Session #6

Sequential Logic

Sequential blocks of logic description in Verilog require procedural block always. The two challenges met while describing sequential circuits are:

- Selection of inputs for the sensitivity list
- Correct layout of the statements inside

Use **if** statement in the main part of the block.

Handle asynchronous inputs **before** anything else
(in **first** branches of of the conditional statement)

Handle synchronous inputs in the branch of **if** statement **after** the last asynchronous input branch

Synchronous input checking may be done any convenient way (e.g. using **case** instead of **if**)

Basic sequential logic elements are: flip-flops and latches

Latches

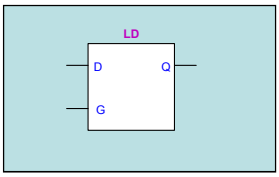
Latches are memory elements that are **level-sensitive**.

They typically consist of **three ports**:

an **input**,

an **enable**

an **output** port.



The enable signal latches or holds the value of the input. It may be active either high or low. The truth–tables for latches are shown in the tables below:

D	G	Q
D	0	Q’
D	1	D

active high

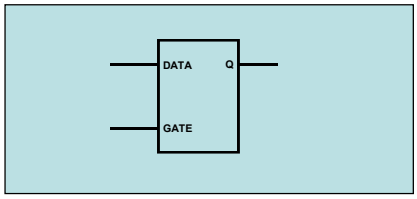
D	G	Q
D	0	D
D	1	Q’

active low

Latches

Simple D Latches are always inferred from incomplete conditional statements.

Designers must be careful while using conditional statements inside **always** blocks: sometimes latches can be inferred when it was not the designer’s intention.

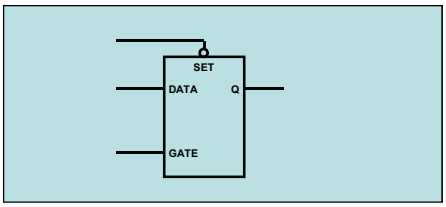



```
module d_latch (GATE, DATA, Q);

  input GATE, DATA;
  output Q;
  reg Q;

  always @(GATE or DATA)
    if (GATE)
      Q = DATA;

endmodule
```

Latches

D Latch with asynchronous SET can be inferred from a slightly modified simple D latch template.

Designers must be aware that if the target technology does not contain latches with an asynchronous set, a plain latch with additional logic will be synthesized.



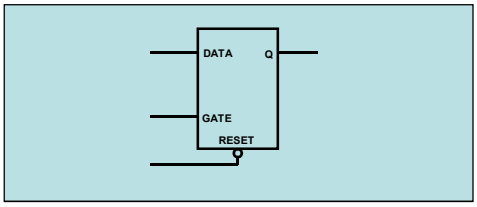

```
module latch_syncset (GATE, DATA, SET, Q);

  input GATE, DATA, SET;
  output Q;
  reg Q;

  always @(GATE or DATA or SET)
    if (~SET)
      Q = 1'b1;
    else if (GATE)
      Q = DATA;

endmodule
```

Latches

D Latch with asynchronous RESET can be inferred from the slightly modified simple D latch template.

Designers must be aware that if the target technology does not contain latches with an asynchronous reset, a plain latch with additional logic will be synthesized.

Remember this as a template for the modeling latches.




```
module latch_arst(RESET, GATE, DATA, Q);

  input RESET, GATE, DATA;
  output Q;
  reg Q;

  always @(RESET or GATE or DATA)
    if (~RESET)
      Q = 1'b0;
    else if (GATE)
      Q = DATA;

endmodule
```

Procedural Assignments

There are two types of procedural assignments

Blocking assignment: **For modeling combinational blocks**

Shall be executed before the execution of the statement that follows it in sequential block.

Does not block the execution of statement that follows it in a parallel block

Non-blocking assignment: **For modeling sequential blocks**

Allows assignment ***scheduling*** without blocking the procedural flow.

Can be used whenever several register assignments within the same time step can be made without regard to order or dependence upon each other.

Procedural Assignments

Blocking assignments:

```
register = 0;
register [4]= 1'b1;   // a bit-select
register [4:0]= 5'b01011; // a part-select
mem [4]= 8'h0F;       // assignment to a memory element
{reg_1,reg_2} = register; // a concatenation
```

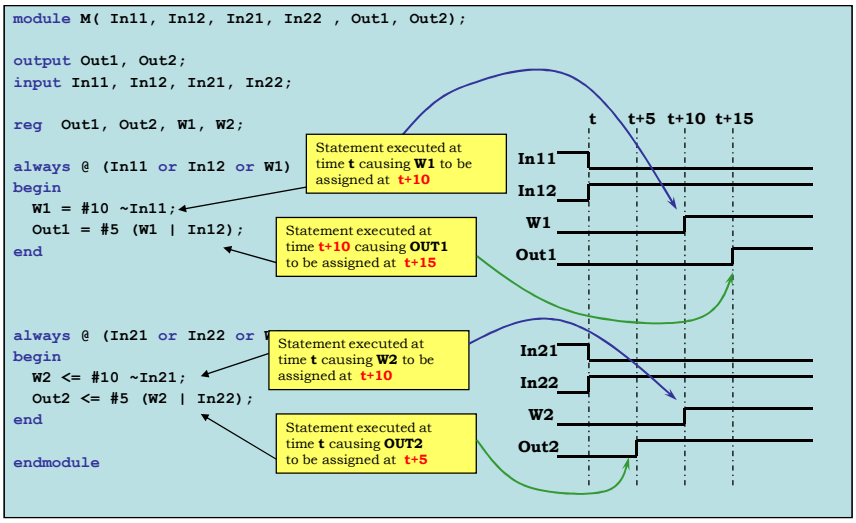
Non-blocking assignments:

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

= acts as blocking operator when used inside always block. Not to be confused with the continuous assignment done with assign.

It is always recommended to use = operator inside initial

Illustration



Blocking vs. non-blocking

Blocking assignments

1.

```
begin
    a = #20 c;        <-- t=20
    b = #30 d;        <-- t=50
end
// a = c at time 20;
// b = d at time 50;
```

2.

```
// a =50;
begin
    a = #20 20;
    b = #10 a+100;
end
// b =120;
```

Non-blocking assignments

1.

```
begin
    a <= #20 c;       <-- t=0
    b <= #30 d;       <-- t=0
end
// a = c at time 20;
// b = d at time 30;
```

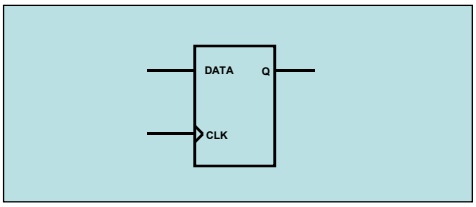
2.

```
// a =50;
begin
    a <= #20 20;
    b <= #10 a+100;
end
// b = 150;
```

Use blocking statements to model combinational always blocks and non-blocking to sequential always blocks.

Flip Flops

Simple D Flip-flop can be inferred from the **always** block sensitive to the edge of clock signal.
posedge CLK should be used to make a block sensitive to the rising edge of CLK.
negedge CLK should be used to make a block sensitive to the falling edge of CLK.

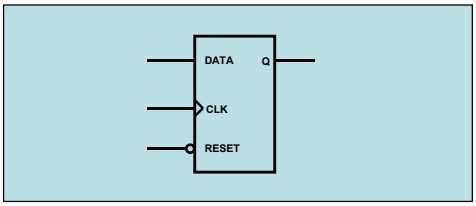



```
module dff (DATA, CLK, Q);

  input DATA, CLK;
  output Q;
  reg Q;

  always @(posedge CLK)
    Q <= DATA;

endmodule
```

Flip Flops

D Flip-flop with synchronous RESET can be inferred from the **always** block sensitive to the edge of the clock signal.

Please note that RESET input is not used in the sensitivity list of the **always** block, although it is checked inside the block.



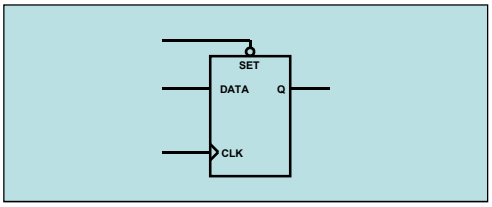

```
module dff_srst (DATA, CLK, RESET, Q);
  input DATA, CLK, RESET;
  output Q;
  reg Q;

  always @(posedge CLK)
    if (~RESET)
      Q <= 1'b0;
    else
      Q <= DATA;
endmodule
```

Flip Flops

D Flip-flop with asynchronous SET can be inferred from the **always** block sensitive to the edge of the clock signal and the set signal.

If reverse polarity of SET is required, both the sensitivity list of the always block and **if** condition inside must be modified.

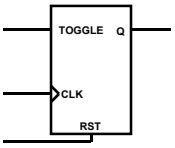
Depending on the resources available in the target architecture, the synthesizer may infer inverters in the SET signal path.



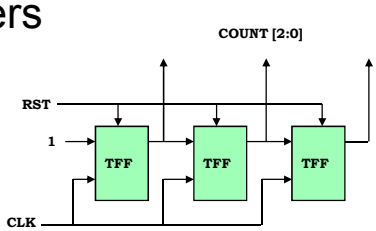

```
module dff_aset (DATA, CLK, SET, Q);

  input DATA, CLK, SET;
  output Q;
  reg Q;

  always @(posedge CLK or negedge SET)
    if (~SET)
      Q <= 1'b1;
    else
      Q <= DATA;

endmodule
```

Registers

How do you model a counter?



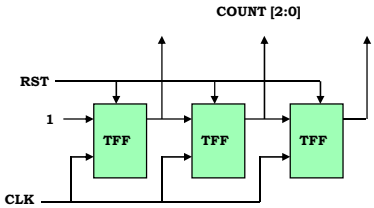
First model a TFF and then write the counter module by instantiating 3 TFFs to get a 3 – bit counter




```
module tff (RST, T, CLK, Q);

  input RST, T, CLK;
  output Q;
  reg Q;

  always @ (posedge CLK or posedge RST)
    if (RESET)
      Q = 0;
    else if (T)
      Q = ~Q;

endmodule
```

```
module CNTR (RST, CLK, Q);

  input RST,CLK;
  output [2:0] COUNT;

  tff U1 ( .RST(RST), .T(1'b1),
           .CLK(CLK), .Q(COUNT[2]) );
  tff U2 ( .RST(RST), .T(COUNT[2]),
           .CLK(CLK), .Q(COUNT[1]) );
  tff U3 ( .RST(RST), .T(COUNT[1]),
           .CLK(CLK), .Q(COUNT[0]) );

endmodule
```

Another way to model a counter



Write each TFF model using `always` block.

```
reg Q2;
always @ (posedge CLK )
    if (RESET)
      Q2 <= 0;
    else if (1)
      Q2 <= ~Q2;
```

```
reg Q1
always @ (posedge CLK )
    if (RESET)
      Q1 <= 0;
    else if (Q2)
      Q1 <= ~Q1;
```

```
reg Q0;
always @ (posedge CLK )
    if (RESET)
      Q0 = 0;
    else if (Q1)
      Q0 = ~Q0;
```

Put all these **`always`** blocks in a single Verilog Module

```
module CNTR (RST, CLK, Q);

  input RST,CLK;
  output [2:0] COUNT;
  reg Q2, Q1, Q0;

always @ (posedge CLK )
    if (RESET)
      Q2 <= 0;
    else if (1)
      Q2 <= ~Q2;
always @ (posedge CLK )
    if (RESET)
      Q1 <= 0;
    else if (Q2)
      Q1 <= ~Q1;
always @ (posedge CLK )
    if (RESET)
      Q0 <= 0;
    else if (Q1)
      Q0 <= ~Q0;

assign COUNT[2] = Q2;
assign COUNT[1] = Q1;
assign COUNT[0] = Q0;

endmodule
```

What can be learned from this?

Any given digital circuit can be broken down in to simpler parts of meaningful functionality and their behavior can be expressed in terms of separate always blocks.

These always blocks can be connected as per the circuit via the signal names.

In this case of a counter, observe the communication of always blocks in which TFFs are described as in the counter.

But, is there a much better way to describe the functionality of a counter?

Is there better way to describe a counter?

Look at the counter from its top level, i.e. from the port level. How does it behave?

This is at a higher level than going to the flip flop level details.

Its behavior can be described as follows:

If RST is HIGH, then counter reset to zero. Otherwise, counter count from 0 to maximum value and again comes back to 0 as long as CLK is running. The count value reflects at the COUNT [2:0].

```
module COUNTER (RST, CLK, COUNT);

  input RST,CLK;
  output [2:0] COUNT;
  reg [2:0] TEMP;

  always @ (posedge CLK )
    if (RST)
      TEMP <= 3'd0;
    else
      TEMP <= TEMP + 3'd1;

  assign COUNT = TEMP;

endmodule
```

Register is a set of flip flops described in Verilog as an array of **reg** type.

Each element is a flip flop if it is processed in a clocked **always** blocks (with **posedge** or **negedge** events). Hence, each element can be independently accessed as a flip flop output.

What can be learnt from this?

We can describe the top level behavior of the given digital circuit using Verilog and allow the synthesizer to infer the logical elements needed to construct the circuit.

Case Study: Shift Register

Specifications:

1. If RST = 1, all bits of the 8-bit shift register should become zero.
2. When the CLK is running, the shift register should shift left at the negative edge of CLK when SHDIR = 0.
3. When the CLK is running, the shift register should shift right at the negative edge of CLK when SHDIR = 1.

Exercise 1

- Flip flop with synchronous set and asynchronous reset.
- Flip flop with asynchronous set and asynchronous reset
- Write test benches for the above to verify the model's functionality

30 Min

Exercise 2

- Model a universal shift register using the following specs.

```
Type: Universal shift register
Width: 16
Shifts right/left (right when SHDIR input set high)
Clock input CLK active high
Clock enable input CE active high
Synchronous Clear input CLR active high
Synchronous Set input SET active high
Load input LOAD active high
Serial input SI
```

Write a test bench to verify the above functional model

30 Min

Exercise 3

- Design, implement and verify the clock divider circuit which can divide the input clock by 3
- What steps should be taken to make the output clock to have 50% duty cycle if it does not have 50% duty cycle? Prove your solution through simulation.

45 Min

Exercise 4

- Model a counter with the following specs.

```
Details:
       Count direction: up/down
       Clock input CLK active high
       Clock enable input CE active high
       Synchronous clear input CLR active high
       Synchronous set input SET active high
       Load input LOAD active high
       Output enable input OE active high
       Terminal count output TC active high
```

Write a test bench to verify the above functional model

30 Min

Exercise 5

- Model a barrel shifter in Verilog and verify the same using a testbench.

30 Min

Exercise 5

ALU Design

- In computing, an arithmetic and logic unit is a digital circuit that performs arithmetic and logic operations. The ALU is a fundamental building block of a central processing unit of a computer.
- Mathemetician John von Neumann proposed the ALU concept in 1945, when he wrote a report on the foundations of a new computer called EDVAC.
- In this drill, we will design a Arithmetic and Logic Unit for the given specifications and model it in Verilog.
- Refer to the following slides to extract the details.

90 Min

ALU Details

- Type: Arithmetic Logic Unit
- Operations performed: arithmetic and logic
- Data width: 8
- All outputs are registered with respect to CLK signal
- Clock input CLK is edge sensitive, active high
- Asynchronous Clear input CLR is active high
- Clock enable input CE is active high
- Mode input M enables arithmetic operations when set high
- Instruction input
- Carry In input CI active high
- Carry Out CO active high
- Overrflow output OV active high

Identify the inputs and outputs from the above details and draw the block diagram.

ALU Design Specifications

- The ALU should perform the following functions:

Arithmetic Functions:

ADD_M: ADD inputs A and B
SUB_M: SUBTRACT inputs A and B
ADD_CIN: ADD inputs A, B and carry in
SUB_CIN: SUBTRACT inputs B nd CIN from A
INC_A: Increment inputs A by 1
DEC_A: Decrement input A by 1
INC_B: Increment inputs B by 1
DEC_B: Decrement input B by 1

Instructions

Logical Functions:

AND: AND function on inputs A and B
NAND : NAND function on inputs A and B
OR: OR function on inputs A and B
NOR : NOR function on inputs A and B
XOR : XOR function on inputs A and B
XNOR : XNOR function on inputs A and B
NOT_A : NOT function on input A
NOT_B : NOT function on input B

Exercise 6

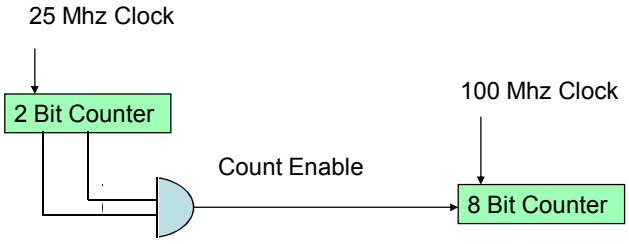
- Design and model an alarm clock with the following specification. Assume appropriate details.
  - Maintains time from 00:00 to 23:59
  - A new time can be loaded
  - Alarm on a preset time

Write a test bench to verify the above functional model

120 Min

Exercise 7

Model the following and resolve if there are any problems. The given counter should count only once every time the circuit A gives out a pulse.

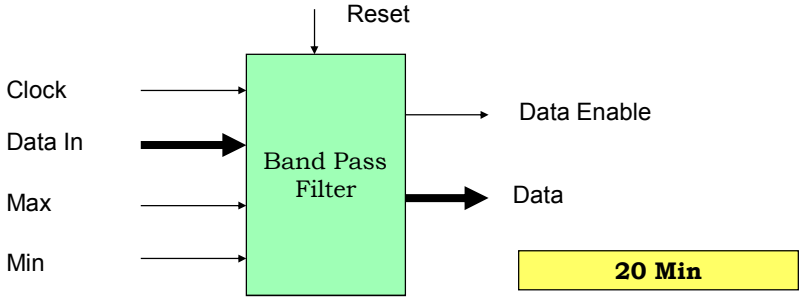


Write a test bench to verify the above functional model

30 Min

Exercise 8

- Model a band pass filter for an incoming data stream.
  - The band pass filter has the upper and lower values as limits.
  - Any value lying in between these values will get a data enable HIGH output, other wise data enable will be LOW.



20 Min

Exercise 9

- Design and implement a 8 bit adder cum subtractor with registered inputs and output.

Comment on why the inputs and outputs are to be registered.

20 Min

Exercise 10

- Design a pipelined data flow architecture for 3 bit multiplier and implement it in Verilog.
- Write a testbench to analyze the same.

What are the implications of not implementing a pipelined architecture for this design?

30 Min