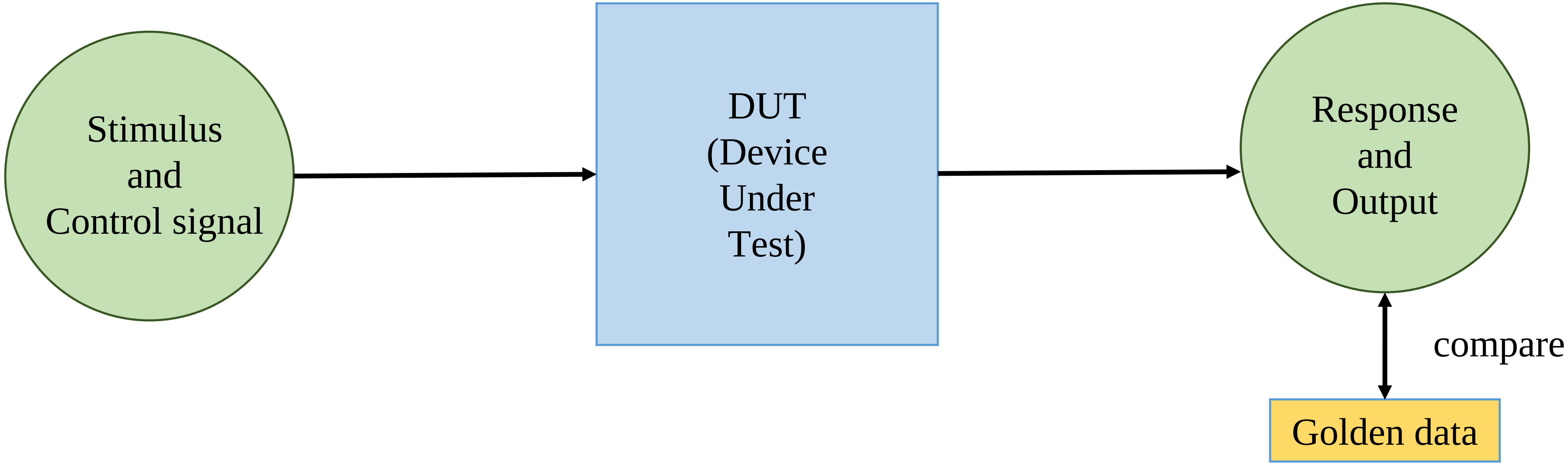
Syntax for verification

Verification vs Testing

- Verification(驗證) : Verify the correctness of your design **before** turning it into real hardware.
- Testing(測試) : Test the hardware whether it works as anticipation or not.

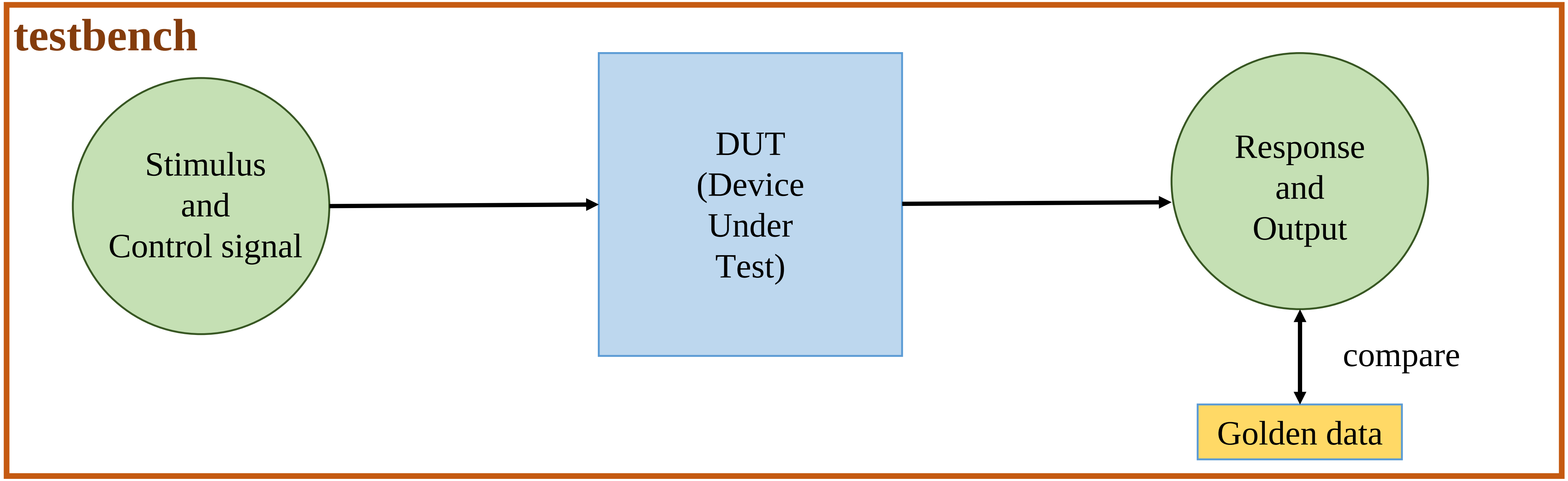
Verification - Simulation

- **Simulation** : Given defined/ random stimulus and control signal, check the correctness of output.



Testbench

- In previous homework, we provide testbench for you to verify your design. In this chapter, we introduce contents in testbench.



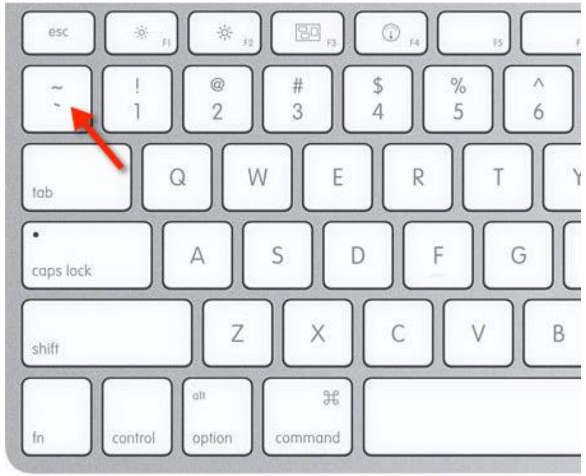
Structure of testbench

```
module testbench ;

    // Data type declaration(Control signal/ Output)

    // Instantiate modules

    // Apply stimulus

   // Verify results

endmodule
```

No ports in testbench

Important syntax in testbench - timescale

- Syntax with ` is called compiler directive and will cause Verilog compiler to do special actions.
- The `timescale compiler directives declares the **time unit** and its **precision.**




```
`timescale 1ns/10ps
//All the time units are in multiples of 1 nanosecond
//You can specify decimal number down to 0.01ns(10ps)
```

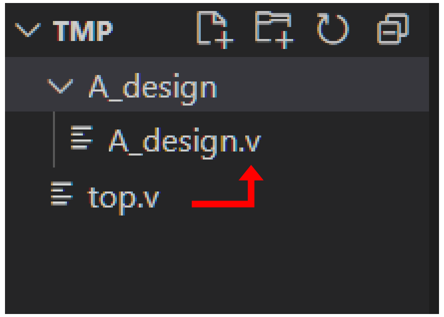
- Simulation speed is greatly affected if there is a large difference between the time units and precision.

Important syntax in testbench - include

- In testbench, we need to include the file which contains our design.
- Use the `include compiler directive to insert the contents of an entire file.

```
`include “top.v” // Make sure “” is not missing.
```

- In ModelSim, relative path is used, but might not be true for the other Verilog simulator.



In top.v

Relative :

`include “A_design/A_design.v”

Absolute :

`include “TMP/ A_design/A_design.v”

Important syntax in testbench - define

- The `define compiler directive provides a simple text-substitution facility.
- Make your code more readable and easier to modify.
- `define <name> <macro text>
- `<name> will substitute <macro text> at compile time.

```
`define s0 1’b0
`define s1 1’b1
always@(posedge clk) begin
    state <= (state)? `s0 : `s1;
end
```

define state

Use of macro

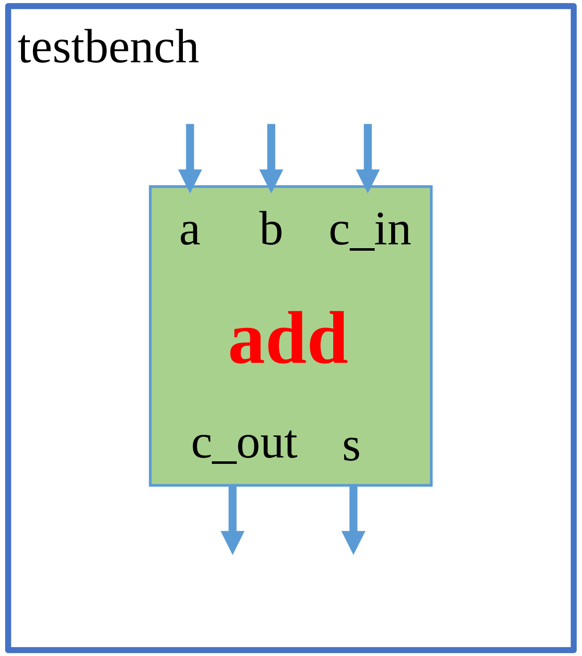
Instantiating module

- We need to instantiate(or declare) our design in testbench. First make sure that you have included the file.
- <module_name> <instance_name> (<port mapping>);
- In tb.v
  - `reg a, b, c_in;`
  - `wire c_out, s;`
  - `Adder add(a, b, c_in, c_out, s);`

Use reg to drive stimulus

Use wire to receive result

- In Adder.v
  - `module Adder(a, b, c_in, c_out, s);`
  - `input a, b, c_in;`
  - `output reg c_out, s;`



Instantiate module – port mapping

- We need to explicitly specify each port connection. And there are 2 ways to do it.

1. Position mapping :
   - `module Adder(a, b, c_in, c_out, s);`
   - `Adder add(a, b, c_in, c_out, s);`

   Must follow same order!

2. **Name mapping(recommended) :**
   - `module Adder(a, b, c_in, c_out, s);`
   - `Adder add( .b(b), .a(a), .s(s),`

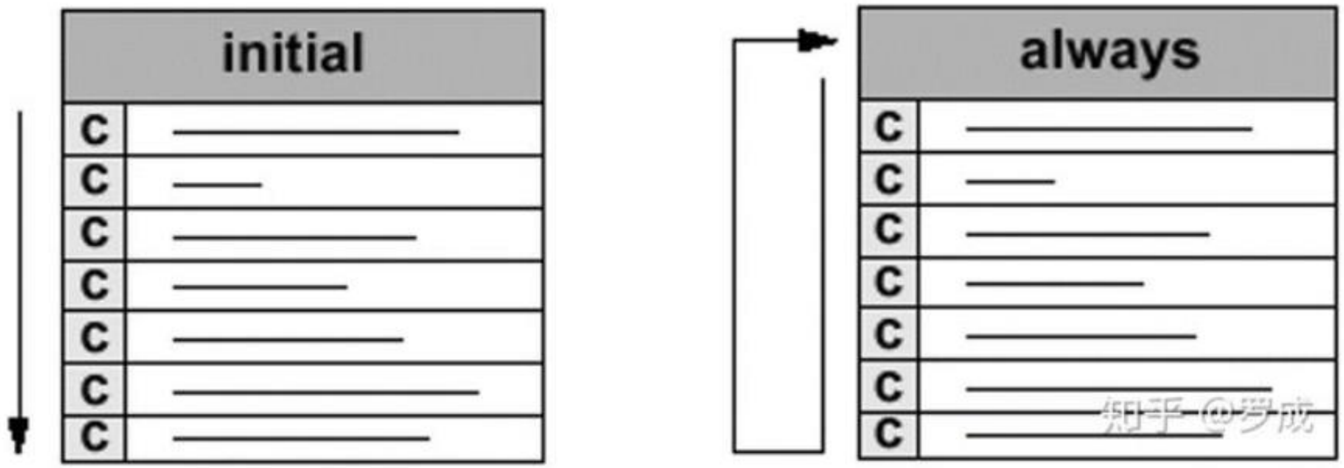
     `.c_in(c_in), .c_out(c_out));`

   Order is not important!

   Switch to next line for cleaner code

Initial block

- In Verilog, we use **always** to drive value, but it will execute infinitely until the simulation ends(like a loop).
- If we want some statements to only happen once, we may use Initial block.



Source : https://zhuanlan.zhihu.com/p/72078544

Delay specification

- In a sequentially executed procedural block, you might need to insert some delay between statements.
- The pound sign (#) character denotes the delay specification for both gate instances and procedural statements.

```
`timescale 10ns/1ns
initial begin
    #5 counter <= counter + 1;
    #5 counter <= counter + 1;
    #5 counter <= counter + 1;
end
```

Counter will increment every 50ns

Example – Apply stimulus

```
initial begin
            reset = 1;
            clock = 0;
        #5  reset = 0;
        #5  input1 = 12;
            input2 = 11;
        #20 //Verify result
end
always #10 clock = ~clock;
```

Active high reset

Clock period = 20 time units

Display Results

- $time is a system function that returns the current simulation time.
- $monitor is a system task that displays the values of the argument list whenever any of the arguments change.
- $display also displays the values, but only execute when the statement is reached.
- Examples:

```
$monitor($time, “out= %d”, out);
$monitor($time, “%b %h %d %o”,sig1, sig2, sig3, sig4);
```

Format spcifier must follow same order

End the simulation

- Once all procedural blocks stops executing, the simulation will stop. However, sometimes you might fall into some infinite loop.
- Use $finish/ $stop system task to force the simulation to end.

```
initial begin
    #10000 $finish;
end
```

Simulation will be force to end after 10000 time units.

```
always #10 clock = ~clock;
```

You'd have to force the simulation to end if you create clock like this.

Other useful syntax

For loop

- Loop in Verilog is not like loop in C/C++. Verilog will unroll the loop and each iteration represents a real hardware. Therefore, you must be very careful when using it.

```
integer i;
for(i=0; i<10; i=i+1)begin
      a[i] = b[i]*c[i];
end
```

Verilog will instantiate 10 multipliers!!

For loop – cond.

- When you are fully aware of what hardware will be instantiate, loop can save you a lot of time

```
reg [7:0] a [31:0];
reg [7:0] b [31:0];
a[0] = b[0];
a[1] = b[1];
      .
      .
a[31] = b[31];
```

=

```
integer i;
for(i=0; i<32; i=i+1)begin
      a[i] = b[i];
end
```

Signed notation

- In Verilog, all signals are default unsigned. It may cause some confusion when it is signed operation.

```
wire [3:0] a = -5;
wire [3:0] b = 3;
always@(*) begin
      if(a>b) $display(“a>b”);
      else    $display(“a<=b”);
end
```

- Result :

```
VSIM 4> run -all
# a>b
```

Signed notation – cond.

- Declare the signal as signed to prevent this error.

```
wire signed [3:0] a = -5;
wire signed [3:0] b = 3;
always@(*) begin
      if(a>b) $display(“a>b”);
      else    $display(“a<=b”);
end
```

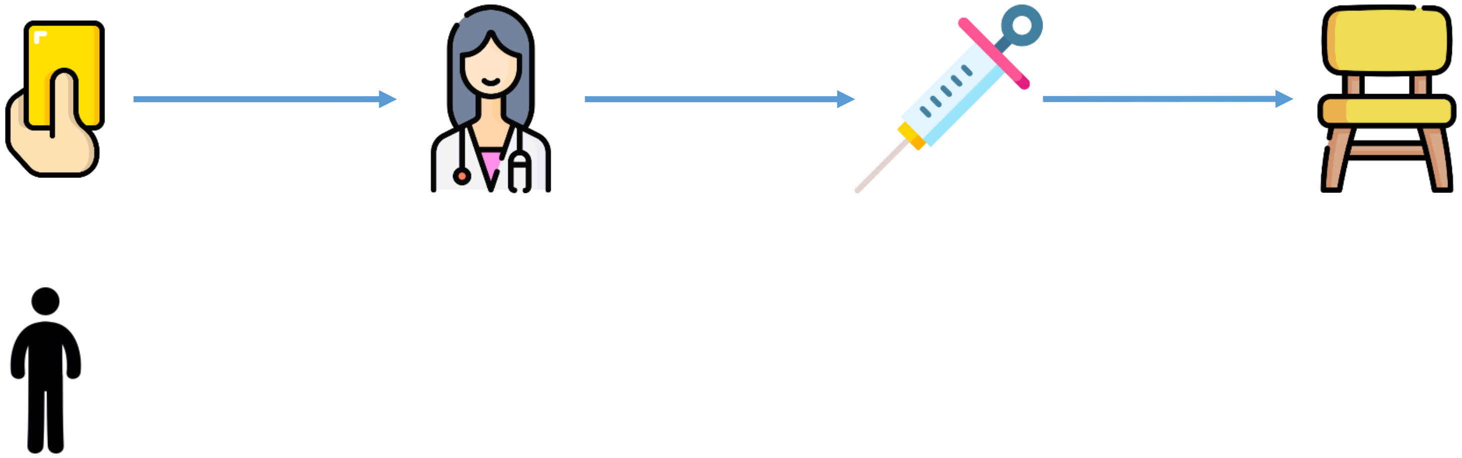
- Result :

```
VSIM 8> run -all
#  a<=b
```

Pipeline

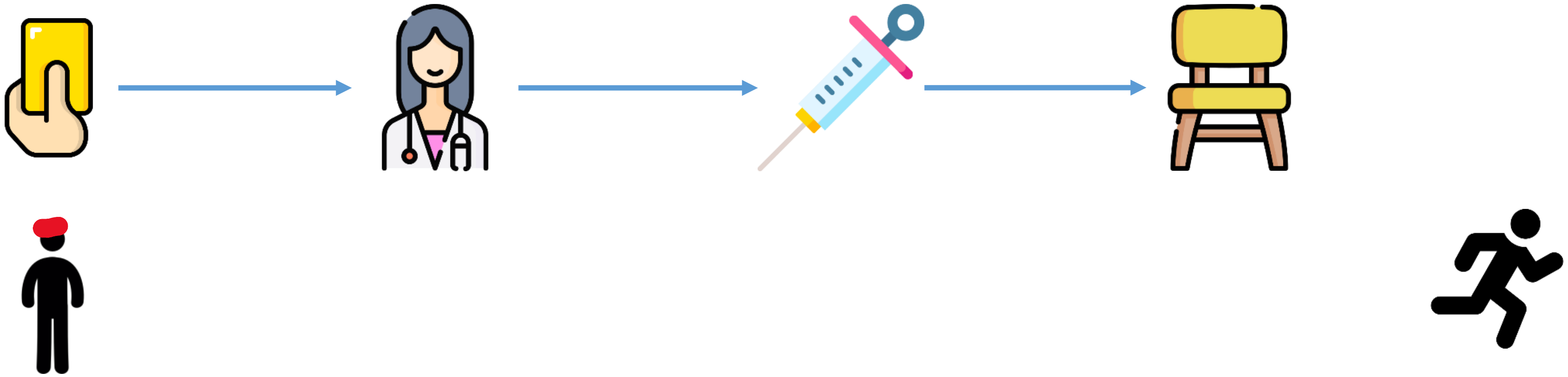
Covid-19 Vaccination : non-pipeline

- There are 4 stages in the vaccination, assume only one person is allowed to enter the hospital at a time.



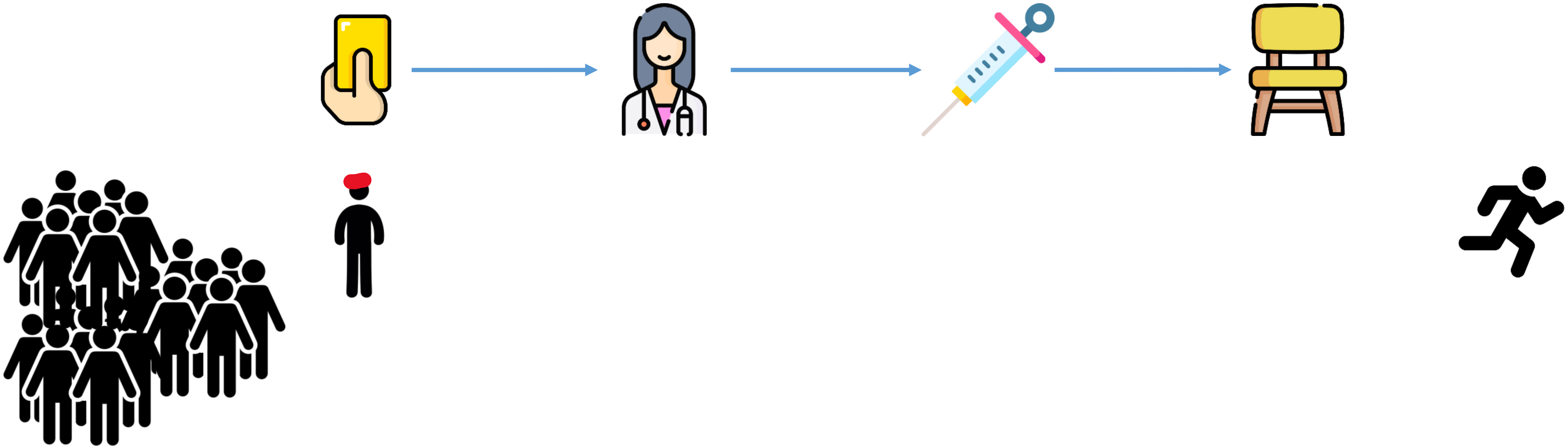
Covid-19 Vaccination : non-pipeline

- Next person is allowed to fill out his/her yellow card when the previous person finishes resting.



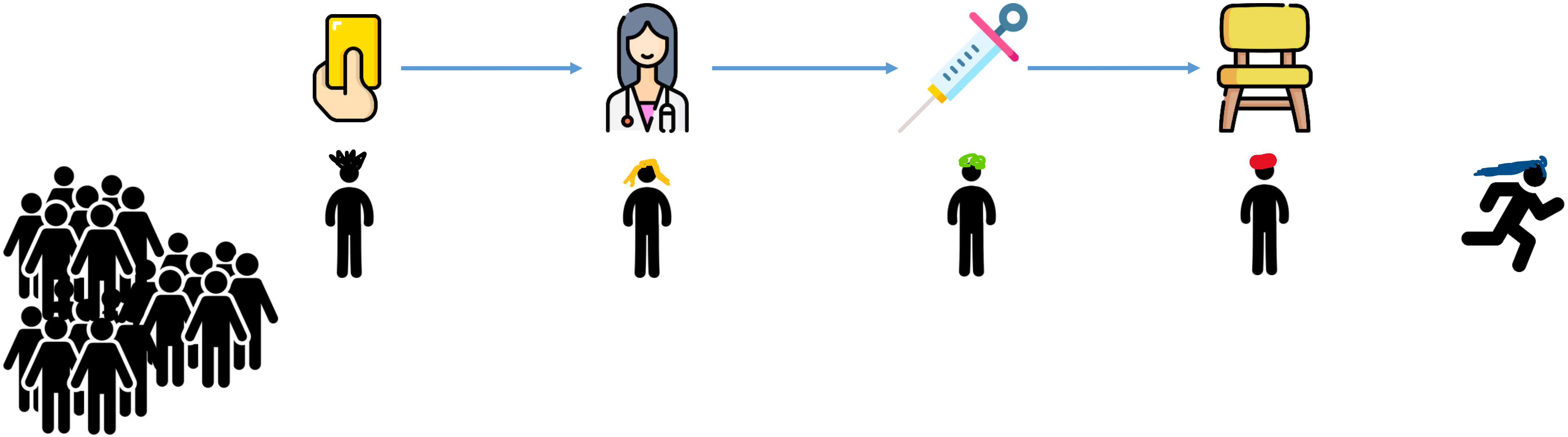
Covid-19 Vaccination : non-pipeline

- There will be people mountain people sea……



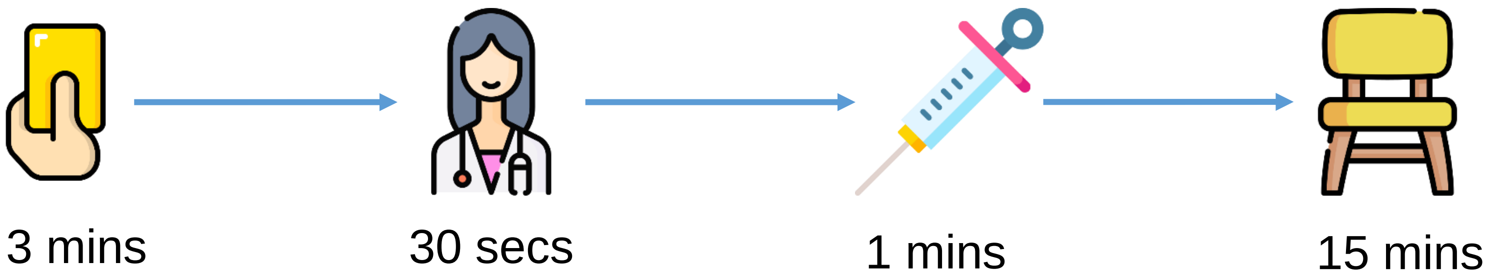
Covid-19 Vaccination : pipeline

- Assume that we allow more than one person entering the hospital.



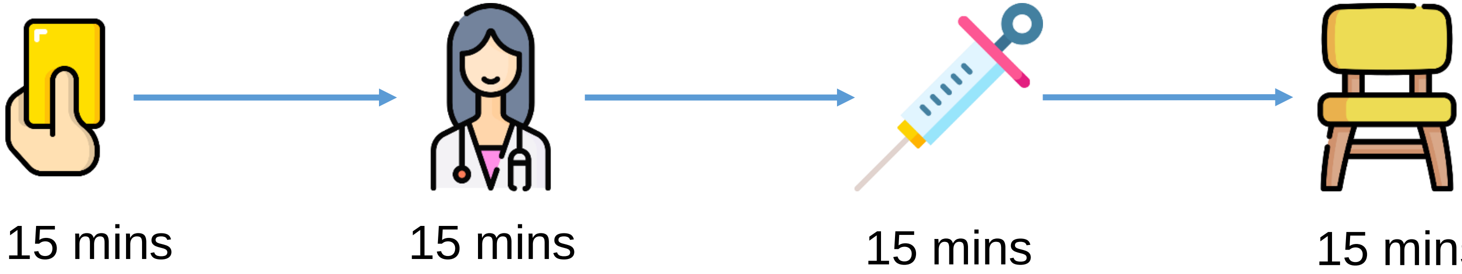
Covid-19 Vaccination : cycles

- Because a stage can only allow one person. Even though a person only takes 3 mins to fill out his/her yellow card, he/she still needs to wait 15 mins to get to the doctor.



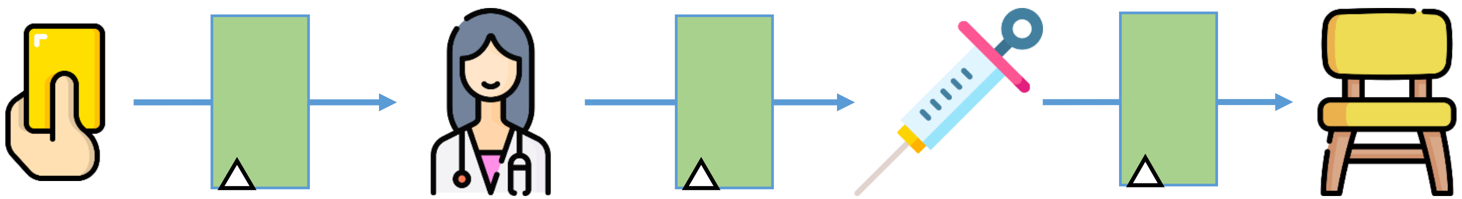
Covid-19 Vaccination : cycles

- Because a stage can only allow one person. Even though a person only takes 3 mins to fill out his/her yellow card, he/she still needs to wait 15 mins to get to the doctor.
- It is like every stages takes 15 mins.



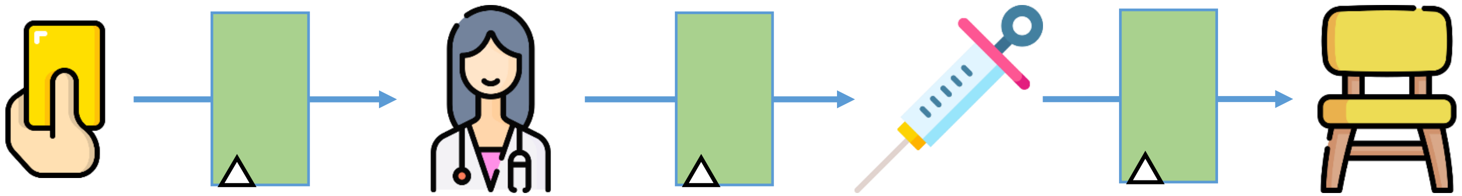
Covid-19 Vaccination : cycles

- In hospital, you might check if the next state is empty by your eyes. In real hardware, we want a more robust solution.
- We use clock to tell each patients to move to next stage. In real hardware, flip flops will receive clock signal and move data to next stage.



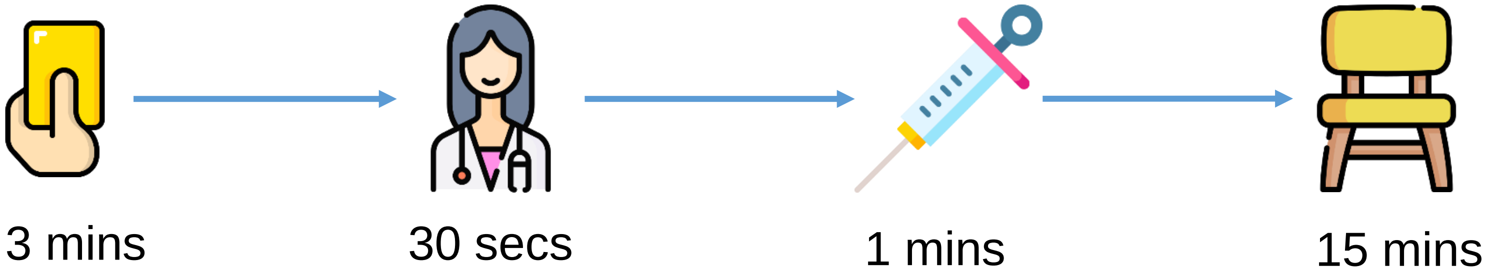
Covid-19 Vaccination : cycles

- In this example, we need the clock period to be 15 mins so that no traffic jam will happen.
- We refer to the longest path that decides the clock period as “Critical path”.



Covid-19 Vaccination : pl vs npl

- Now let’s compare the performance of non-pipeline & pipeline. Assume there are 100 people waiting.

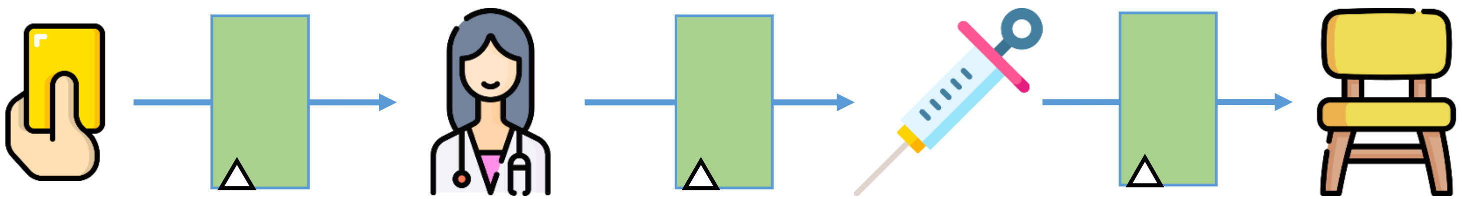


- non-pipeline : (3+0.5+1+15)* 100 = 1950 mins
- pipeline : (15+15+15+15) + 15*99 = 1545 mins

first person After first person, the rest of the people will take 15mins to finish.

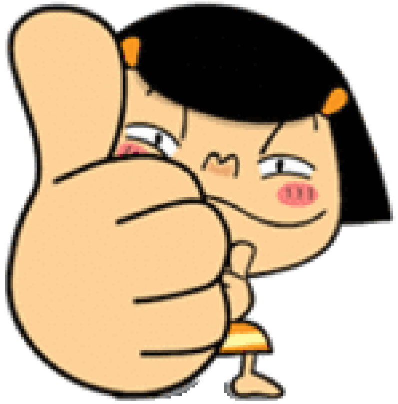
Example : how to pipeline

- We use real Verilog code to illustrate pipeline.



Pipelined CPU

- The reason why pipeline can increase performance is increasing hardware usage.
- The most famous example of pipeline is pipelined CPU, a.k.a your final project.
- Be ready and enjoy it.



Synthesis

After RTL…

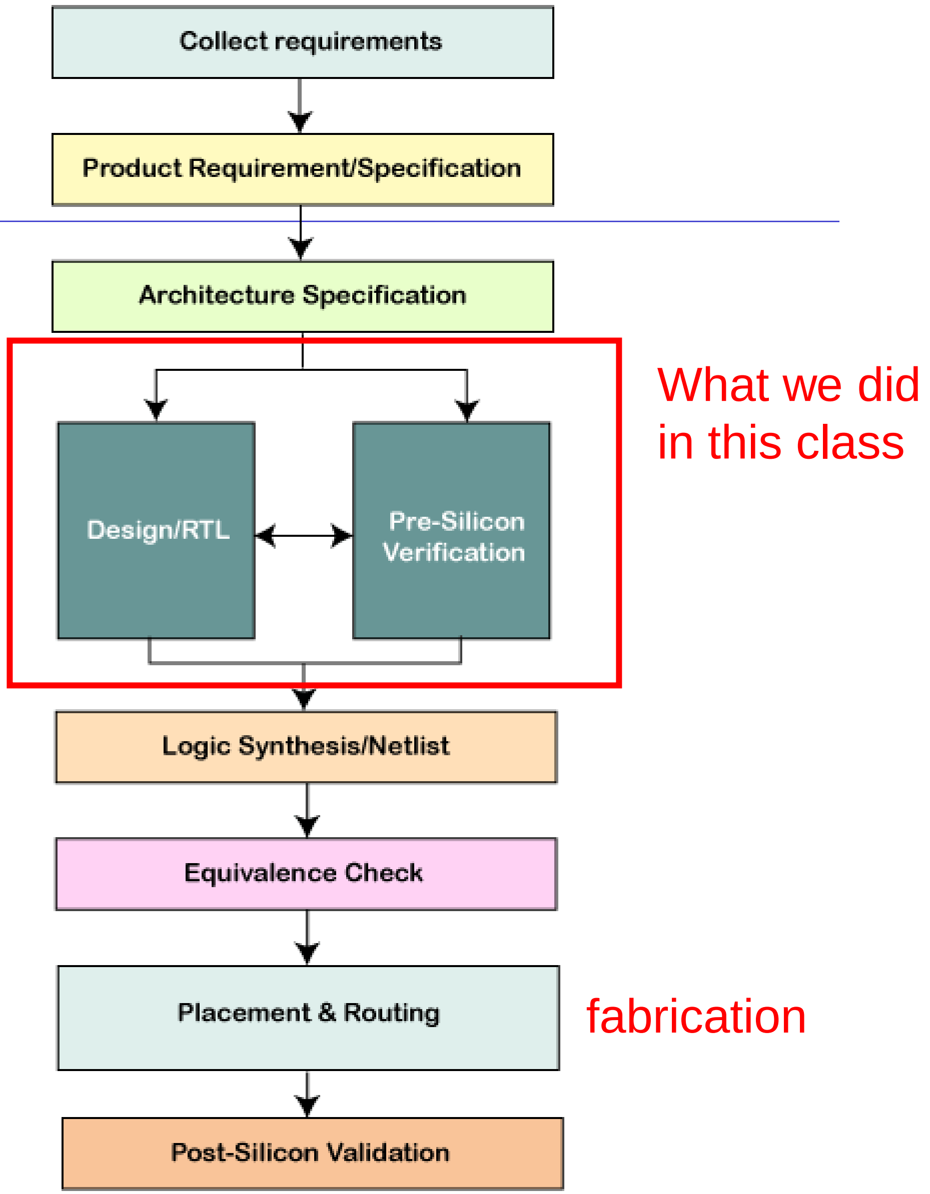
- After you finish your RTL design, there are still some steps before it becomes a real chip.

- Logic Synthesis :

  Convert your RTL design to netlist.

- Placement & Routing :

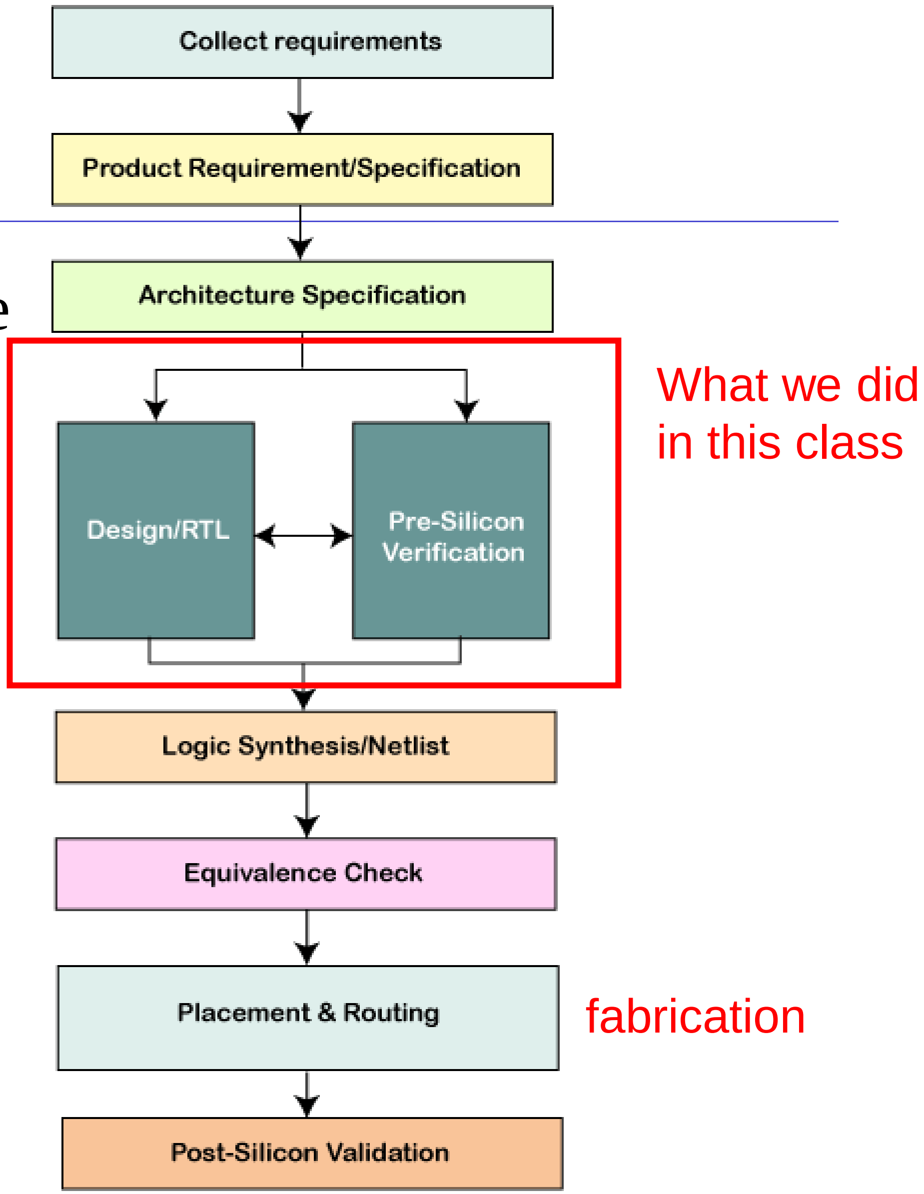
  Put your module and connections on a real chip.



Source : https://www.javatpoint.com/verilog-asic-design-flow

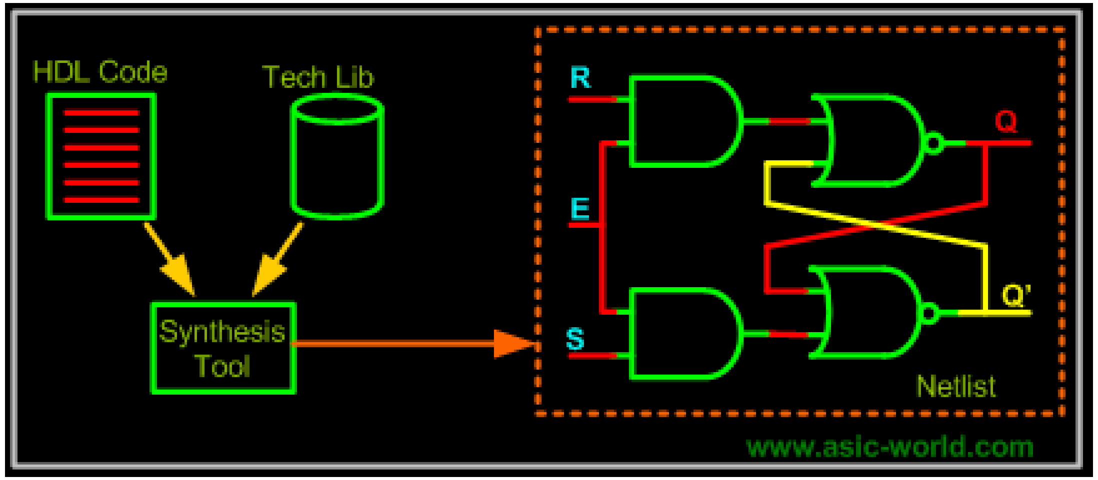
After RTL…

- In this part, we mainly talk about synthesis. But we won’t make you guys do that, no worries ☺



Standard cells

- Standard cells are hardware that your vendor(the people who will manufacture your chip) support. For example, gates, adder, flip-flops… etc.
- Vendor(for example, *TSMC*) will provide a library specifying what standard cells they support.



Source : https://www.asic-world.com/

- You need synthesis tool to convert your design. For example, *design compiler*.
- After synthesis, a circuit composed of only standard cells are created, called netlist.

Synthesis limitations

- In this course, we only consider pre-synthesis design, which is a totally ideal condition. Some Verilog syntaxes or semantics are not synthesizable.
- Un-synthesizable syntaxes introduced in this class :
  - `initial`
  - `delay(#)`
  - `$display, $monitor, $finish,` everything start with $
  - Assignment statements with a variable used as a bit select on the left side of the equal sign.

    ```
    i = 10; a[i] = 1000;
    ```

  - Although variable as bit select is not synthesizable, it is okay if it's in loop. But make sure the number of loops is predefined.

Coding styles for synthesis

- Following semantics are not synthesizable(or synthesizable but should be avoided) :

```
always@(posedge clk or negedge clk) begin
    //...
end
```

triggering at both posedge and negedge

```
always@(posedge clk) begin
    if(b == 15) a <= 15;
end
```

multi-driven nets

```
always@(posedge clk) begin
    if(b != 15) a <= 0;
end
```

```
always@(a) begin
    b = a + 1'b1;
end
always@(b) begin
    a = b + 1'b1;
end
```

Combination loop

Coding styles for synthesis

- Make sure all if statements come with an else statement. All case statements include a default case(or make sure all cases are covered.)
- Make sure you use :
  - Blocking assignment (=) in combinational block.
  - Non-blocking assignment (<=) in sequential block.
- Use continuous assignment (`assign a = b`) outside of any always block.

Verilog

- To become a master in Verilog takes a lot of practice and frustration. Even you really become a Verilog master, the knowledge of VLSI system and Digital IC design is what really matters. Verilog is simply a tool.

- Following the coding guideline may be painful, but it will save you a lot of time and effort in the future.
- Good luck.

Reference

- IEEE Standard 1364-2005

Thanks for listening