Table of Contents

Table of Figures

Table of Tables

1 Digital design

1.1 Questions with answers

1.1.1 Given only two xor gates one must function as buffer and another as inverter?

Tie one of xor gates input to 1 it will act as inverter.

Tie one of xor gates input to 0 it will act as buffer.

1.1.2 Can u tell me the differences between latches & flipflops?

There are 2 types of circuits:

1. Combinational

2. Sequential

Latches and flipflops both come under the category of "sequential circuits", whose output depends not only on the current inputs, but also on previous inputs and outputs.

Difference: Latches are level-sensitive, whereas, FF are edge sensitive. By edge sensitive, I mean O/p changes only when there is a clock transition.(from 1 to 0, or from 0 to 1)

Example: In a flipflop, inputs have arrived on the input lines at time= 2 seconds. But, output won't change immediately. At time = 3 seconds, clock transition takes place. After that, O/P will change.

Flip-flops are of 2 types:

1.Positive edge triggered

2. negative edge triggered

Latches are also 2 types

Negative latch (transparent when CLK= 0)

Positive latch (transparent when CLK= 1)

1)fllipflops take twice the nymber of gates as latches

2) so automatically delay is more for flipflops

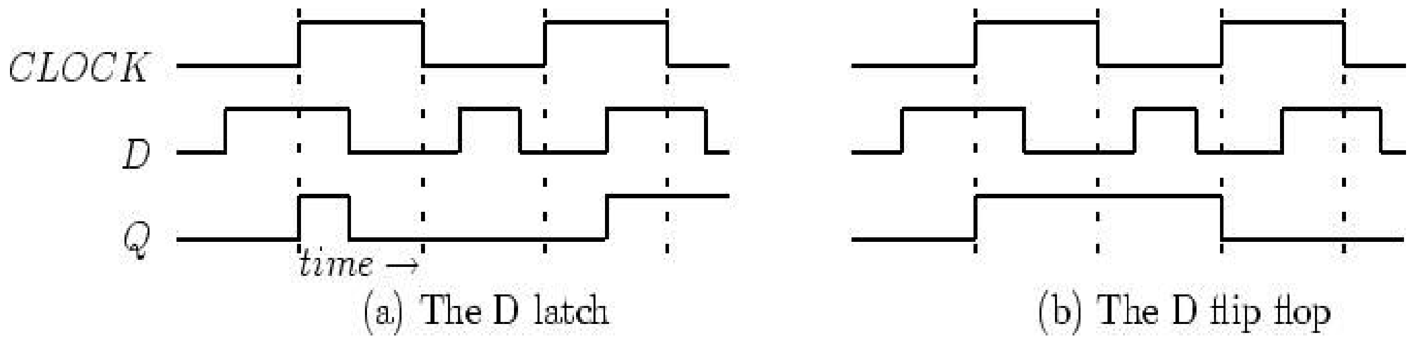
3)power consumption is also more

Latch (level-sensitive, transparent)

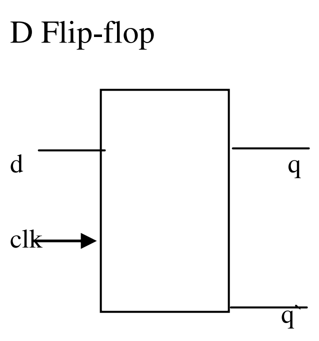
- ❖ When the clock is high it passes input value to Output
- ❖ When the clock is low, it holds value that input had when the clock fell

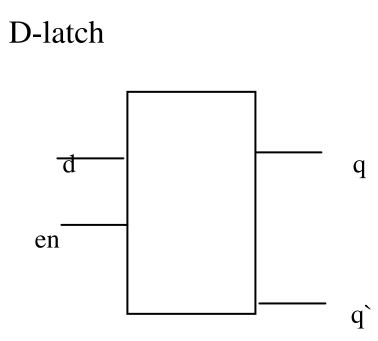
Flip-Flop (edge-triggered, non transparent)

- ❖ On the rising edge of clock (pos-edge trig), it transfers the value of input to output It holds the value at all other times.



In the other way





The example shows D-latch and D-FF.

The simplest form of data storage is latch.It's output responds immediately to changes at the input and the input state will be remembered, or "latched" onto. While "enable" input is active the input of the latch is transparant to the output, once "enable" is disactivated the output remains locked.

Flip flops use clock as a control input. The transition in output Q occurs only at the edge of the clock pulse. Input data must present T_setup time beforeclock edge and remain T_hold time after.

* RESET input, while it is not shown, is present in most FF.

```
module DFF (Q,_Q,D,clk,rst);
output Q,_Q;
input D,clk,rst;
reg Q,_Q;
```

```
always @(posedge clk or posedge rst)
 begin
  if (rst) Q  <= 0;
  else     Q  <= D;
  _Q <= !Q;
 end

endmodule

module DLatch (Q,_Q,D,en,rst);
output Q,_Q;
input D,en,rst;
reg Q,_Q;

always @(en or D or posedge rst)
 begin
  if (rst)     Q  <= 0;
  else if (en) Q  <= D;
  _Q <= !Q;
 end
endmodule
```

Both latches and FFs have their relative advantages and disadvantages in their implications, as summarized in the table below:

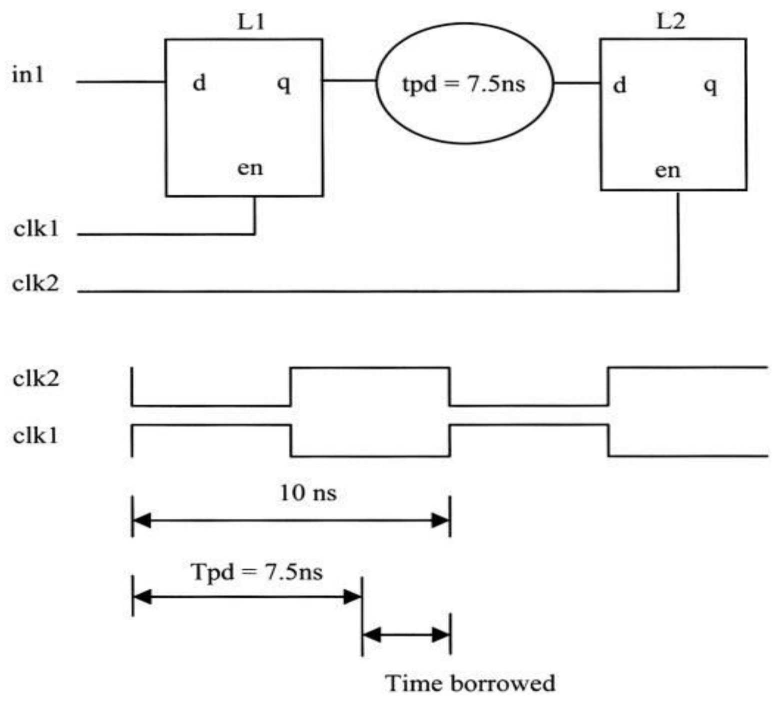
Latch	Flip-flop
Area of a latch is typically less than that of a Flip-flop.	Area of a Flip-flop for same features in more than that of a latch.
Consumes less power, due to lesser switching activity and lesser area.	Power consumption is typically higher,due to the area and free running clock. Additional control required to save power.
Facilitates the time borrowing or cycle stealing. Helps increase pipeline depth with lesser area. Even if the path is longer than a clock cycle for a latch based pipeline, it is okay as long as it meets	Since the clock boundaries are rigid, the facility of time barrowing or cycle stealing does'nt exists with FF's. A negative slack cannot be propagated to the timeing of the next stage pipeline and

the next latch setup margin.	hence must execute within a clock cycle.
In multiple clock schemes, the clock edges must not be overlapping. It makes the logic design, vector generation for verification and clock tree synthesis difficult.	Clock tree synthesis is less tedious in FF based designs. Since the stimulus needs to be stable before the setup time of the clock, the vector genration is relatively easier.
With barrowing* and cycle stealing , the operating frequency is higher than the slower logic path.	Due to rigid timing boundaries, the slowest path pretty much decides the operating frequency.
Makes time buggeting and characterizing the interfaces tedious.	The time budgeting is clearer and characterizing the interfaces is easier.

(*) Time borrowing is a mechanism in which a latch based design takes advantage of the transparency between two back to back latches that are enabled in order to meet the propagation delay between the two latches. This is best illustrated by a simple analysis as follows:

Consider two latches L1 and L2. While both of them have the same clock frequency, the enables for L1 and L2 are opposite in polarity. The L1 is enabled in the high phase of the clock, while L2 is enabled in the low

phase of the clock. This connection is shown in the following figure:



For the purpose of simplifying the analysis, the delay or delay of L1 and L2 is assumed to be 0ns. The propagation delay of the combinatorial logic is 7.5ns. If it were a flip-flop based design with the same

Rising edge clock in place of c1k1 and clk2, this would be clearly a setup violation. However, in a latch based design above, since the delay through latch is 0ns, the input in1 is latched immediately at the output of L1, and begins to propagate. The propagation delay enters into the ON time of the second latch L2, and settles at some point during its ON time. The propagation delay has caused the logic to borrow time from the second latch, in order to settle its outputs, and hence is called time borrowing.

1.1.3 What is the difference between latches and flip-flops based designs

Latches are level sensitive and flip-flops are edge sensitive. Latch based design and flop based design is that latch allows time borrowing which a tradition flop does not. That makes latch based design more efficient. But at the same time, latch based design is more complicated and has more issues in min timing (races). Its STA with time borrowing in deep pipelining can be quite complex.

1.1.4 What is meant by inferring latches, how to avoid it?

Consider the following :

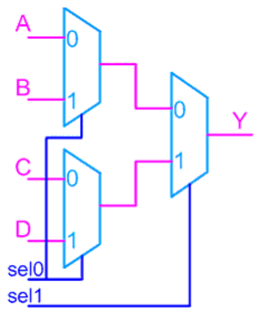
```
always @(s1 or s0 or i0 or i1 or i2 or i3)
case ({s1, s0})
2'd0 : out = i0;
2'd1 : out = i1;
2'd2 : out = i2;
endcase
```

in a case statement if all the possible combinations are not compared and default is also not specified like in example above a latch will be inferred ,a latch is inferred because to reproduce the previous value when unknown branch is specified.

For example in above case if {s1,s0}=3 , the previous stored value is reproduced for this storing a latch is inferred. The same may be observed in IF statement in case an ELSE IF is not specified.

To avoid inferring latches make sure that all the cases are mentioned if not default condition is provided.

1.1.5 Build a 4:1 MUX using only 2:1 MUX?



1.1.6 Difference between mealy and Moore state machine?

A) Mealy and Moore models are the basic models of state machines. A state machine, which uses only Entry Actions, so that its output depends on the state, is called a Moore model. A state machine, which uses only Input Actions, so that the output depends on the state and also on inputs, is called a Mealy model. The models selected will influence a design but there are no general indications as to which model is better. Choice of a model depends on the application, execution means (for instance, hardware systems are usually best realized as Moore models) and personal preferences of a designer or programmer

B) Mealy machine has outputs that depend on the state and input (thus, the FSM has the output written on edges)

Moore machine has outputs that depend on state only (thus, the FSM has the output written in the state itself.

Adv and Disadv

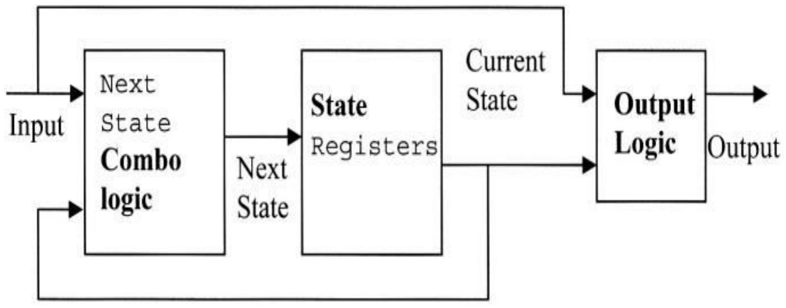
In Mealy as the output variable is a function both input and state, changes of state of the state variables will be delayed with respect to changes of signal level in the input variables, there are possibilities of glitches appearing in the output variables. Moore overcomes glitches as output dependent on only states and not the input signal level.

All of the concepts can be applied to Moore-model state machines because any Moore state machine can be implemented as a Mealy state machine, although the converse is not true.

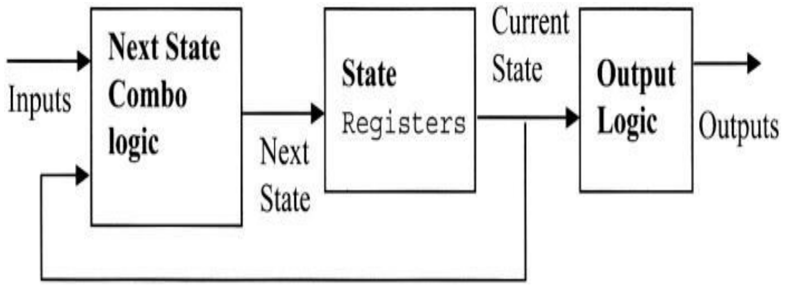
Moore machine: the outputs are properties of states themselves... which means that you get the output after the machine reaches a particular state, or to get some output your machine has to be taken to a state which provides you the output.The outputs are held until you go to some other state Mealy machine:

Mealy machines give you outputs instantly, that is immediately upon receiving input, but the output is not held after that clock cycle.

Block diagram of a Mealy machine



Block diagram of a Moore machine



Mealy machine	Moore machine
Outputs are function of current state and inputs.	Outputs are functions of current states only.
Output can change between changes between the state.	Output change only when the current state changes.
Output can change any number of times during the clock cycle, which may result in glitches on the output.	Output is delayed by one clock cycle, but is stable.
More output combinations are possible as the outputs are functions of inputs too.	Since the outputs are functions of only the current state, the number of output combinations are fewer with mealy machine.
If the inputs are not registered, the combinational paths could potentially be larger than the moore machine. Hence, a relatively lower frequency is ecpected compared to the moore machine.	Can expect higher frequency compared to mealy machine, as the combinational paths are typically shorter, and no input paths are involved.

1.1.7 Difference between one hot and binary encoding?

Common classifications used to describe the state encoding of an FSM are Binary (or highly encoded) and One hot.

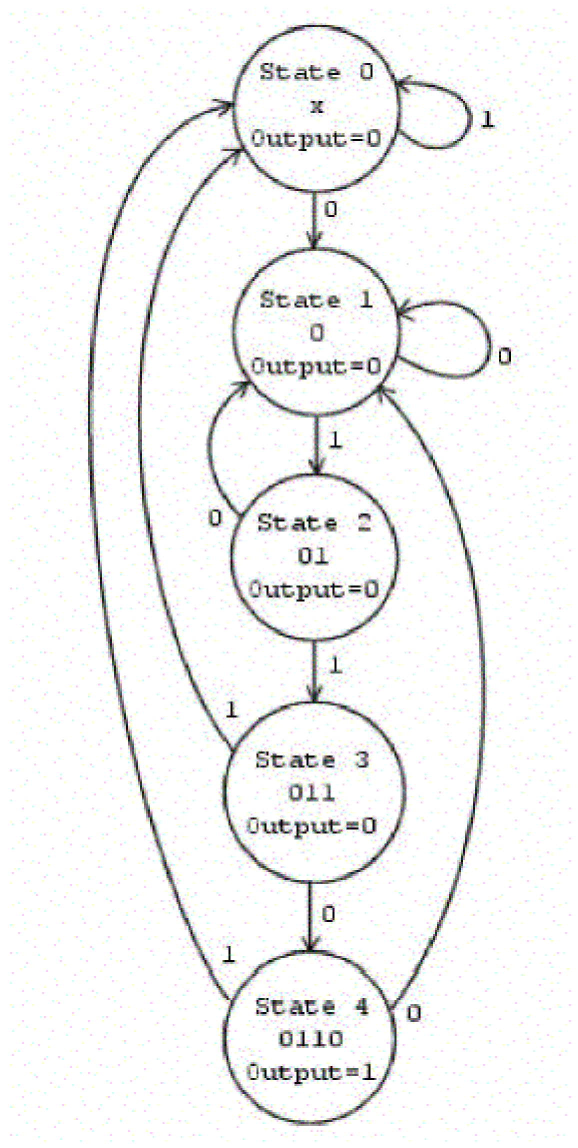
A binary-encoded FSM design only requires as many flip-flops as are needed to uniquely encode the number of states in the state machine. The actual number of flip-flops required is equal to the ceiling of the log-base-2 of the number of states in the FSM.

A onehot FSM design requires a flip-flop for each state in the design and only one flip-flop (the flip-flop representing the current or "hot" state) is set at a time in a one hot FSM design. For a state machine with 9- 16 states, a binary FSM only requires 4 flip-flops while a onehot FSM requires a flip-flop for each state in the design

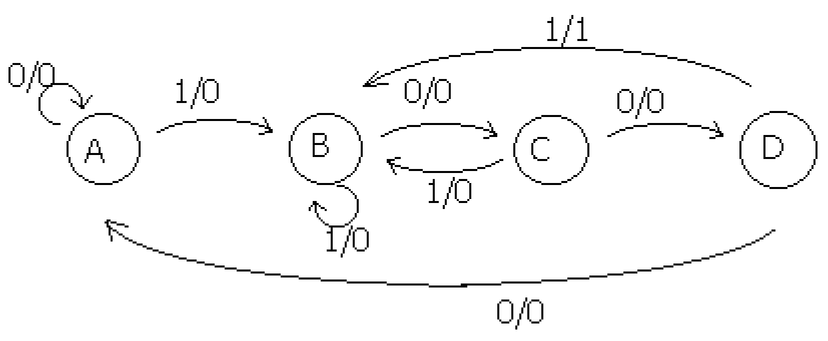
FPGA vendors frequently recommend using a onehot state encoding style because flip-flops are plentiful in an FPGA and the combinational logic required to implement a onehot FSM design is typically smaller than most binary encoding styles. Since FPGA performance is typically related to the combinational logic size of the FPGA design, onehot FSMs typically run faster than a binary encoded FSM with larger combinational logic blocks.

Binary encoding	**One-hot encoding**
Requires the fewer number of FF's to the represent the current state.	Number of FF's required is equal to the number of states in the FSM.
As there is combinational path in the output logic, its timing is not as good as the one-hot encoding mechanism.	Better output timing, as there is no output logic. Only clk-> q delays, and hence faster.
Preffered approach in ASICs unless the timing in output path is critical.	Don't need to optimize the state encoding, as each state has unique flop anyway.
Adding or deleting states requires tracking the side effects to the other states in the FSM.	Easy to mainain, that is adding or deleting states is easy, and doesn't effect the rest of the states.
Tedious to debug, since a wrong state transistion needs a walk through of the next state combinational logic.	Easy to debug, since a wrong state trasistion can be easily detected by looking at the current state values.
Critical path analysis requires tracking the combinational logic.	Easy to find critical paths during the Static Timing analysis.

1.1.8 Draw the state diagram to output a "1" for one cycle if the sequence "0110" shows up (the leading 0s cannot be used in more than one sequence)?

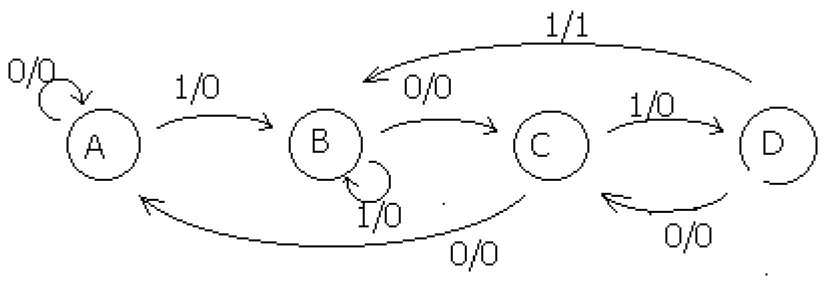


1.1.9 Sequence detector of 1001



State diagram of 1001 sequence recognizer

1.1.10 Sequence detector of 1011

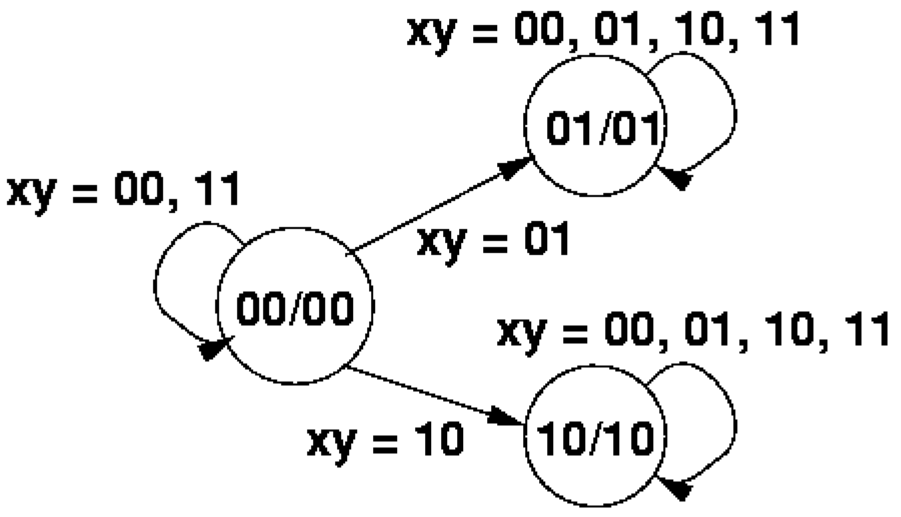


State diagram of 1011 sequence recognizer

1.1.11 Sequence detector for multiple bit inputs

Draw an FSM that takes two bits of input at a time. These two bits are bits i from two arbitrarily long strings. Assume the two strings are being fed in from the most significant bits first to the least significant bits. At time unit 1, An - 1 and Bn-1 are fed to the machine. At time unit 2, An - 2 and Bn-2 are fed to the machine.

The FSM should output 00 if the two values are equal so far, 10 if string A has a larger value (assume UB), and 01 if string B has a larger value.

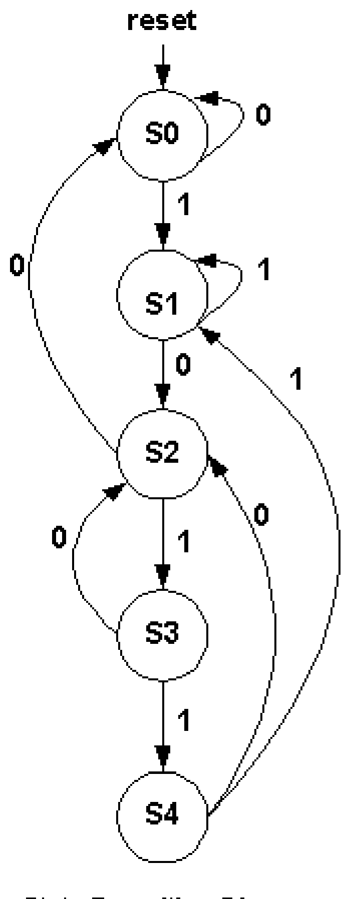


Basically, when you compare two n-bit strings, once you determine one number is larger than the other, then it stays larger. For example, consider X = 10100 and Y = 10011. The left most bit of both is 1, so when we see XY = 11, we know so far, they're equal (with the information provided). Then the next two bits are X3 = 0 and Y3 = 0 (which would be fed in as XY = 00). Again, they're equal so far. Finally, we reach the third bit, where X2 = 1 and Y2 = 0 (so XY = 10). At this point, we know that X is larger than Y (and we go into state 10).

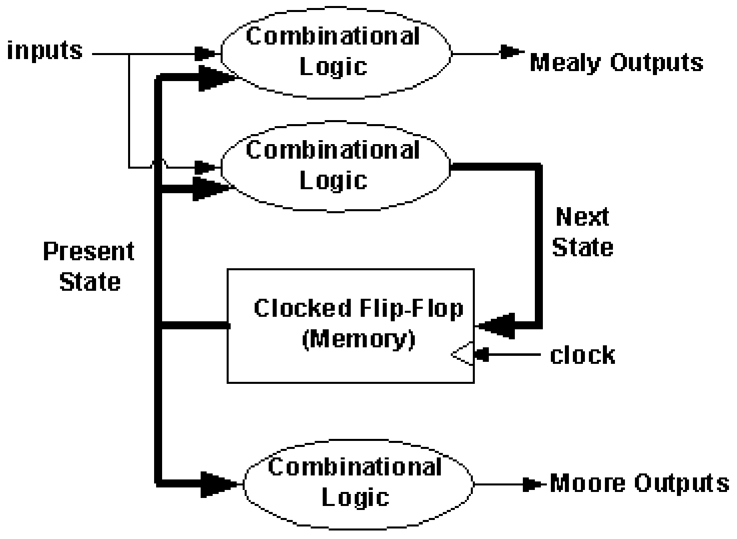
Even though more bits may be fed in, X will remain larger than Y.

To save space, we use one edge with multiple labels. Thus, XY = 00, 11 is equivalent to two edges, with one edge being labelled XY = 00 and the other edge being labelled XY = 11.

1.1.12 Design a FSM (Finite State Machine) to detect a 10110.



State Transition Diagram



FSM Block Diagram

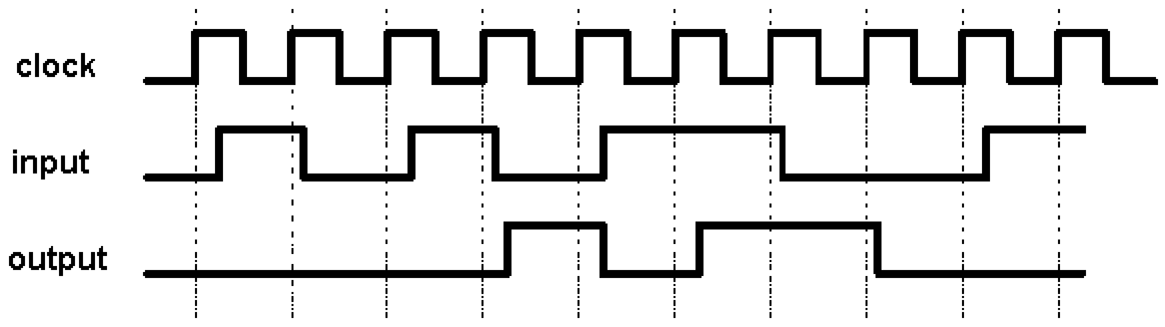
- ❖ Have a good approach to solve the design problem.
- ❖ Know the difference between Mealy, Moore, 1-Hot type of state encoding.
- ❖ Each state should have output transitions for all combinations of inputs.
- ❖ All states make transition to appropriate states and not to default if sequence is broken. e.g. S3 makes transition to S2 in example shown.
- ❖ Take help of FSM block diagram to write Verilog code.

1.1.13 Design a FSM (Finite State Machine) to detect more than one "1"s in last 3 samples

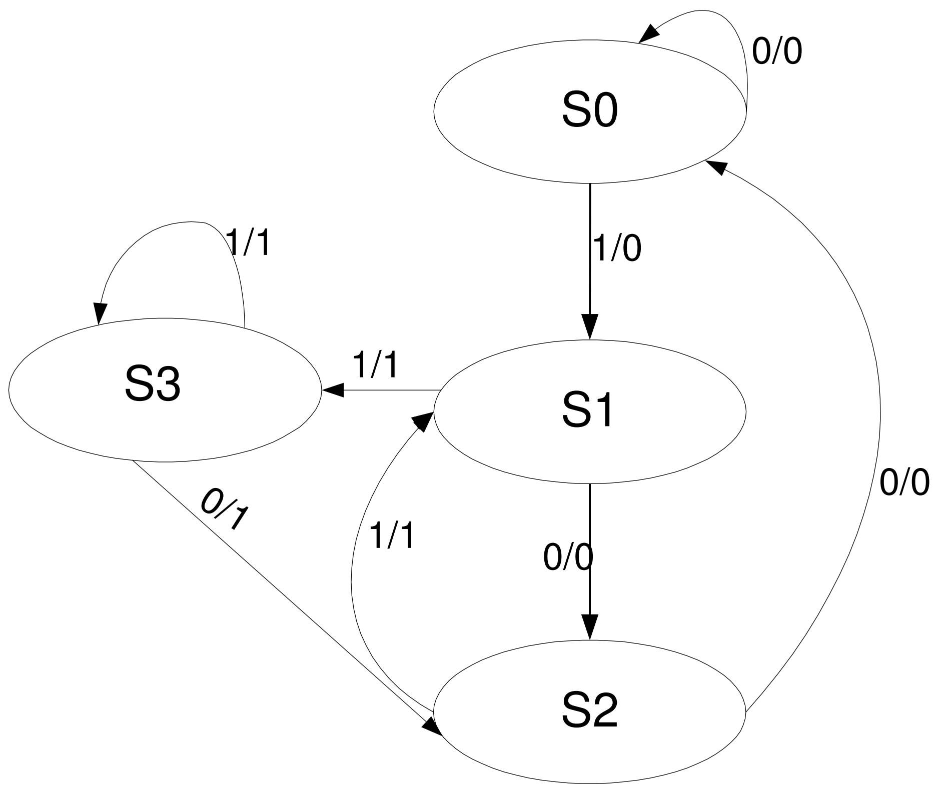
For example: If the input sampled at clock edges is 0 1 0 1 0 1 1 0 0 1

then output should be 0 0 0 1 0 1 1 1 0 0 as shown in timing diagram.

And yes, you have to design this FSM using not more than 4 states!!



The following is its Algorithm State Machine (ASM) flowchart based on Mealy machine



The following code shows using one always block to model the Mealy machine.

```
//Detect more than one 1s in last 3
//samples in a bit stream based on
//Mealy machine using one always block



module detectMoreThanOne1sWith1Always(
input reset, clock, D, output reg F);

    localparam[1:0] S0=2'b00,S1=2'b01,S2=2'b10,S3=2'b11;
    reg[1:0] CS;

    always@(negedge reset or posedge clock)
    begin
      if(!reset)
        CS<=S0;
      else
      begin
       case(CS)
       S0:
       begin
         F=0;
         if(D)
           CS=S1;
         else
           CS=S0;
       end
       S1:
       begin
         F=0;
         if(D)
         begin
           F=1;
           CS=S3;
         end
         else if(!D)
           CS=S2;
         else
           CS=S0;
       end
       S2:
       begin
         if(D)
         begin
           F=1;
           CS=S1;
         end
         else
         begin
           F=0;
           CS=S0;
         end
       end
       S3:
       begin
         F=1;
         if(D)
           CS=S3;
         else
           CS=S2;
       end
       endcase
      end
    end

endmodule
```

The following code shows using two always blocks to model the Mealy machine. One more is used to remove glitches.

```
//Detect more than one 1s in last 3
//samples in a bit stream based on
//Mealy machine
module detectMoreThanOne1sWith2Always(
input reset, clock, D, output reg F);

   localparam[1:0] S0=2'b00,S1=2'b01,S2=2'b10,S3=2'b11;
   reg[1:0] CS, NS;
   reg FF;

   always@(negedge reset or posedge clock)
   begin
     if(!reset)
       CS<=S0;
     else
       CS<=NS;
   end

   always@(*)
   begin
      case(CS)
      S0:
      begin
        FF=0;
        if(D)
          NS=S1;
        else
          NS=S0;
      end
      S1:
      begin
        FF=0;
        if(D)
        begin
```

```
          FF=1;
          NS=S3;
        end
        else if(!D)
          NS=S2;
        else
          NS=S0;
      end
      S2:
      begin
        if(D)
        begin
          FF=1;
          NS=S1;
        end
        else
        begin
          FF=0;
          NS=S0;
        end
      end
      S3:
      begin
        FF=1;
        if(D)
          NS=S3;
        else
          NS=S2;
      end
      endcase
   end

 //To remove glitches
   always@(negedge reset or posedge clock)
```

```
  begin
    if(!reset)
      F<=1'b0;
    else
      F<=FF;
  end

endmodule
```

1.1.14 Tell some of applications of buffer?

- They are used to introduce small delays
- They are used to eliminate cross talk caused due to inter electrode capacitance due to close routing.
- They are used to support high fanout,eg:bufg

1.1.15 Implement an AND gate using MUX?

This is the basic question that many interviewers ask. for and gate, give one input as select line, incase if u r giving b as select line, connect one input to logic '0' and other input to a.

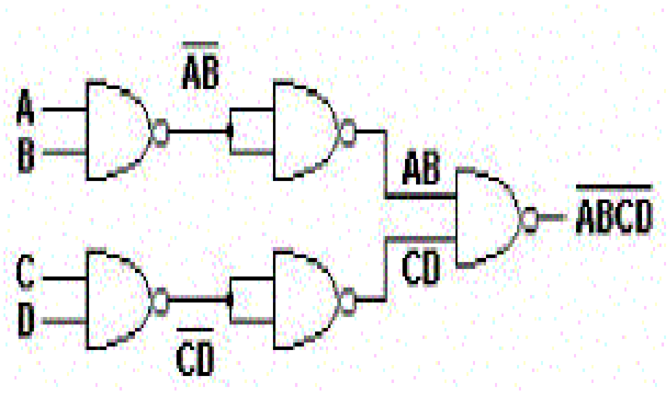
1.1.16 What will happen if contents of register are shifter left, right?

It is well known that in left shift all bits will be shifted left and LSB will be appended with 0 and in right shift all bits will be shifted right and MSB will be appended with 0 this is a straightforward answer

What is expected is in a left shift 2 multiply value eg: consider 0000_1110=14 a left shifts will make it 0001_110=28, it the same fashion right shifts will Divide the value by 2.

1.1.17 Design a four-input NAND gate using only two-input NAND gates.

Basically, you can tie the inputs of a NAND gate together to get an inverter, so...



1.1.18 Why are most interrupts active low?

This answers why most signals are active low

If you consider the transistor level of a module, active low means the capacitor in the output terminal gets charged or discharged based on low to high and high to low transition respectively. when it goes from high to low it depends on the pull down resistor that pulls it down and it is relatively easy for the output capacitance to discharge rather than charging. hence people prefer using active low signals.

1.1.19 Give two ways of converting a two input NAND gate to an inverter?

- Short the 2 inputs of the NAND gate and apply the single input to it.
- Connect the output to one of the input and the other to the input signal.

1.1.20 How can you convert an SR Flip-flop to a JK Flip-flop?

By giving the feed back we can convert, i.e !Q=>S and Q=>R.

Hence the S and R inputs will act as J and K respectively.

1.1.21 How can you convert the JK Flip-flop to a D Flip-flop?

By connecting the J input to the K through the inverter.

1.1.22 What is Race-around problem? How can you rectify it?

The clock pulse that remains in the 1 state while both J and K are equal to 1 will cause the output to complement again and repeat complementing until the pulse goes back to 0, this is called the race around problem. To avoid this undesirable operation, the clock pulse must have a time duration that is shorter than the propagation delay time of the F-F this is restrictive so the alternative is master-slave or edge-triggered construction.

1.1.23 How do you detect if two 8-bit signals are same?

XOR each bits of A with B (for e.g. A[0] xor B[0]) and so on.the o/p of 8 xor gates are then given as i/p to an 8-i/p nor gate. if o/p is 1 then A=B.

1.1.24 7-bit ring counter's initial state is 0100010. After how many clock cycles will it return to the initial state?

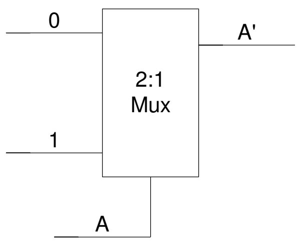
6 cycles

1.1.25 Design all the basic gates using 2:1 Multiplexer?

Using 2:1 MUX, (2 inputs, 1 output and a select line)

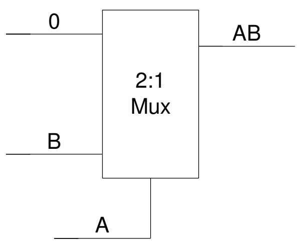
- **NOT Gate**

Give the input at the select line and connect I0 to 1 & I1 to 0. So if A is 1, we will get I1 that is 0 at the O/P.



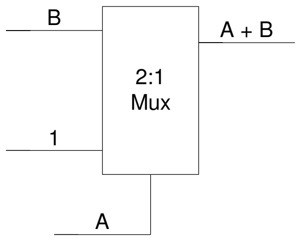
- **AND Gate**

Give input A at the select line and 0 to I0 and B to I1. O/p is A & B



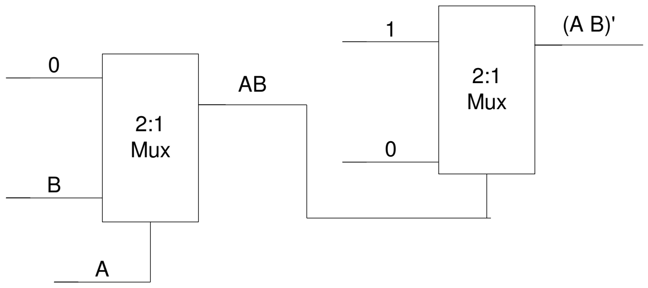
❖ OR Gate

Give input A at the select line and 1 to I1 and B to I0. O/p will be A | B



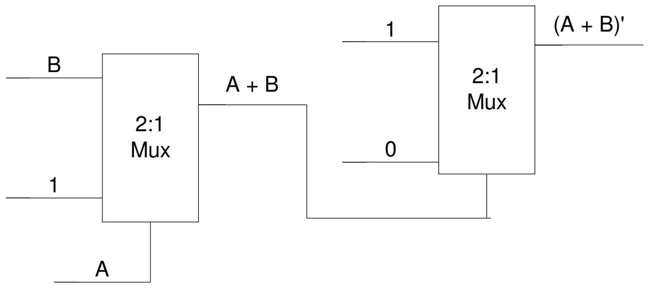
❖ NAND Gate

AND + NOT implementations together



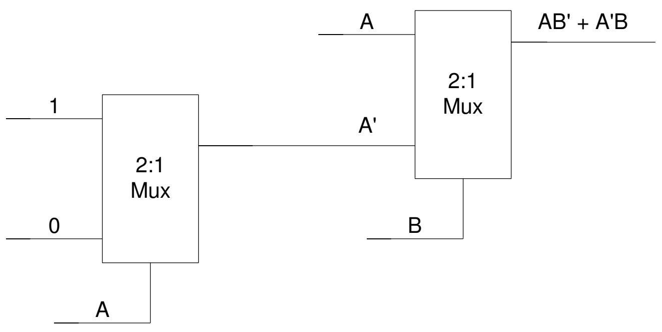
❖ NOR Gate

OR + NOT implementations together



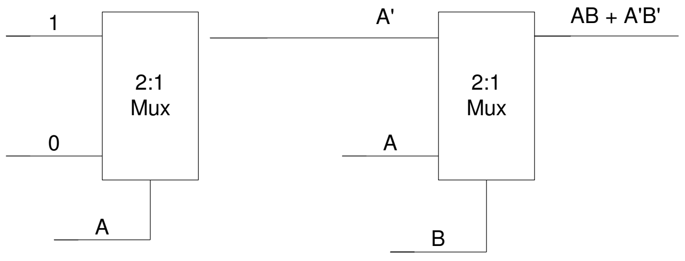
❖ XOR Gate

A at the select line B at I0 and ~B at I1.



❖ XNOR Gate

A at the select line B at I1 and ~B at I0.



1.1.26 Parity Generator

N number of XNOR gates are connected in series such that the N inputs (A0,A1,A2......) are given in the following way: A0 & A1 to first XNOR gate and A2 & O/P of First XNOR to second XNOR gate and so on..... Nth XNOR gates output is final output. How does this circuit work? Explain in detail?

If N=Odd, the circuit acts as even parity detector, ie the output will 1 if there are even number of 1's in the N input...This could also be called as odd parity generator since with this additional 1 as output the total number of 1's will be ODD.

If N=Even, just the opposite, it will be Odd parity detector or Even Parity Generator

1.1.27 Design a circuit that calculates the square of a number? It should not use any multiplier circuits. It should use Multiplexers and other logic?

This is interesting....

1^2=0+1=1

2^2=1+3=4

3^2=4+5=9

4^2=9+7=16

5^2=16+9=25

and so on

See a pattern yet? To get the next square, all you have to do is add the next odd number to the previous square that you found.See how 1,3,5,7 and finally 9 are added. Wouldn't this be a possible solution to your question since it only will use a counter,multiplexer and a couple of adders? It seems it would take n clock cycles to calculate square of n.

1.1.28 How will you implement a Full sub tractor from a Full adder?

All the bits of subtrahend should be connected to the XOR gate. Other input to the XOR being one. The input carry bit to the full adder should be made 1. Then the full adder works like a full sub tractor.

1.1.29 In a 3-bit Johnson's counter what are the unused states?

2(power n)-2n is the one used to find the unused states in Johnson counter.

So for a 3-bit counter it is 8-6=2.Unused states=2. The two unused states are 010 and 101.

1.1.30 What is an LFSR? List a few of its industry applications?

LFSR is a linear feedback shift register where the input bit is driven by a linear function of the overall shift register value. coming to industrial applications, as far as I know, it is used for encryption and decryption and in BIST(built-in-self-test) based applications.

1.1.31 Which circuit has a less propagation delay out of two counters specified below

First circuit is synchronous and second is "ripple" (cascading),

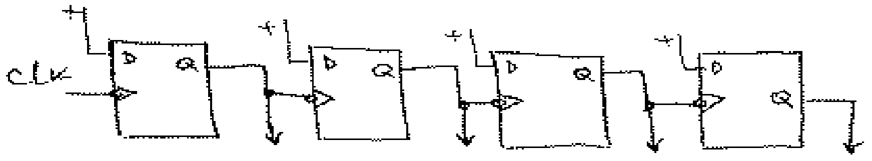
The synchronous counter will have lesser delay, as the input to each flop is readily available before the clock edge. Whereas the cascade counter will take long time as the output of one flop is used as clock to the other. So the delay will be propagating. For Eg: 16 state counter = 4 bit counter = 4 Flip flops Let 10ns be the delay of each flop The worst case delay of ripple counter = 10 * 4 = 40ns The delay of synchronous counter = 10ns only.(Delay of 1 flop).

1 - is ripple counter;

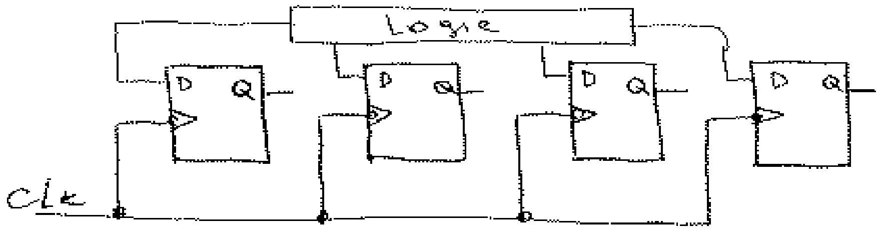
2 - synchronous.

Both consist of 4 FF, synchronous counter also has some logic to control it's operation.

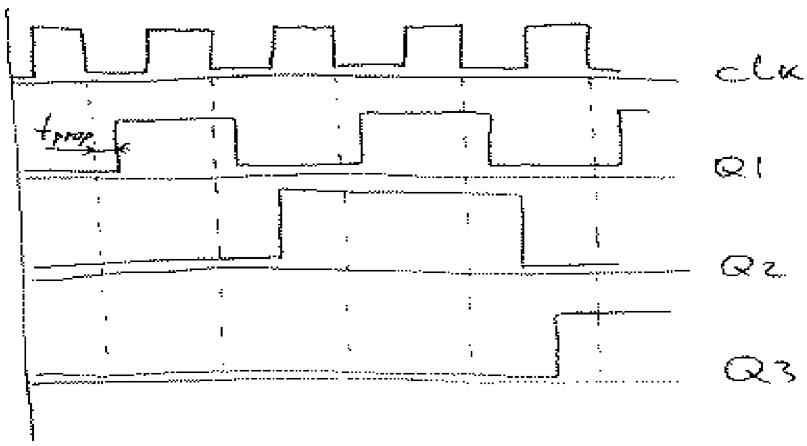
From diagram 3 (for ripple) and 4 (for synchronous) it is seen that propagation delay of ripple counter is 4* t_prop , while synchronous counter has only 1*



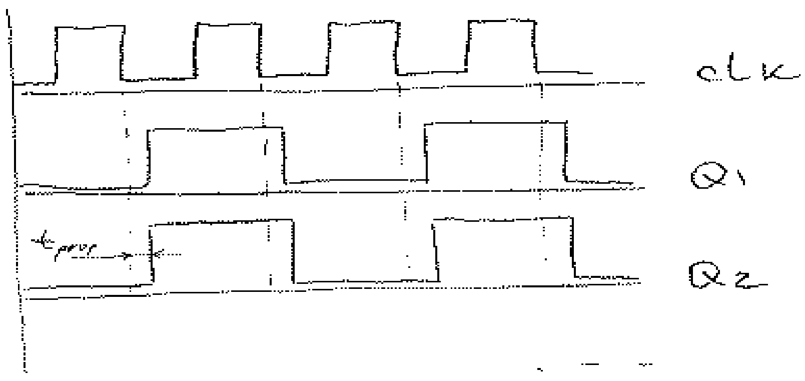
1



2



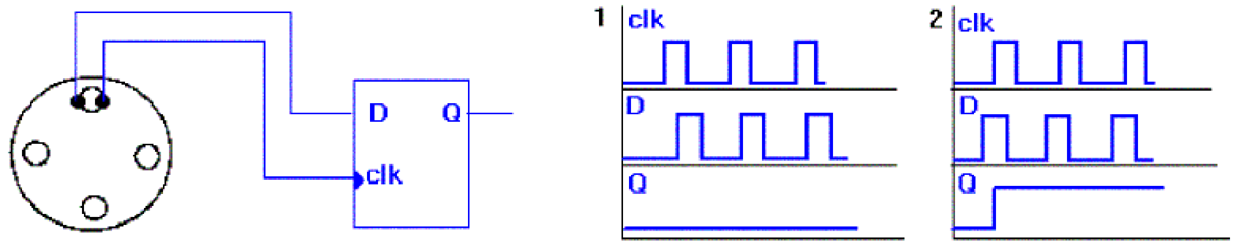
3



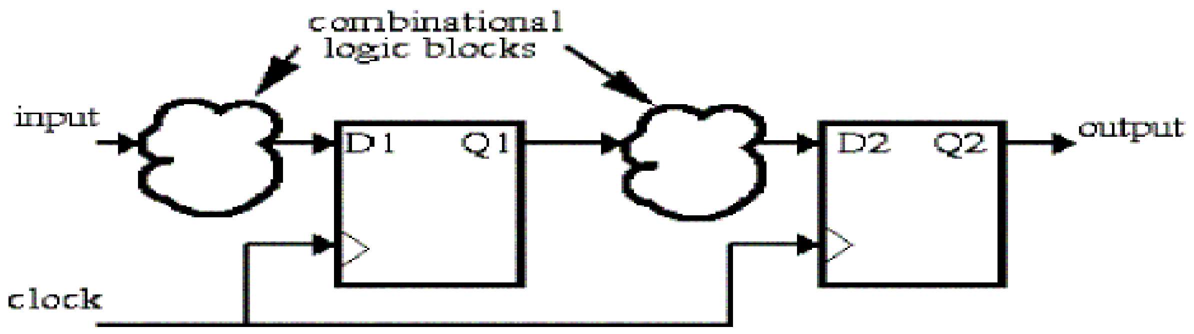
4

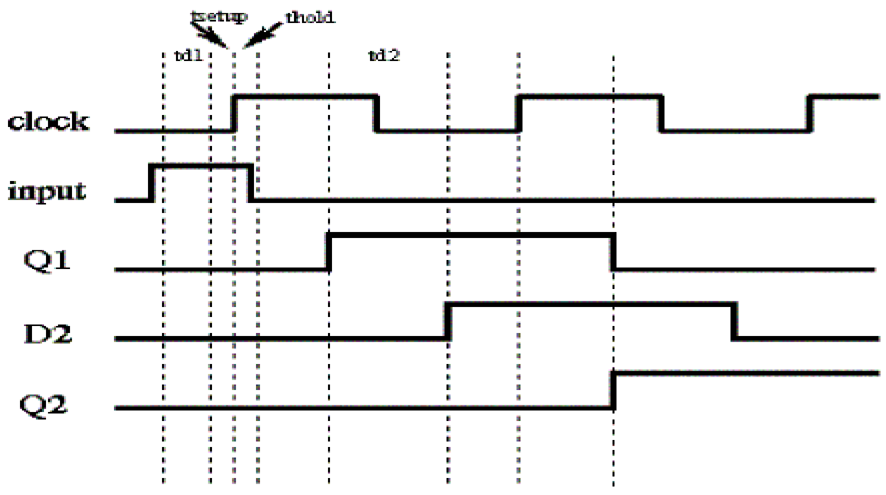
1.1.32 Derive minimum hardware to build a circuit to indicate the direction of rotating?

2 sensors are required to find out the direction of rotating. They are placed like at the drawing. One of them is connected to the data input of D flip-flop, and a second one - to the clock input. If the circle rotates the way clock sensor sees the light first while D input (second sensor) is zero - the output of the flip-flop equals zero, and if D input sensor "fires" first - the output of the flip-flop becomes high.



1.1.33 Draw timing diagrams for following circuit?





Timing Diagram

1.1.34 Implement the following circuits with two input NAND gates

3 input NAND

Connect :

a) A and B to the first NAND gate

b) Output of first Nand gate is given to the two inputs of the second NAND gate (this basically realizes the inverter functionality)

c) Output of second NAND gate is given to the input of the third NAND gate, whose other input is C

((A NAND B) NAND (A NAND B)) NAND C Thus, can be implemented using '3' 2-input NAND gates. I guess this is the minimum number of gates that need to be used.

3 input NOR:

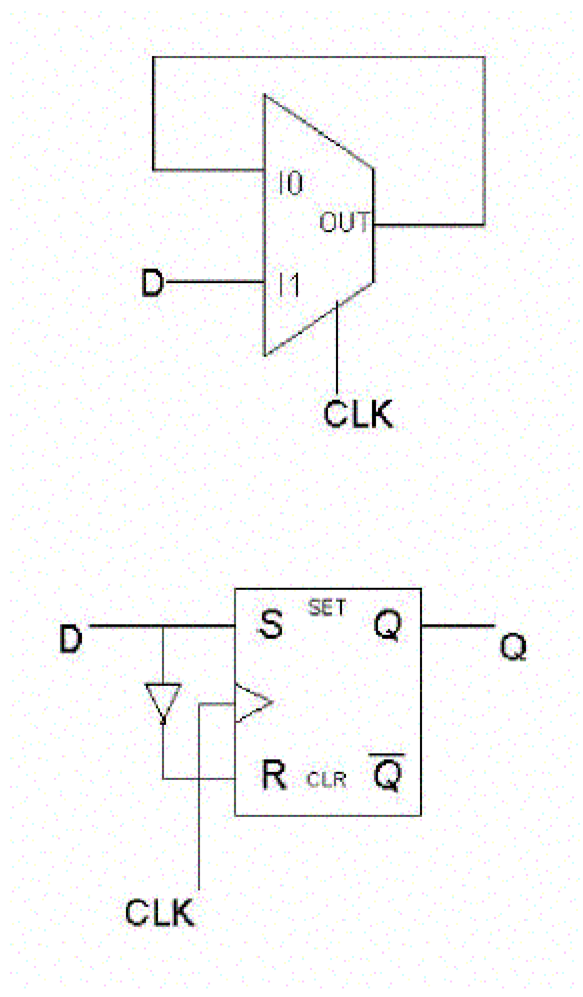
Same as above just interchange NAND with NOR ((A NOR B) NOR (A NOR B)) NOR C

3 input XNOR:

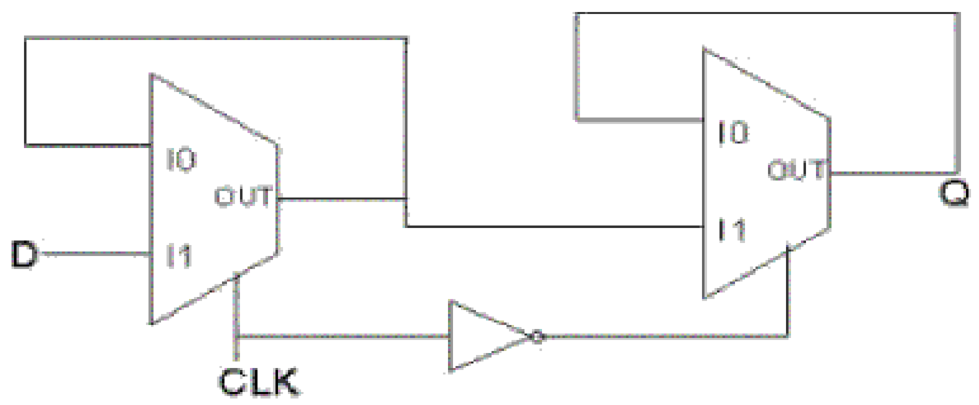
Same as above except the inputs for the second XNOR gate, Output of the first XNOR gate is one of the inputs and connect the second input to ground or logical '0'

((A XNOR B) XNOR 0)) XNOR C.

1.1.35 Design a D-latch using (a) using 2:1 MUX (b) from S-R Latch ?



1.1.36 How to implement a Master Slave flip flop using a 2 to 1 mux?



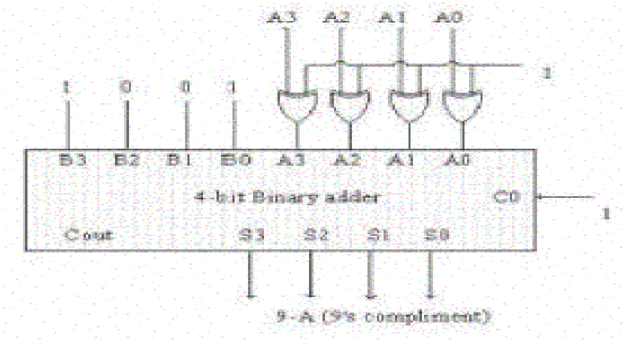
1.1.37 How many 2 input XOR gates required implementing 16-bit parity generator?

It is always n-1 where n is number of inputs. So 16 input parity generator will require 15 two input XOR gates.

1.1.38 9's complement of a BCD number.

Design a circuit for finding the 9's compliment of a BCD number using 4-bit binary adder and some external logic gates?

9's compliment is nothing but subtracting the given no from 9.So using a 4 bit binary adder we can just subtract the given binary no from 1001(i.e. 9). Here we can use the 2's compliment method addition.



1.1.39 What is Difference between write back and write through cache?

A caching method in which modifications to data in the cache aren't copied to the cache source until absolutely necessary. Write-back caching is available on many microprocessors, including all Intel processors since the 80486. With these microprocessors, data modifications to data stored in the L1 cache aren't copied to main memory until absolutely necessary. In contrast, a write-through cache performs all write operations in parallel -- data is written to main memory and the L1 cache simultaneously. Write-back caching yields somewhat better performance than write-through caching because it reduces the number of write operations to main memory. With this performance improvement comes a slight risk that data may be lost if the system crashes.

A write-back cache is also called a copy-back cache.

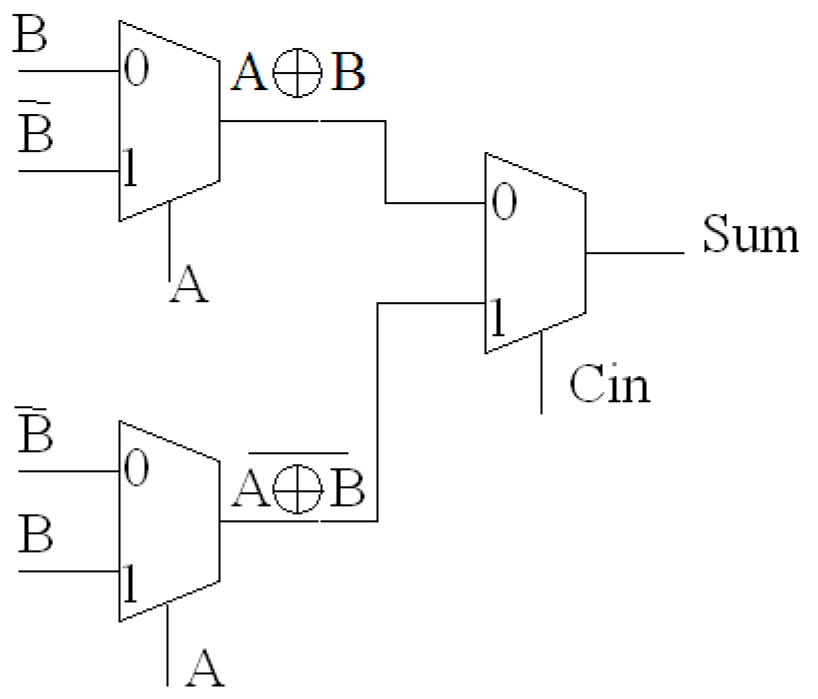
1.1.40 Difference between Synchronous, Asynchronous & Isynchronous communication?

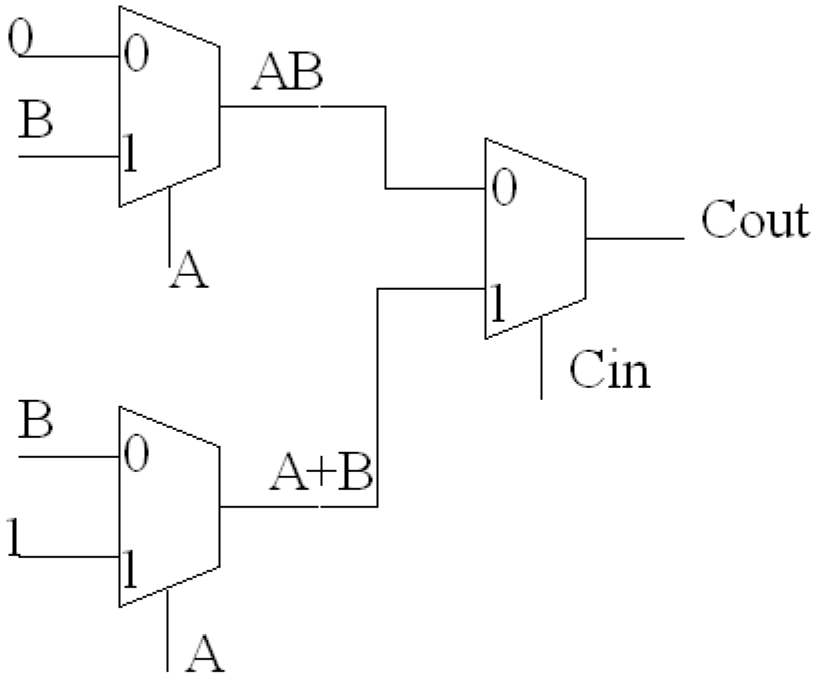
Sending data encoded into your signal requires that the sender and receiver are both using the same encoding/decoding method, and know where to look in the signal to find data. Asynchronous systems do not send separate information to indicate the encoding or clocking information. The receiver must decide the clocking of the signal on it's own. This means that the receiver must decide where to look in the signal stream to find ones and zeroes, and decide for itself where each individual bit stops and starts. This information is not in the data in the signal sent from transmitting unit.

Synchronous systems negotiate the connection at the data-link level before communication begins. Basic synchronous systems will synchronize two clocks before transmission, and reset their numeric counters for errors etc. More advanced systems may negotiate things like error correction and compression.

Time-dependent. it refers to processes where data must be delivered within certain time constraints. For example, Multimedia stream require an isochronous transport mechanism to ensure that data is delivered as fast as it is displayed and to ensure that the audio is synchronized with the video.

1.1.41 Implement a 1-bit full adder with 2 to 1 multiplexers?





www.asichowto.com

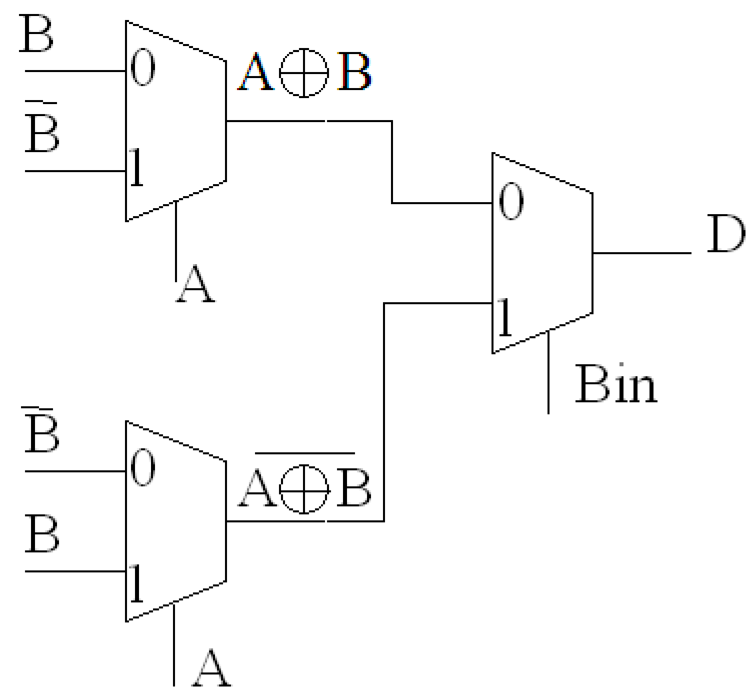
1.1.42 How to implement a 1-bit full sub tractor with 2 to 1 multiplexers?

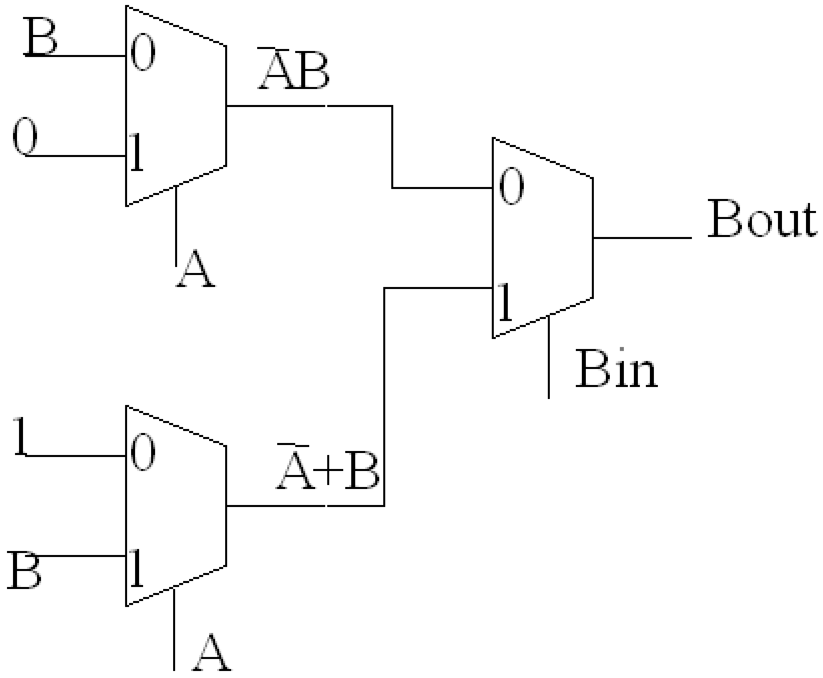
A 1-bit full subtractor is as follows.

1-bit full subtractor truth table

Bin	A	B	Bout	D(A-Bin-B)
0	0	0	0	0 0-0-0= 0
0	0	1	1	1 0-0-1=-1
0	1	0	0	1 1-0-0= 1
0	1	1	0	0 1-0-1= 0
1	0	0	1	1 0-1-0=-1
1	0	1	1	0 0-1-1=-2
1	1	0	0	0 1-1-0= 0
1	1	1	1	1 1-1-1=-1

One solution shows below.


B
$\overline{B}$
0
1
$A \oplus B$
A
0
1
D
Bin
$\overline{B}$
B
0
1
$\overline{A \oplus B}$
A


B
0
0
1
$\overline{A}B$
A
0
1
$Bout$
Bin
1
B
0
1
$\overline{A}+B$
A

1.1.43 What is Clock Gating?

Clock gating is one of the power-saving techniques used on many synchronous circuits including the Pentium 4 processor. To save power, clock gating refers to adding additional logic to a circuit to prune the clock tree, thus disabling portions of the circuitry where flip flops do not change state. Although asynchronous circuits by definition do not have a "clock", the term "perfect clock gating" is used to illustrate how various clock gating techniques are simply approximations of the data-dependent behavior exhibited by asynchronous circuitry, and that as the granularity on which you gate the clock of a synchronous circuit approaches zero, the power consumption of that circuit approaches that of an asynchronous circuit.

1.1.44 What is the basic difference between Analog and Digital Design?

Digital design is distinct from analog design. In analog circuits we deal with physical signals which are continuous in amplitude and time. Ex: biological data, sesimic signals, sensor output, audio, video etc.

Analog design is quite challenging than digital design as analog circuits are sensitive to noise, operating voltages, loading conditions and other conditions which has severe effects on performance. Even process technology poses certain topological limitations on the circuit. Analog designer has to deal with real time continuous signals and even manipulate them effectively even in harsh environment and in brutal operating conditions.

Digital design on the other hand is easier to process and has great immunity to noise. No room for automation in analog design as every application requires a different design. Where as digital design can be automated. Analog circuits generally deal with instantaneous value of voltage and current(real time). Can take any value within the domain of specifications for the device.consists of passive elements which contribute to the noise(thermal) of the circuit . They are usually more sensitive to external noise more so because for a particular function a analog design uses lot less transistors providing design challenges over process corners and temperature ranges. deals with a lot of device level physics and the state of the transistor plays a very important role Digital Circuits on the other hand deal with only two logic levels 0 and 1(Is it true that according to quantum mechanics there is a third logic level?) deal with lot more transistors for a particular logic, easier to design complex designs, flexible logic synthesis and greater speed although at the cost of greater power. Less sensitive to noise. design and analysis of such circuits is dependant on the clock. challenge lies in negating the timing and load delays and ensuring there is no set up or hold violation.

1.1.45 What are RTL, Gate, Metal and FIB fixes? What is a "sewing kits"?

There are several ways to fix an ASIC-based design. >From easiest to most extreme:

RTL Fix -> Gate Fix -> Metal Fix -> FIB Fix

First, let's review fundementals. A standard-cell ASIC consists of at least 2 dozen manufactured layers/masks. Lower layers conists of materialsmaking up the actual CMOS transistors and gates of the design. The upper 3-6 layers are metal layers used ti connect everything together. ASICs, of course, are not intended to be flexible like an FPGA, however, important "fixes" can be made during the manufacturing process. The progression of possible fixes in the manufacturing life cycle is as listed above.

An RTL fix means you change the Verilog/VHDL code and you resynthesize. This usually implies a new Plance&Route. RTL fixes would also imply new masks, etc. etc. In other words - start from scratch.

A Gate Fix means that a select number of gates and their interconections may be added or subtracted from the design (e.g. the netlist). This avoids resynthesis. Gate fixes preserve the previous synthesis effort and involve manually editing a gate-level netlist - adding gates, removing gates, etc. Gate level fixes affect ALL layers of the chip and all masks.

A Metal Fix means that only the upper metal interconnect layers are affected. Connections may be broken or made, but new cells may not be added. A Sewing Kit is a means of adding a new gate into the design while only affecting the metal layers. Sewing Kits are typically added into the initial design either at the RTL level or during synthesis by the customer and are part of the netlist. A Metal Fix affects only the top layers of the wafers and does not affect the "base" layers.

Sewing Kits are modules that contain an unused mix of gates, flip-flops or any other cells considered potentially useful for an unforseen metal fix. A Sewing Kit may be specified in RTL by instantiating the literal cells from the vendor library. The cells in the kit are usually connected such that each cell's output is unconnected and the inputs are tied to ground. Clocks and resets may be wired into the larger design's signals, or not.

A FIB Fix (Focussed Ion Beam) Fix is only performed on a completed chip. FIB is a somewhat exotic technology where a particle beam is able to make and break connections on a completed die. FIB fixes are done on individual chips and would only be done as a last resort to repair an otherwise defective prototype chip. Masks are not affected since it is the final chip that is intrusively repaired.

Clearly, these sorts of fixes are tricky and risky. They are available to the ASIC developer, but must be negotiated and coordinated with the foundry. ASIC designers who have been through enough of these fixes appreciate the value of adding test and fault-tolerant design features into the RTL code so that Software Fixes can correct minor silicon problems!.

1.1.46 Probability of collision. There is a triangle and on it there are 3 ants one on each corner and are free to move along sides of triangle what is probability that they will collide?

Ants can move only along edges of triangle in either of direction, let’s say one is represented by 1 and another by 0, since there are 3 sides eight combinations are possible, when all ants are going in same direction they won’t collide that is 111 or 000 so probability of collision is 2/8=1/4

1.1.47 FSM for the aggregate serial binary input divisible by 5. Draw the state diagram for a circuit that outputs a "1" if the aggregate serial binary input is divisible by 5. For instance, if the input stream is 1, 0, 1, we output a "1" (since 101 is 5). If we then get a "0", the aggregate total is 10, so we output another "1" (and so on).

Input	Sequence	Value	Output
1	1	1	0

0	10	2	0
1	101	5	1
0	1010	10	1
1	10101	21	0

We don't need to keep track of the entire string of numbers - if something is divisible by 5, it doesn't matter if it's 250 or 0, so we can just reset to 0.

So we really only need to keep track of "0" through "4".

Any number will be in one of the following forms:

5*n

5*n+1

5*n+2

5*n+3

5*n+4

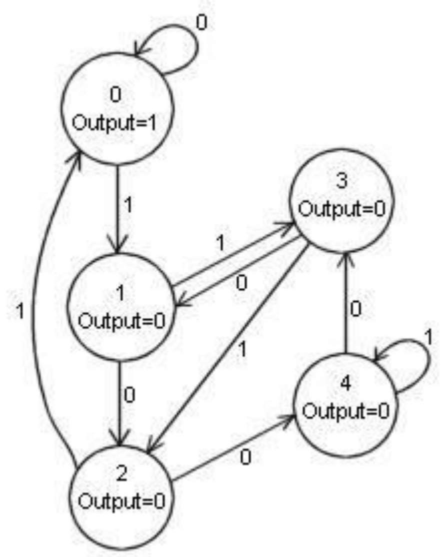
So you can have 5 states representing each of the above cases. Let us say X is the accumulated number. With the arrival of every new bit N the accumulated number becomes 2*X+N (left shift accumulated number + new bit).

You can draw the state diagram from here. I will just do it for (5*n+2) state.

If you get a 1 in this state the next state becomes, 2*(5n+2)+1 = 10n+5 is of form 5*n.

If you get a 0 in this state the next state becomes, 2*(5n+2)+0 = 10*n+4 is of form 5*n+4.....

and so on

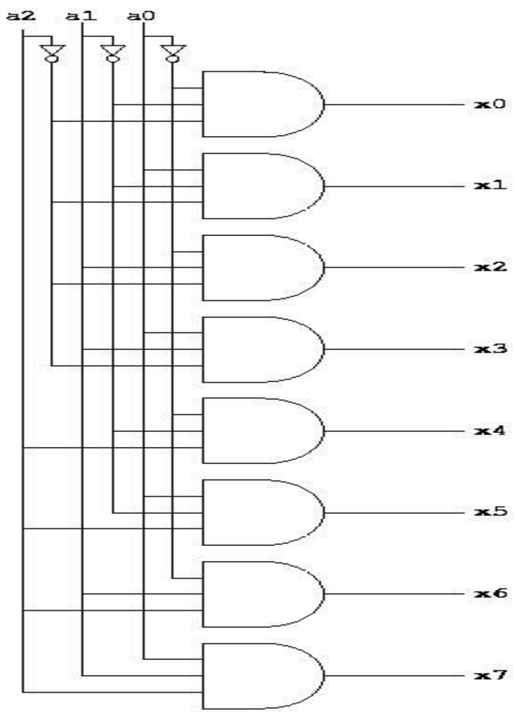


1.1.48 Design a 1-bit full adder using a decoder and 2 "or" gates?

Circuits decode the address inputs, i.e. it translates a binary number of n digits to 2n outputs, one of which (the one that corresponds to the value of the binary number) is 1 and the others of which are 0.

3:8 Decoder

a2	a1	a0	x7	x6	x5	x4	x3	x2	x1	x0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0



sum = OR(x1,x2,x3,x4,x5,x6,x7)

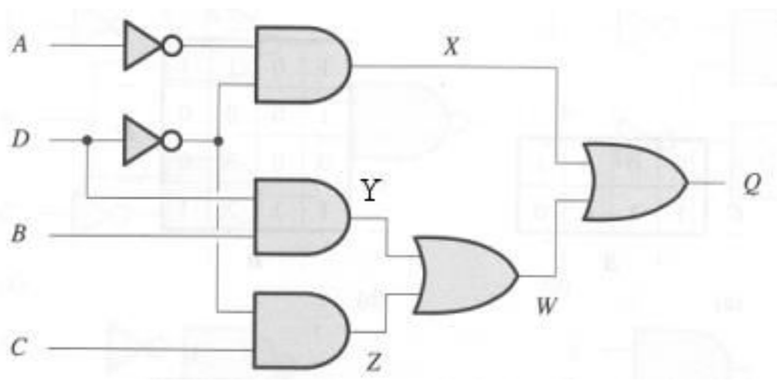
Cout= OR(x3,x7)

1.1.49 What is the function of a D-flip-flop, whose inverted outputs are connected to its input?

It acts as clock divider (divided by 2) circuits

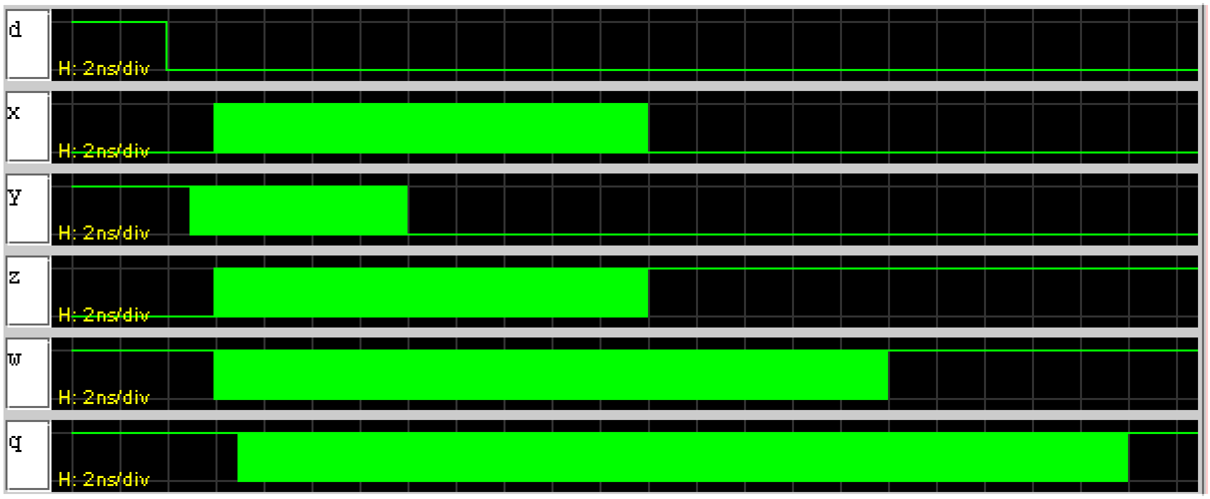
1.1.50 Consider the following combinational circuit

Suppose that each component in the circuit below has a propagation delay (tpd) of 10ns, a contamination delay (tcd) of 1ns, and negligable rise and fall times. Suppose initially that all four inputs are 1 for a long time and then the input D changes to 0.

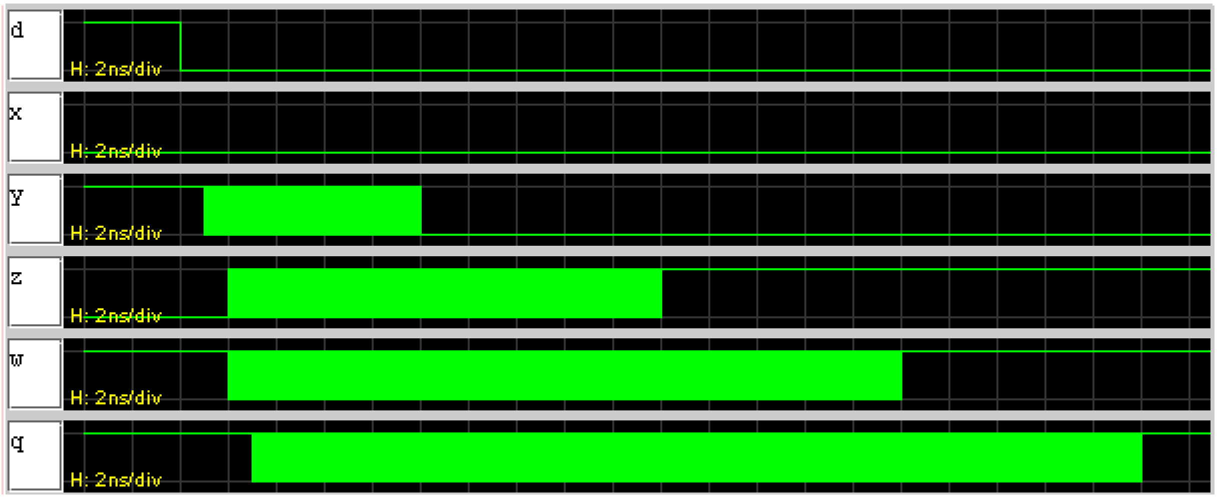


- **Draw a waveform plot showing how X, Y, Z, W and Q change with time after the input transition on D. First assume that the gates are not lenient. How will the waveforms change if the gates are lenient?**

Waveforms with non-lenient gates:



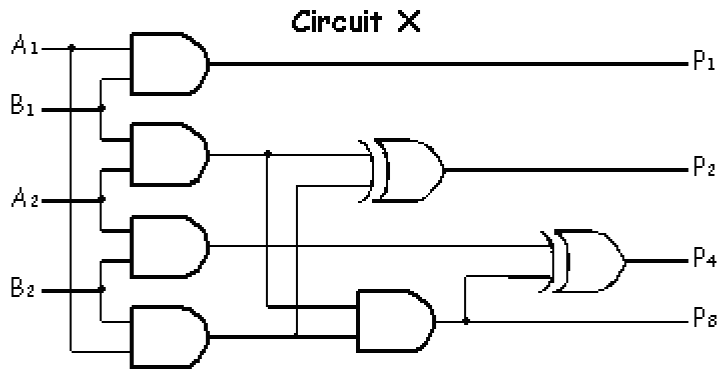
Waveforms with lenient gates:



Where we see that X doesn't change since the value of A is sufficient to determine the value of X.

1.1.51 The Mysterious Circuit X

- **Determine the function of the Circuit X, below, by writing out and examining its truth table. Give a minimal sum-of-products Boolean expression for each output.**



A_2	A_1	B_2	B_1	P_8	P_4	P_2	P_1
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	0	0	0	0	0
0	1	0	1	0	0	0	1
0	1	1	0	0	0	1	0
0	1	1	1	0	0	1	1
1	0	0	0	0	0	0	0
1	0	0	1	0	0	1	0
1	0	1	0	0	1	0	0
1	0	1	1	0	1	1	0
1	1	0	0	0	0	0	0
1	1	0	1	0	0	1	1
1	1	1	0	0	1	1	0
1	1	1	1	1	0	0	1

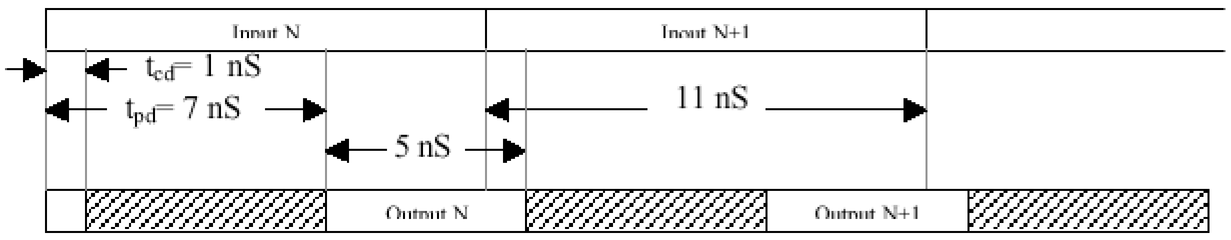
$$P_8 = A_2A_1B_2B_1$$
$$P_4 = A_2B_2\overline{B}_1 + A_2\overline{A}_1B_2$$
$$P_2 = A_2\overline{B}_2B_1 + A_2\overline{A}_1B_1 + A_1B_2\overline{B}_1 + \overline{A}_2A_1B_2$$
$$P_1 = A_1B_1$$

- **For Circuit X assume that AND gates have a propagation of 2 nS and a contamination delay of 1nS, while XOR gates have a propagation delay of 3 nS and contamination delay of 2 nS. Compute the aggregate contamination and propagation delays for Circuit X. What is the maximum frequency that the inputs of Circuit X be changed while insuring that all outputs are stable for 5 nS?**

The contamination delay of the circuit is obtained from the shortest path form an input to an output. In circuit X this path start at A1 (or B1) and ends at P1, encountering only one AND gates. Thus tCD = 1ns.

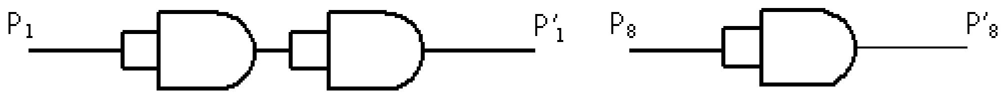
The propagation delay of the circuit is obatined from the longest path from an input to an output. In circuit X this path starts at any of the inputs and ends at P4, encoutering two AND gates and one XOR gate. Thus tPD = 2ns + 2ns + 3ns = 7ns.

The answer to the next part is best understood by drawing a timing diagram:



Thus if the inputs transition no faster than every 11ns (~90 MHz), the outputs will be stable for at least 5ns.

- **Suppose the gates below are added to Circuit X. How are the answers to part b) affected?**

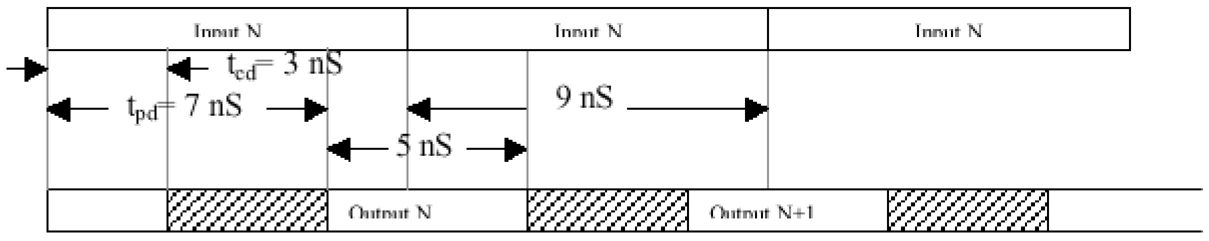


The shortest path from input to output now passes through three AND gates for outputs P1 and P8 and one AND gate and an XOR gate for outputs P2 and P4. Thus

tCD = min(1ns + 1ns + 1ns, 1ns + 2ns) = 3ns.

The path that creats the largest propagation delay in the circuit is still the path from any input to P4, so tPD is still 7ns.

With this new circuit the inputs can transition every 9ns and still guarantee that the outputs will be stable for 5ns.



1.1.52 Divisible by 3 FSM

Construct a "divisible-by-3" FSM that accepts a binary number entered one bit at a time, most significant bit first, and indicates with a light if the number entered so far is divisible by 3.

- **Draw a state transition diagram for your FSM indicating the initial state and for which states the light should be turned on. Hint: the FSM has 3 states.**

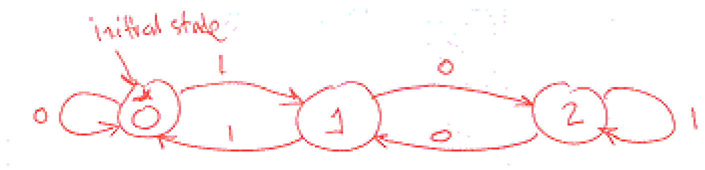
If the value of the number entered so far is N, then after the digit b is entered, the value of the new number N' is 2N + b. Using this fact:

if N is 0 mod 3 then for some p, N = 3p + 0. After the digit b is entered, N' = 6p + b. So N' is b mod 3.

if N is 1 mod 3 then for some p, N = 3p + 1. After the digit b is entered, N' = 6p + 2 + b. So N' is b+2 mod 3.

if N is 2 mod 3 then for some p, N = 3p + 2. After the digit b is entered, N' = 6p + 4 + b. So N' is b+1 mod 3.

This leads to the following transition diagram where the states are labeled with the value of N mod 3.



- **Construct a truth table for the FSM logic. Inputs include the state bits and the next bit of the number; outputs include the next state bits and the control for the light**

S1	S0	b	S1'	S0'	light
0	0	0	0	0	1
0	0	1	0	1	1
0	1	0	1	0	0
0	1	1	0	0	0
1	0	0	0	1	0
1	0	1	1	0	0

1.1.53 Questions on FSM

- **An FSM, M, is constructed by connecting the output of a 3-state FSM to the inputs of an 9-state FSM. M is then reimplemented using a state register with the minimum**

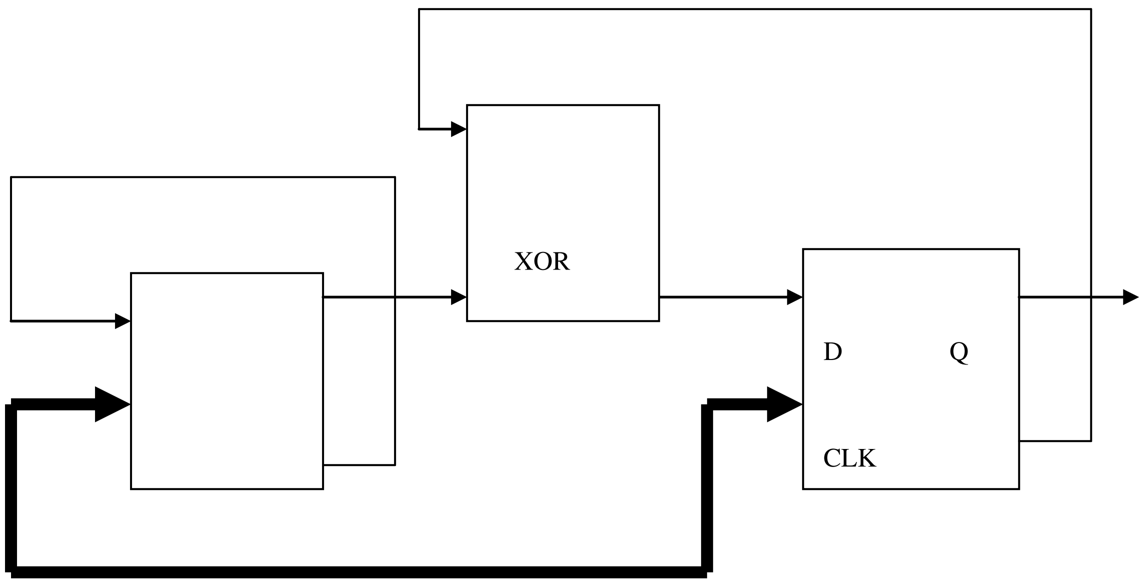
number of bits. What is the maximum number of bits that may be needed to re implement M?

M has 27 states, which require a total of 5 bits in the state register (not 2 + 4 bits!).

❖ **You connect M N-state FSMs, each have 1 input and 1 output, in series. What's an upper bound on the number of states in the resulting FSM?**

Each FSM can in theory be in one of its N states, so an upper bound on the number of states in the combined machine is NM.

1.1.54 For the two flip-flop configuration below, what is the relationship of the output at B to the clock frequency?



Output frequency is 1/4th the clock frequency, with 50% duty cycle

1.1.55 The fastest memory is

(i) DRAM, (ii) ROM, (iii) SRAM, (iv) Main memory

Ans : SRAM

1.1.56 Swapping without using a temporary variables.

(i) x = x+y;

y = x-y;

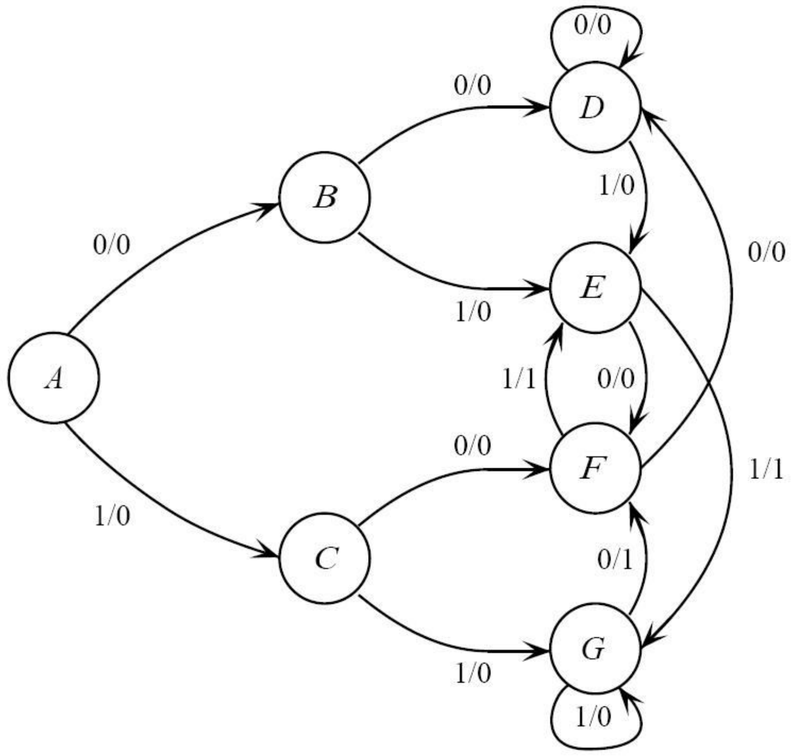
x = x-y;

(ii) x = x^y;

y = x^y;

x = x^y;

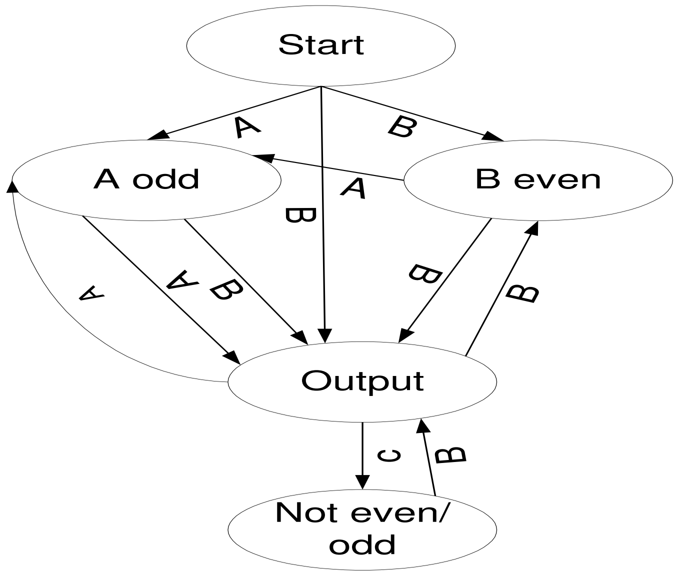
1.1.57 Design a state machine, that outputs a '1' one and only when two of the last 3 inputs are '1'. For example, if the input sequence is 0110_1110 then the output will be 0011_1101. Assume that the input 'x' is a single bit serial line.



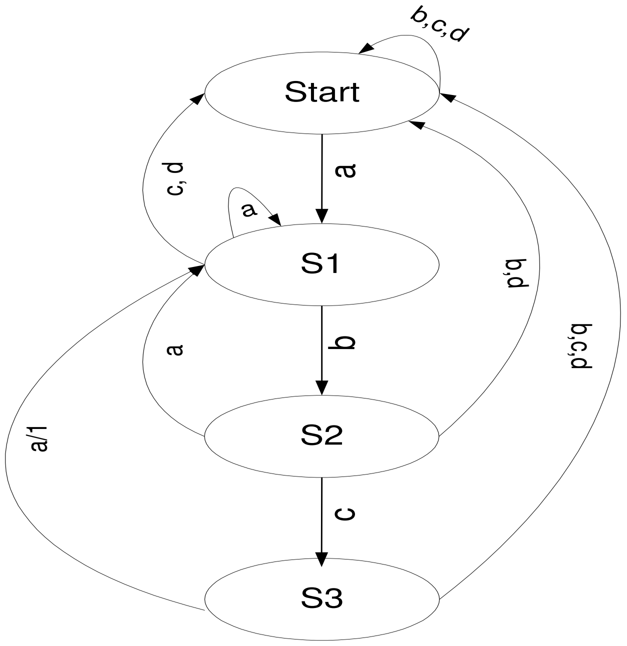
1.1.58 Minimum number of 2-input NAND gates that will be required to implement the function: Y = AB + CD + EF is

6 NAND gates are required.

1.1.59 Design a state-machine (or draw a state-diagram) to give an output '1' when the # of A's are even and of B's are odd. The input is in the form of a serial-stream (one-bit per clock cycle). The inputs could be of the type A, B or C. At any given clock cycle, the output is a '1', provided the # of A's are even and # of B's are odd. At any given clock cycle, the output is a '0', if the above condition is not satisfied.



1.1.60 To detect the sequence "abca" when the inputs can be a b c d.



1.2 Questions without answers

1.2.1 When will you use a latch and a flip-flop in a sequential design?

1.2.2 Given Two 4 bit nos A= 1001 B = 1100 HOW DO YOU XOR them using minimum number of gates? How do you nand them? How do your or them? (Hint: 4:1 Muxes).

1.2.3 Pulse clipper (or a return to zero) circuit. Design a clock to pulse circuit in Verilog / hardware gates.

1.2.4 Design a simple circuit based on combinational logic to double the output frequency.

1.2.5 Implement comparator that compares two 2-bit numbers A and B. The comparator should have 3 outputs: A > B, A < B, A = B. (Reduce the equations using Karnaugh Map) Do it two ways: - using combinational logic; - using multiplexers. Write HDL code for your schematic at RTL and gate level.

1.2.6 To enter the office people have to pass through the corridor. Once someone gets into the office the light turns on. It goes off when none is present in the room. There are two registration sensors in the corridor. Build a state machine diagram and design a circuit to control the light.

1.2.7 Design a 2bit up/down counter with clear using gates. (No verilog or vhdl).

1.2.8 Given a function whose inputs are dependent on its outputs. Design a sequential circuit.

1.2.9 Design a finite state machine to give a modulo 3 counter when x=0 and modulo 4 counter when x=1.

1.2.10 Minimize: S= A' + AB

1.2.11 Optical sensors A and B are positioned at 90 degrees to each other as shown in Figure. Half od the disc is white and remaining is black. When black portion is under sensor it generates logic 0 and logic 1 when white portion is under sensor. Design Direction finder block using digital components (flip flops and gates) to indicate speed. Logic 0 for clockwise and Logic 1 for counter clockwise. Cannot assume any fixed position for start.

1.2.12 Describe a finite state machine that will detect three consecutive coin tosses (of one coin) that results in heads.

1.2.13 In what cases do you need to double clock a signal before presenting it to a synchronous state machine?

1.2.14 Design a 2X1 mux using half adders

1.2.15 Using D FF and combo logic realize T FF.

1.2.16 Using D FF and COMBO logic realize JK FF.

1.2.17 What are the advantages and disadvanteages of Dynamic Logic ?

1.2.18 If the clock and D input of a D flipflop are shoted and clock connected to this circuit, how will it respond?

1.2.19 Make a JK FF using a D FF and 4->1 MUX.

1.2.20 What are the issues if the duty cycle of the clock in a digital ckt is changed from 50%?

1.2.21 Design a two bit comparator with and without using MUX.

1.2.22 Design a square wave generator which takes only one positive edge trigger

1.2.23 For the given _expression Y=A'B'C+A'BC+AB'C+ABC+ABC' realize using the following

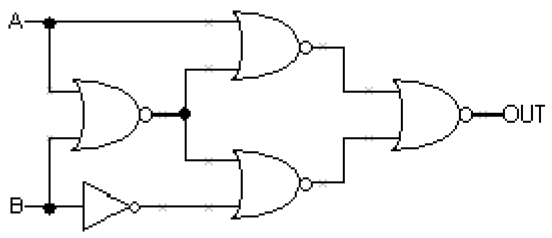
- 2 input and 3input NAND gate
- 2 input and 3 input NOR gate
- AND,OR, INVERTER.
- INVERTER;

1.2.24 Use 2->1 MUX to implement the following _expression Y=A+BC'+BC(A+B).

1.2.25 Using a FF and gates. Make a memory (i.e include RD, WR etc.)

1.2.26 Suppose we are building the following circuit using only three components:

- Inverter: tcd = 0.5ns, tpd = 1.0ns, tr = tf = 0.7ns
- 2-input NAND: tcd = 0.5ns, tpd = 2.0ns, tr = tf = 1.2ns
- 2-input NOR: tcd = 0.5ns, tpd = 2.0ns, tr = tf = 1.2ns



What is tpd (Propagation delay) for this circuit?

What is tcd (Contamination delay) for this circuit?

What is tpd of the fastest equivalent circuit built using only above 3 components?

Hint:

tpd - Max. cumulative propagation delay considering all path b/w input and output.

tcd - Min. cumulative contamination delay.

2 Verilog

2.1 Questions with answers

2.1.1 Blocking Vs Non-Blocking

self triggering blocks -

```
module osc2 (clk);
output clk;
reg clk;
initial #10 clk = 0;
always @(clk) #10 clk <= ~clk;
endmodule
```

After the first @(clk) trigger, the RHS expression of the nonblocking assignment is evaluated and the LHS value scheduled into the nonblocking assign updates event queue.

Before the nonblocking assign updates event queue is "activated," the @(clk) trigger statement is encountered and the always block again becomes sensitive to changes on the clk signal. When the nonblocking LHS value is updated later in the same time step, the @(clk) is again triggered.

```
module osc1 (clk);
output clk;
reg clk;
initial #10 clk = 0;
always @(clk) #10 clk = ~clk;
endmodule
```

Blocking assignments evaluate their RHS expression and update their LHS value without interruption. The blocking assignment must complete before the @(clk) edge-trigger event can be scheduled. By the time the trigger event has been scheduled, the blocking clk assignment has completed; therefore, there is no trigger event from within the always block to trigger the @(clk) trigger.

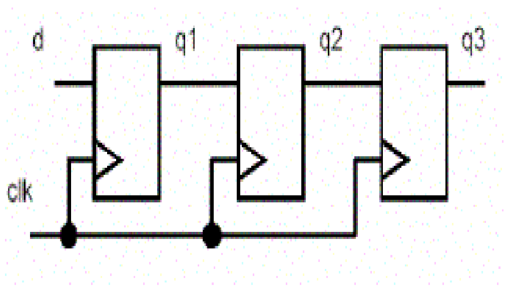
Bad modeling: - (using blocking for seq. logic)

```
always @(posedge clk) begin
q1 = d;
q2 = q1;
q3 = q2;
end
```

Race Condition

```
always @(posedge clk) q1=d;
always @(posedge clk) q2=q1;
always @(posedge clk) q3=q2;

always @(posedge clk) q2=q1;
always @(posedge clk) q3=q2;
always @(posedge clk) q1=d;

always @(posedge clk) begin
q3 = q2;
q2 = q1;
q1 = d;
end
```

Bad style but still works



Good modeling: -

```
always @(posedge clk) begin
q1 <= d;
q2 <= q1;
q3 <= q2;
end
```

```
always @(posedge clk) begin
q3 <= q2;
q2 <= q1;
q1 <= d;
end
```

No matter of sequence for Nonblocking

```
always @(posedge clk) q1<=d;
always @(posedge clk) q2<=q1;
always @(posedge clk) q3<=q2;

always @(posedge clk) q2<=q1;
always @(posedge clk) q3<=q2;
always @(posedge clk) q1<=d;
```

Good Combinational logic :- (Blocking)

```
always @(a or b or c or d) begin
tmp1 = a & b;
tmp2 = c & d;
y = tmp1 | tmp2;
end
```

Bad Combinational logic :- (Nonblocking)

```
always @(a or b or c or d) begin will simulate incorrectly…
tmp1 <= a & b; need tmp1, tmp2 insensitivity
tmp2 <= c & d;
y <= tmp1 | tmp2;
end
```

Mixed design: -

Use Nonblocking assignment.

In case on multiple non-blocking assignments last one will win.

2.1.2 What are the differences between SIMULATION and SYNTHESIS

Simulation <= verify your design.

synthesis <= Check for your timing

Simulation is used to verify the functionality of the circuit.. a)Functional Simulation:study of ckt's operation independent of timing parameters and gate delays. b) Timing Simulation :study including estimated delays, verify setup,hold and other timing requirements of devices like flip flops are met.

Synthesis:One of the foremost in back end steps where by synthesizing is nothing but converting VHDL or VERILOG description to a set of primitives(equations as in CPLD) or components(as in FPGA'S)to fit into the target technology.Basically the synthesis tools convert the design description into equations or components

2.1.3 Equivalence between VHDL and C?

There is concept of understanding in C there is structure.Based upon requirement structure provide facility to store collection of different data types.

In VHDL we have direct access to memory so instead of using pointer in C (and member of structure) we can write interface store data in memory and access it.

2.1.4 Difference between RTL and Behavioral modeling

Register transfer language means there should be data flow between two registers and logic is in between them for end registers data should flow.

Behavioral means how hardware behaves determine the exact way it works we write using HDL syntax.For complex projects it is better mixed approach or more behavioral is used.

Behavioral modeling represents the circuit at a very high level of abstraction. Design at this level resembles

C programming more than it resembles digital circuit design.

2.1.5 What is the difference between a Verilog task and a Verilog function?

Functions:

- A function is unable to enable a task however functions can enable other functions.
- A function will carry out its required duty in zero simulation time.
- Within a function, no event, delay or timing control statements are permitted.
- In the invocation of a function their must be at least one argument to be passed.
- Functions will only return a single value and can not use either output or inout statements.
- Functions are synthesysable.
- Disable statements canot be used.

- Function canot have nonblocking statements.

Task:

- Tasks are capable of enabling a function as well as enabling other versions of a Task.
- Tasks also run with a zero simulation however they can if required be executed in a non zero simulation time.
- Tasks are allowed to contain any of these statements.
- A task is allowed to use zero or more arguments which are of type output, input or inout.
- A Task is unable to return a value but has the facility to pass multiple values via the output and inout statements.
- Tasks are not synthesisable.
- Disable statements can be used.

2.1.6 Why a function canot call a task?

As functions does not consume time,it can do any operation which doesnot consume time. Mostly tasks are written which consumes time. So a task call inside a function blocks the further execution of function until it finished. But its not true. A function can call task if the task call consumes zero time, but the IEEE LRM doesn't allow.

2.1.7 Why tasks are not synthesized?

Wrong question! Tasks can be synthesized if it doesn't consume time.

2.1.8 Why a function should return a value?

There is no strong reason for this in Verilog. This restriction is removed in SystemVerilog.

2.1.9 Why a function should have at least one input?

There is no strong reason for this in verilog. I think this restriction is not removed fin SystemVerilog.Some requirements where the inputs are taken from the global signal,those functions dont need any input. A work around is to use a dummy input.

2.1.10 Why a task canot return a value?

If tasks can return values,then Lest take a look at the following example.

A=f1(B)+f2(C);

and f1 and f2 had delays of say 5 and 10? When would B and C be sampled, or globals inside f1 and f2 be sampled? How long does then entire statement block? This is going to put programmers in a bad situation. So languages gurus made that tasks can't return.

2.1.11 Why a function canot have delays?

The answer is same as above. But in Open Vera, delays are allowed in function. A function returns a value and therefore can be used as a part of any expression. This does not allow any delay in the function.

2.1.12 Why disable statements are not allowed in functions?

If disble statement is used in function,it invalids the function and its return value. So disable statements are not allowed in function.

2.1.13 What is reentrant tasks and functions

Tasks and functions without the optional keyword automatic are static , with all declared items being statically allocated. These items shall be shared across all uses of the task and functions executing concurrently. Task and functions with the optional keyword automatic are automatic tasks and functions. All items declared inside automatic tasks and functions are allocated dynamically for each invocation. Automatic task items and function items can not be accessed by hierarchical references.

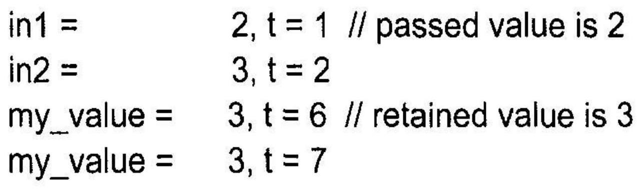
Reentrant task	Static task
Has the key word **automatic** between the **task** keyword and identifier.	Doesn’t have the keyword **automatic** between the **task** keyword and identifier.
Variables declared within the task are allocated dynamically for each concurrent task call	Variable declarations within the task are allocated statically.
All variables will be replicated in the each cocurrent call to store state specific to that invocation.	Each concurrent call to the task will OVERWRITE the statically allocated local variables of the task from all other concurrent calls to the task.
Variables declared are de-allocated at the end of the task invocation.	Variable retain their values between the invocations.
Task items cannot be accessed by hierarchical inferences.	Task can be accessed by hierarchical inferences.
Task items shall be allocated new across all uses of the task executing concurrently.	Task items can be shared across all uses of the task executing concurrently.

```
module modify_taskval;

integer out_val;

task automatic modify_value;
  input [1:0] in_value;
  output [3:0] out_value;
  reg [1:0] my_value;
begin
// syntax error to use nonblocking assignment with
```

```
// automatic variables
  my_value = in_value; // blocking assignment
#5
  $display("my_value = \t%0d, t = %0d",
            my_value, $time);
  out_value = my_value + 2;
end
endtask

initial begin
  fork
    begin // First parallel call
      #1
      $display("in1= \t\t%0d, t = %0d",2, $time);
      modify_value(2, out_val);
    end
    begin // Second parallel call
      #2
      $display("in2 = \t\t%0d, t = %0d",3, $time);
      modify_value(3, out_val);
    end
  join
end

endmodule
```

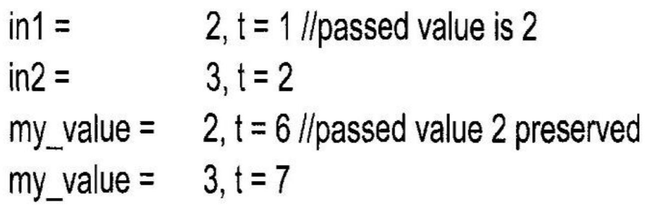
In the above example, my_value is a local variable in the task modify_value. Whenever this task is called, the input in_value is assigned to the local variable after 5 simulation timeunits. Within the initialbegin,there is a fork-join, which launches two parallel processes. One starts after simulation timeunit #1, and other after #2. The first process assigns a value of 2 to the output of the task, and the second one assigns a value of 3 to the output of the task.

Running the simulation with the above code, but without the automatic keyword, provides the following display:



```
in1 =          2, t = 1  // passed value is 2
in2 =          3, t = 2
my_value =     3, t = 6  // retained value is 3
my_value =     3, t = 7
```

Now, with the keyword automatic between the task and task name, the following is the output:



```
in1 =          2, t = 1 //passed value is 2
in2 =          3, t = 2
my_value =     2, t = 6 //passed value 2 preserved
my_value =     3, t = 7
```

2.1.14 Given the following Verilog code, what value of "a" is displayed?

```
always @(clk) begin
a = 0;
a <= 1;
$display(a);
end
```

This is a tricky one! Verilog scheduling semantics basically imply a four-level deep queue for the current simulation time:

1: Active Events (blocking statements)

2: Inactive Events (#0 delays, etc)

3: Non-Blocking Assign Updates (non-blocking statements)

4: Monitor Events ($display, $monitor, etc).

Since the "a = 0" is an active event, it is scheduled into the 1st "queue".The "a <= 1" is a non-blocking event, so it's placed into the 3rd queue.

Finally, the display statement is placed into the 4th queue.

Only events in the active queue are completed this sim cycle, so the "a = 0" happens, and then the display shows a = 0. If we were to look at the value of a in the next sim cycle, it would show 1.

```
always @(clk) begin
a = 0;
a <= 1;
#1;
$display(a);
end
```

The output will be 1 printed for every clock cycle.

```
always @(clk) begin
a = 0;
$display(a);
a <= 1;
$display(a);
end
```

The output will be 0 printed for every clock cycle.

```
always @(clk) begin
a = 0;
$display(a);

a <= 1;
#1;
$display(a);
end
```

The output will be 0,1 printed for every clock cycle.

```
always @(clk) begin
a = 0;
a <= 1;
$monitor(a);
end
```

The output will be 1 printed for every clock cycle.

2.1.15 What is the difference conditional operator and if-else

Let us take example of c = foo ? a : b; and if (foo) c = a; else c = b;

The ? merges answers if the condition is "x", so for instance if foo = 1'bx, a = 'b10, and b = 'b11, you'd get c = 'b1x.

On the other hand, if treats Xs or Zs as FALSE, so you'd always get c = b.

2.1.16 What are the various functional verification methodologies

Ans: TLM(Transaction Level Modelling)

Linting

RTL Simulation (Enivronment involving : stimulus generators, monitors, response checkers, transactors)

Gate level Simulation

Mixed-signal simulations

Regression

2.1.17 Difference between signal/variable.

Variables

- ❖ Variables are used for local storage of data
- ❖ Variables are generally not available to multiple components and processes.
- ❖ Variables will be updated after the signals.

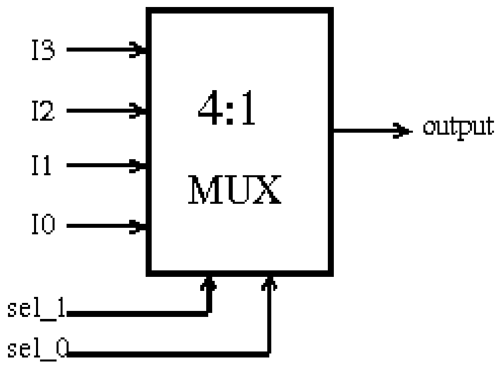
Signals

- ❖ Signals are used for communication between components
- ❖ Signals can be seen as real, physical signals
- ❖ Some delay must be incurred in a signal assignment.
- ❖ Signals will be updated first, these are placed in active event queue.

2.1.18 Write decoder functionality in only one statement in verilog

```
<code>
module decoder(
  // Outputs
  dout,
  // Inputs
  din
  );
  input [3:0] din;
  output [15:0] dout;
  assign dout =  (din==15)? 15:
                (din==14)? 14:
                (din==13)? 13:
                (din==12)? 12:
                (din==11)? 11:
                (din==10)? 10:
                (din==9)? 9:
                (din==8)? 8:
                (din==7)? 7:
                (din==6)? 6:
                (din==5)? 5:
                (din==4)? 4:
                (din==3)? 3:
                (din==2)? 2:
                (din== 1)? 1:0;
endmodule // decoder
```

2.1.19 Design a 4:1 mux in Verilog.



Multiple styles of coding. e.g.

Using if-else statements

```
if(sel_1 == 0 && sel_0 == 0) output = I0;
else if(sel_1 == 0 && sel_0 == 1) output = I1;
else if(sel_1 == 1 && sel_0 == 0) output = I2;
else if(sel_1 == 1 && sel_0 == 1) output = I3;
```

Using case statement

```
case ({sel_1, sel_0})
    00 : output = I0;
    01 : output = I1;
    10 : output = I2;
    11 : output = I3;
    default : output = I0;
endcase
```

- ❖ **What are the advantages / disadvantages of each coding style shown above?**
- ❖ **How Synthesis tool will give result for above codes?**
- ❖ **What happens if default statement is removed in case statement?**
- ❖ **What happens if combination 11 and default statement is removed? (Hint Latch inference)**

2.1.20 What is the difference between $display and $monitor and $write and $strobe?

These commands have the same syntax, and display text on the screen during simulation.$display and $strobe display once every time they are executed, whereas $monitor displays every time one of its parameters changes.

$display and $write two are the same except that $display always prints a newline character at the end of its execution.

The difference between $display and $strobe is that $strobe displays the parameters at the very end of the current simulation time unit rather than exactly when it is executed.

$strobe. This task is very similar to the $display task except for a slight difference. If many other statements are executed in the same time unit as the $display task, the order in which the statements and the $display task are executed is nondeterministic. If $strobe is used, it is always executed after all other assignment statements in the same time unit have executed. Thus, $strobe provides a synchronization mechanism to

ensure that data is displayed only after all other assignment statements, which change the data in that time step, have executed.

```
//strobing
always @(posedge clock)
begin
a = b;
c = d;
always @(posedge clock)
$strobe ("Displaying a = %b, c = %b" , a, c) ; / / display values at posedge
```

The values at positive edge of clock will be displayed only after statements a = b and c = d execute. If $display was used, $display might execute before statements a = b and c = d, thus displaying different values.

Append b, h, o to the task name to change default format to binary, octal or hexadecimal.

syntax

- $display ("format_string", par_1, par_2, ...);
- $strobe ("format_string", par_1, par_2, ...);
- $monitor ("format_string", par_1, par_2, ...);
- $displayb (as above but defaults to binary..);
- $strobeh (as above but defaults to hex..);

- $monitoro (as above but defaults to octal..);

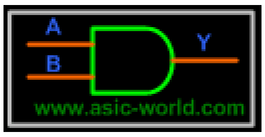
2.1.21 What is the difference between wire and reg?

Wire

Wire is used for designing combinational logic, as we all know that this kind of logic can not store a value. As you can see from the example above, a wire can be assigned a value by an assign statement. Default data type is wire: this means that if you declare a variable without specifying reg or wire, it will be a 1-bit wide wire.

```
module wire_example( a, b, y);
  input a, b;
  output y;

  wire a, b, y;

  assign y = a & b;

endmodule
```

SYNTHESIS OUTPUT



Register

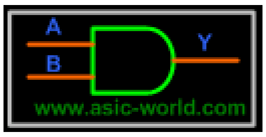
Reg can store value and drive strength. Something that we need to know about reg is that it can be used for modeling both combinational and sequential logic. Reg data type can be driven from initial and always block.

Reg data type as Combinational element

```
module reg_combo_example( a, b, y);
input a, b;
output y;
reg   y;
```

```
wire a, b;
always @ ( a or b)
begin
  y = a & b;
end
endmodule
```

SYNTHESIS OUTPUT

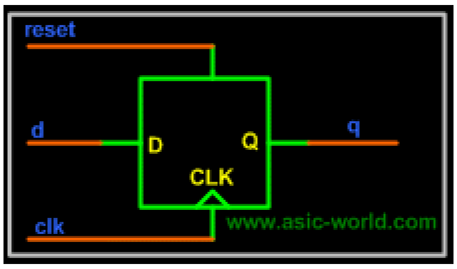


This gives the same output as that of the assign statement, with the only difference that y is declared as reg. There are distinct advantages to have reg modeled as combinational element; reg type is useful when a "case" statement is required.

Reg data type as Sequential element

```
module reg_seq_example( clk, reset, d, q);
input clk, reset, d;
output q;

reg   q;
wire clk, reset, d;

always @ (posedge clk or posedge reset)
if (reset) begin
  q <= 1'b0;
end else begin
  q <= d;
end

endmodule
```

SYNTHESIS OUTPUT



There is a difference in the way of assigning to reg when modeling combinational logic: in this logic we use blocking assignments while modeling sequential logic we use nonblocking ones.

2.1.22 What is the significance Timescale directive?

Defines the time units and simulation precision (smallest increment).

Syntax

`timescale TimeUnit / PrecisionUnit

TimeUnit = specifies the unit of measurement for times and delays.

PrecisionUnit = specifies the precision to which the delays are rounded off during simulation

Time = {either} 1 10 100

Unit = {either} s ms us ns ps fs

Rules

- ❖ The `timescale directive, like all compiler directives, affects all modules compiled after the directive, whether in the same file, or in files that are compiled separately, until the next `timescale or a `resetall directive.
- ❖ The precision unit must be less than or equal to the time unit.
- ❖ The precision for a simulation run is the smallest of all the precision units in `timescale directives. All delays are rounded to the nearest precision unit.

The $time task reports the simulation time in terms of the reference time unit for the module in which it is invoked.

Tips

Include a `timescale directive at the top of every module, even if there are no delays in the module, because some simulators may require this.

`timescale directives can be given before each module to setup the timings for that module, and remain in force until overridden by the next such directive.

Example

`timescale 10ns / 1ps Indicates delays are in 10 nanosecond units with 3 decimal points of precision (1 ps is 1/1000ns which is .001 ns).

`timescale 1 ns / 10 ps. Indicates delays are in 1 nanosecond units with 2 decimal points of precision (10 ps is 1/100 ps is .01 ns).

2.1.23 Parameter vs `define

- Parameter only for "per instance" constants
- `define for "global" constants.

2.1.24 For what is defparam used?

However, during compilation of Verilog modules, parameter values can be altered separately for each module instance. This allows us to pass a distinct set of parameter values to each module during compilation regardless of predefined parameter values.

There are two ways to override parameter values:

- Through the defparam statement
- Through module instance parameter value assignment.

Using defparam:

Parameter values can be changed in any module instance in the design with the keyword defparam. The hierarchical name of the module instance can be used to override parameter values.

```
//Define a module hello-world
module hello-world;
parameter id-num = 0; //define a module identification number = 0
initial //display the module identification number
Sdisplay("Disp1aying hello-world id number = %dM, id-num);
endmodule
```

```
//define top-level module
module top;
//change parameter values in the instantiated modules
//Use defparam statement
defparam wl.id-num = 1, w2.id-num = 2;
//instantiate two hello-world modules
hello-world wl0;
hello-world w2 ( ) ;
endmodule
```

the module hello-world was defined with a default id-num = 0.However, when the module instances wl and w2 of the type hello-world are created, their id-num values are modified with the defparam statement.

If we simulate the above design, we would get the following output:

```
Displaying hello-world id number = 1
Displaying hello-world id number = 2
```

Multiple defparam statements can appear in a module. Any parameter can be overridden with the defparam statement.

Using Module-Instance Parameter:

Parameter values can be overridden when a module is instantiated. The new parameter values are passed during module instantiation. The top-level module can pass parameters.

```
//define top-level module
module top;
//instantiate two hello-world modules; pass new parameter values
hello-world #(l) wl; //pass value 1 to module wl
hello-world #(2) w2; //pass value 2 to module w2
endmodule
```

If multiple parameters are defined in the module, during module instantiation they can be overridden by specifying the new values in the same order as the parameter declarations in the module. If an overriding value is not specified, the default parameter declaration values are taken.

```
//define module with delays
```

```
module bus-master;
parameter delayl = 2;
parameter delay2 = 3;
parameter delay3 = 7;
...
<module internals>
...
endmodule


//top-level module; instantiates two bus-master modules
module top;
//Instantiate the modules with new delay values
bus-master # (4, 5, 6) bl ( ) ; //bl : delayl = 4, delay2 = 5, delay3 = 6
bus-master # (9,4) b2 ( ) ; //b2: delayl = 9, delay2 = 4, delay3 = 7 (default)
endmodule
```

Module-instance, parameter value assignment is a very useful method used to override parameter values and to customize module instances.

2.1.25 What are the pros and cons of specifying the parameters using the defparam construct vs. specifying during instantiation?

The advantages of specifying parameters during instantiation method are:

- All the values to all the parameters don't need to be specified. Only those parameters that are assigned the new values need to be specified. The unspecified parameters will retain their default values specified within its module definition.
- The order of specifying the parameter is not relevant anymore, since the parameters are directly specified and linked by their name.

The disadvantage of specifying parameter during instantiation are:

- This has a lower precedence when compared to assigning using defparam.

The advantages of specifying parameter assignments using defparam are:

- This method always has precedence over specifying parameters during instantiation.

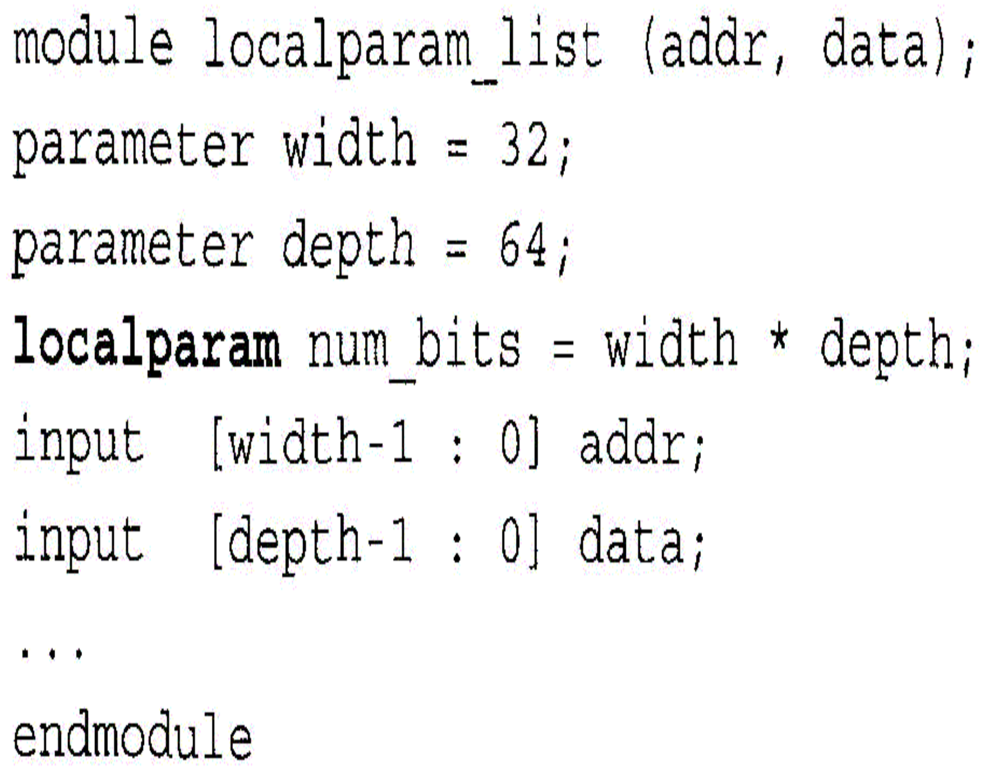
- All the parameter value override assignments can be grouped inside one module and together in one place, typically in the top-level testbench itself.
- When multiple defparams for a single parameter are specified, the parameter takes the value of the last defparam statement encountered in the source if, and only if, the multiple defparam's are in the same file. If there are defparam's in different files that override the same parameter, the final value of the parameter is indeterminate.

The disadvantages of specifying parameter assignments using defparam are:

- The parameter is typically specified by the scope of the hierarchies underneath which it exists. If a particular module gets ungrouped in its hierarchy, [sometimes necessary during synthesis], then the scope to specify the parameter is lost, and is unspecified. B
- For example, if a module is instantiated in a simulation testbench, and its internal parameters are then overridden using hierarchical defparam constructs (For example, defparam U1.U_fifo.width = 32;). Later, when this module is synthesized, the internal hierarchy within U1 may no longer exist in the gate-level netlist, depending upon the synthesis strategy chosen. Therefore post-synthesis simulation will fail on the hierarchical defparam override.

2.1.26 How do I prevent selected parameters of a module from being overridden during instantiation?

If a particular parameter within a module should be prevented from being overridden, then it should be declared using the localparam construct, rather than the parameter construct. The localparam construct has been introduced from Verilog-2001. Note that a localparam variable is fully identical to being defined as a parameter, too. In the following example, the localparam construct is used to specify num_bits, and hence trying to override it directly gives an error message.



```
module localparam_list (addr, data);
parameter width = 32;
parameter depth = 64;
localparam num_bits = width * depth;
input  [width-1 : 0] addr;
input  [depth-1 : 0] data;
...
endmodule
```

Note, however, that, since the width and depth are specified using the parameter construct, they can be overridden during instantiation or using defparam, and hence will indirectly override the num_bits values. In general, localparam constructs are useful in defining new and localized identifiers whose values are derived from regular parameters.

2.1.27 What happens to the logic after synthesis, that is driving an unconnected output port that is left open (, that is, noconnect) during its module instantiation?

An unconnected output port in simulation will drive a value, but this value does not propagate to any other logic. In synthesis, the cone of any combinatorial logic that drives the unconnected output will get optimized away during boundary optimisation, that is, optimization by synthesis tools across hierarchical boundaries.

2.1.28 How is the connectivity established in Verilog when connecting wires of different widths?

When connecting wires or ports of different widths, the connections are right-justified, Starts from the LSB that is, the rightmost bit on the RHS gets connected to the rightmost bit of the LHS and so on, until the MSB of either of the net is reached.

2.1.29 Can I use a Verilog function to define the width of a multi-bit port, wire, or reg type?

The width elements of ports, wire or reg declarations require a constant in both MSB and LSB. Before Verilog 2001, it is a syntax error to specify a function call to evaluate the value of these widths. For example, the following code is erroneous before Verilog 2001 version.

```
reg [ get_high(val1:vla2) : get_low (val3:val4)] reg1;
```

In the above example, get_high and get_low are both function calls of evaluating a constant result for MSB and LSB respectively. However, Verilog-2001 allows the use of a function call to evaluate the MSB or LSB of a width declaration

2.1.30 How can I pass parameters to my simulation?

A test bench and simulation will likely need many different parameters and settings for different sorts of tests and conditions. It is definitely a good idea to concentrate on a single testbench file that is parameterized, rather than create a dozen seperate, yet nearly identical, testbenches. Here are 3 common techniques:

Use a define.

This is almost exactly the same approach as the #define and -D compiler arg that C programs use. In your Verilog code, use a `define to define the variable condition and then use the Verilog preprocessor directives like `ifdef. Use the '+define+' Verilog command line option. For example:

```
... to run the simulation ..
verilog testbench.v cpu.v +define+USEWCSDF
... in your code ...
`ifdef USEWCSDF
initial $sdf_annotate (testbench.cpu, "cpuwc.sdf");
`endif
```

The +define+ can also be filled in from your Makefile invocation, which in turn, can be finally filled in the your UNIX prompt command line.

Defines are a blunt weapon because they are very global and you can only do so much with them since they are a pre-processor trick. Consider the next approach before resorting to defines.

Use parameters and parameter definition modules.

Parameters are not preprocessor definitions and they have scope (e.g. parameters are associated with specific modules). Parameters are therefore more clean, and if you are in the habit of using a lot of defines; consider switching to parameters. As an example, let’s say we have a test (e.g. test12) which needs many parameters to have particular settings. In your code, you might have this sort of stuff:

```
module testbench_uart1 (....)
```

```
parameter BAUDRATE = 9600;
...
if (BAUDRATE > 9600) begin
... E.g. use the parameter in your code like you might any general variable
... BAUDRATE is completely local to this module and this instance. You might
... have the same parameters in 3 other UART instances and they'd all be different
... values...
```

Now, your test12 has all kinds of settings required for it. Let's define a special module called testparams which specifies all these settings. It will itself be a module instantiated under the testbench:

```
module testparams;
defparam testbench.cpu.uart1.BAUDRATE = 19200;
defparam testbench.cpu.uart2.BAUDRATE = 9600;
defparam testbench.cpu.uart3.BAUDRATE = 9600;
defparam testbench.clockrate CLOCKRATE = 200; // Period in ns.
... etc ...
endmodule
```

The above module always has the same module name, but you would have many different filenames; one for each test. So, the above would be kept in test12_params.v. Your Makefile includes the appropriate params file given the desired make target. (BTW: You may run across this sort of technique by ASIC vendors who might have a module containing parameters for a memory model, or you might see this used to collect together a large number of system calls that turn off timing or warnings on particular troublesome nets, etc.)

Use memory blocks.

Not as common a technique, but something to consider. Since Verilog has a very convenient syntax for declaring and loading memories, you can store your input data in a hex file and use $readmemh to read all the data in at once.

In your testbench:

```
module testbench;
...
reg [31:0] control[0:1023];
...
initial $readmemh ("control.hex", control);
...
endmodule
```

You could vary the filename using the previous techniques. The control.hex file is just a file of hex values for the parameters. Luckily, $readmemh allows embedded comments, so you can keep the file very readable:

A000 // Starting address to put boot code in

10 // Activate all ten input pulse sources

... etc...

Obviously, you are limitied to actual hex values with this approach. Note, of course, that you are free to mix and match all of these techniques!

2.1.31 How can I model a bi-directional net with assignments influencing both source and destination?

The assign statement constitutes a continuous assignment. The changes on the RHS of the statement immediately reflect on the LHS net. However, any changes on the LHS don't get reflected on the RHS. For example, in the following statement, changes to the rhs net will update the lhs net, but not vice versa.

wire rhs , lhs;

assign lhs=rhs;

System Verilog has introduced a keyword alias, which can be used only on nets to have a two-way assignment. For example, in the following code, any changes to the rhs is reflected to the lh s , and vice versa.

module test ();

wire rhs,lhs;

alias lhs=rhs;

In the above example, any change to either side of the net gets reflected on the other side.

2.1.32 Differences between inter statement and intra statement delay?

//define register variables

reg a, b, c;

//intra assignment delays

initial

begin

a = 0; c = 0;

b = #5 a + c; //Take value of a and c at the time=0, evaluate

//a + c and then wait 5 time units to assign value to b.

end

```
//Equivalent method with temporary variables and regular delay control
initial
begin
a = 0; c = 0;
temp_ac = a + c;
#5 b = temp_ac; //Take value of a + c at the current time and
//store it in a temporary variable. Even though a and c
//might change between 0 and 5,
//the value assigned to b at time 5 is unaffected.
end
```

2.1.33 What is the difference between the following two lines of Verilog code?

#5 a = b; a = #5 b;

#5 a = b; Wait five time units before doing the action for "a = b;". The value assigned to a will be the value of b 5 time units hence.

a = #5 b; The value of b is calculated and stored in an internal temp register. After five time units, assign this stored value to a.

2.1.34 In a pure combinational circuit is it necessary to mention all the inputs in sensitivity disk? if yes, why?

Yes in a pure combinational circuit is it necessary to mention all the inputs in sensitivity list other wise it will result in pre and post synthesis mismatch.

2.1.35 What is pli? why is it used?

Programming Language Interface (PLI) of Verilog HDL is a mechanism to interface Verilog programs with programs written in C language. It also provides mechanism to access internal databases of the simulator from the C program.

PLI is used for implementing system calls which would have been hard to do otherwise (or impossible) using Verilog syntax. Or, in other words, you can take advantage of both the paradigms - parallel and hardware related features of Verilog and sequential flow of C - using PLI.

2.1.36 What is difference between freeze deposit and force?

$deposit (variable, value);

This system task sets a Verilog register or net to the specified value. Variable is the register or net to be changed; value is the new value for the register or net. The value remains until there is a subsequent driver transaction or another $deposit task for the same register or net. This system task operates identically to the ModelSim force -deposit command.

The force command has -freeze, -drive, and -deposit options. When none of these is specified, then -freeze is assumed for unresolved signals and -drive is assumed for resolved signals. This is designed to provide compatibility with force files. But if you prefer –freeze as the default for both resolved and unresolved signals.

2.1.37 Will case infer priority register if yes how give an example?

Yes case can infer priority register depending on coding style

```
reg r;
// Priority encoded mux,
always @ (a or b or c or select2)
begin
r = c;
case (select2)
2'b00: r = a;
2'b01: r = b;
endcase
end
```

2.1.38 Casex,z difference,which is preferable,why?

CASEZ :

Special version of the case statement which uses a Z logic value to represent don't-care bits.

CASEX :

Special version of the case statement which uses Z or X logic values to represent don't-care bits.

CASEZ should be used for case statements with wildcard don’t cares, otherwise use of CASE is required; CASEX should never be used.

This is because:

Don’t cares are not allowed in the "case" statement. Therefore casex or casez are required. Casex will automatically match any x or z with anything in the case statement. Casez will only match z’s -- x’s require an absolute match.

Case

case Complete bitwise match between expression and case item expression

casez allows for z values to be treated as don't cares.

casex allows for both z and x to be treated as don't cares.

Optional: Use ? to specify don’t-care bits.

Expression or case_item	**case**	**casex**	**casez**
0	0	0	0
1	1	1	1
x	X	0 1 x z	x
z	Z	0 1 x z	0 1 x z
?	NA	NA	0 1 x z
Default	0 1 x z	0 1 x z	0 1 x z

2.1.39 Explain the differences and advantages of casex and casez over the case statement?

The casex operator has to be used when both the high impedance value (z) and unknown (x) in any bit has to be treated as a don’t-care during case comparisons. The casez operator treats the (z) operator as a don’t-care during case comparisons.

In both cases, the bits which that are treated as don’t-care will not be considered for comparison, that is, only bit values other than don’t care bits are used in the comparison. The wildcard character “?” can be used in place of “z” for literal numbers.

The following is an example of a casex statement

```
input [2:0] in1;
reg [2:0] reg1;
casex (in1)
  3'b0x0 : out1 = a & b;  // same as conditions
                          // 3'b010, 3'b000
  3'bx10 : out1 = a | b;  // same as conditions
                          // 3'b110, 3'b010
  default : out1 = a ^ b; // for all other
                          // conditions
endcase
```

The same example, if written with an if-else tree, would look like:

```
// bit in1[1] is not considered at all
  if (!in1[2] & !in1[0]) out1 = (a & b);
// bit in1[2] is not considered
  else if (in1[1] & !in1[0]) out1 = (a | b);
// default clause
else out1 = (a ^ b);
```

Using casex or casez has the following coding advantages:

- ❖ It reduces the number of lines, especially if the number of bits had been more
- ❖ Makes code look more clear and less cluttered
- ❖ Simplifies the optimization, as it is clear that the bits with x are to be ignored.

2.1.40 What is the difference between === and == ?

output of "==" can be 1, 0 or X.

output of "===" can only be 0 or 1.

When you are comparing 2 nos using "==" and if one/both the numbers have one or more bits as "x" then the output would be "X" . But if use "===" outpout would be 0 or 1.

e.g A = 3'b1x0

B = 3'b10x

A == B will give X as output.

A === B will give 0 as output.

"==" is used for comparison of only 1's and 0's .It can't compare Xs. If any bit of the input is X output will be X

"===" is used for comparison of X also.

2.1.41 How to generate sine wav using verilog coding style?

The easiest and efficient way to generate sine wave is using CORDIC Algorithm.

2.1.42 What is the difference between wire and reg?

Net types: (wire,tri)Physical connection between structural elements. Value assigned by a continuous assignment or a gate output. Register type: (reg, integer, time, real, real time) represents abstract data storage element. Assigned values only within an always statement or an initial statement. The main difference between wire and reg is wire cannot hold (store) the value when there no connection between a and b like a->b, if there is no connection in a and b, wire loose value. But reg can hold the value even if there in no connection. Default values:wire is Z,reg is x.

2.1.43 what is verilog case (1) ?

```
wire [3:0] x;
always @(...) begin
case (1'b1)
x[0]: SOMETHING1;
x[1]: SOMETHING2;
x[2]: SOMETHING3;
x[3]: SOMETHING4;
endcase
end
```

The case statement walks down the list of cases and executes the first one that matches. So here, if the lowest 1-bit of x is bit 2, then something3 is the statement that will get executed (or selected by the logic).

2.1.44 Why is it that "if (2'b01 & 2'b10)..." doesn't run the true case?

This is a popular coding error. You used the bit wise AND operator (&) where you meant to use the logical

2.1.45 what are Different types of Verilog Simulators?

There are mainly two types of simulators available.

- Event Driven
- Cycle Based

Event-based Simulator:

This Digital Logic Simulation method sacrifices performance for rich functionality: every active signal is calculated for every device it propagates through during a clock cycle. Full Event-based simulators support 4-28 states; simulation of Behavioral HDL, RTL HDL, gate, and transistor representations; full timing calculations for all devices; and the full HDL standard. Event-based simulators are like a Swiss Army knife with many different features but none are particularly fast.

Cycle Based Simulator:

This is a Digital Logic Simulation method that eliminates unnecessary calculations to achieve huge performance gains in verifying Boolean logic: Cycle based simulators work only with synchronous designs.

1) Results are only examined at the end of every clock cycle; and

2) The digital logic is the only part of the design simulated (no timing calculations). By limiting the calculations, Cycle based Simulators can provide huge increases in performance over conventional Event-based simulators.

Cycle based simulators are more like a high speed electric carving knife in comparison because they focus on a subset of the biggest problem: logic verification.

Cycle based simulators are almost invariably used along with Static Timing verifier to compensate for the lost timing information coverage.

2.1.46 What is Constrained-Random Verification ?

As ASIC and system-on-chip (SoC) designs continue to increase in size and complexity, there is an equal or greater increase in the size of the verification effort required to achieve functional coverage goals. This has created a trend in RTL verification techniques to employ constrained-random verification, which shifts the emphasis from hand-authored tests to utilization of compute resources. With the corresponding emergence of faster, more complex bus standards to handle the massive volume of data traffic there has also been a renewed significance for verification IP to speed the time taken to develop advanced testbench environments that include randomization of bus traffic.

Directed-Test Methodology

Building a directed verification environment with a comprehensive set of directed tests is extremely time-consuming and difficult. Since directed tests only cover conditions that have been anticipated by the verification team, they do a poor job of covering corner cases. This can lead to costly re-spins or, worse still, missed market windows.

Traditionally verification IP works in a directed-test environment by acting on specific testbench commands such as read, write or burst to generate transactions for whichever protocol is being tested. This directed traffic is used to verify that an interface behaves as expected in response to valid transactions and error conditions. The drawback is that, in this directed methodology, the task of writing the command code and checking the responses across the full breadth of a protocol is an overwhelming task. The verification team frequently runs out of time before a mandated tape-out date, leading to poorly tested interfaces. However, the bigger issue is that directed tests only test for predicted behavior and it is typically the unforeseen that trips up design teams and leads to extremely costly bugs found in silicon.

Constrained-Random Verification Methodology

The advent of constrained-random verification gives verification engineers an effective method to achieve coverage goals faster and also help find corner-case problems. It shifts the emphasis from writing an enormous number of directed tests to writing a smaller set of constrained-random scenarios that let the compute resources do the work. Coverage goals are achieved not by the sheer weight of manual labor required to hand-write directed tests but by the number of processors that can be utilized to run random seeds. This significantly reduces the time required to achieve the coverage goals.

Scoreboards are used to verify that data has successfully reached its destination, while monitors snoop the interfaces to provide coverage information. New or revised constraints focus verification on the uncovered parts of the design under test. As verification progresses, the simulation tool identifies the best seeds, which are then retained as regression tests to create a set of scenarios, constraints, and seeds that provide high coverage of the design.

2.1.47 Difference between blocking and non-blocking

The Verilog language has two forms of the procedural assignment statement: blocking and non-blocking. The two are distinguished by the = and <= assignment operators. The blocking assignment statement (= operator) acts much like in traditional programming languages. The whole statement is done before control passes on to the next statement. The non-blocking (<= operator) evaluates all the right-hand sides for the current time unit and assigns the left-hand sides at the end of the time unit. For example, the following Verilog program

```
// testing blocking and non-blocking assignment
module blocking;
reg [0:7] A, B;
initial begin: init1
A = 3;
#1 A = A + 1; // blocking procedural assignment
B = A + 1;
$display("Blocking: A= %b B= %b", A, B );
A = 3;
#1 A <= A + 1; // non-blocking procedural assignment
B <= A + 1;
#1 $display("Non-blocking: A= %b B= %b", A, B );
```

```
end
endmodule
```

produces the following output:

```
Blocking: A= 00000100 B= 00000101
Non-blocking: A= 00000100 B= 00000100
```

The effect is for all the non-blocking assignments to use the old values of the variables at the beginning of the current time unit and to assign the registers new values at the end of the current time unit. This reflects how register transfers occur in some hardware systems. blocking procedural assignment is used for combinational logic and non-blocking procedural assignment for sequential

2.1.48 How blocking and non blocking statements get executed?

Execution of blocking assignments can be viewed as a one-step process:

1. Evaluate the RHS (right-hand side equation) and update the LHS (left-hand side expression) of the blocking assignment without interruption from any other Verilog statement. A blocking assignment "blocks" trailing assignments in the same always block from occurring until after the current assignment has been completed

Execution of nonblocking assignments can be viewed as a two-step process:

1. Evaluate the RHS of nonblocking statements at the beginning of the time step.
2. Update the LHS of nonblocking statements at the end of the time step.

2.1.49 Write a verilog code to swap contents of two registers with and without a temporary register?

With temp reg :

```
always @ (posedge clock)
begin
temp=b;
b=a;
a=temp;
end
```

Without temp reg:

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

end

2.1.50 What is sensitivity list?

A list of signals that trigger execution of the block when they change value.

The sensitivity list indicates that when a change occurs to any one of elements in the list change, begin…end statement inside that always block will get executed.

2.1.51 In a pure combinational circuit is it necessary to mention all the inputs in sensitivity disk? if yes, why?

Yes in a pure combinational circuit is it necessary to mention all the inputs in sensitivity lisk other wise it will result in pre and post synthesis mismatch.

2.1.52 Tell me structure of Verilog code you follow?

A good template for your Verilog file is shown below.

```
// timescale directive tells the simulator the base units and precision of the simulation
`timescale 1 ns / 10 ps
module name (input and outputs);
// parameter declarations
parameter parameter_name = parameter value;
// Input output declarations
input in1;
input in2; // single bit inputs
output [msb:lsb] out; // a bus output
// internal signal register type declaration - register types (only assigned within always statements). reg register variable 1;
reg [msb:lsb] register variable 2;
// internal signal. net type declaration - (only assigned outside always statements) wire net variable 1;
// hierarchy - instantiating another module
reference name instance name (
.pin1 (net1),
.pin2 (net2),
.
.pinn (netn)
);
```

```
// synchronous procedures
always @ (posedge clock)
begin
.
end
// combinatinal procedures
always @ (signal1 or signal2 or signal3)
begin
.
end
assign net variable = combinational logic;
endmodule
```

2.1.53 Difference between Verilog and vhdl?

Compilation

VHDL. Multiple design-units (entity/architecture pairs), that reside in the same system file, may be separately compiled if so desired. However, it is good design practice to keep each design unit in it's own system file in which case separate compilation should not be an issue.

Verilog. The Verilog language is still rooted in it's native interpretative mode. Compilation is a means of speeding up simulation, but has not changed the original nature of the language. As a result care must be taken with both the compilation order of code written in a single file and the compilation order of multiple files. Simulation results can change by simply changing the order of compilation.

Data types

VHDL. A multitude of language or user defined data types can be used. This may mean dedicated conversion functions are needed to convert objects from one type to another. The choice of which data types to use should be considered wisely, especially enumerated (abstract) data types. This will make models easier to write, clearer to read and avoid unnecessary conversion functions that can clutter the code. VHDL may be preferred because it allows a multitude of language or user defined data types to be used.

Verilog. Compared to VHDL, Verilog data types a re very simple, easy to use and very much geared towards modeling hardware structure as opposed to abstract hardware modeling. Unlike VHDL, all data types used in a Verilog model are defined by the Verilog language and not by the user. There are net data types, for example wire, and a register data type called reg. A model with a signal whose type is one of the net data types has a corresponding electrical wire in the implied modeled circuit. Objects, that is signals, of type reg hold their value over simulation delta cycles and should not be confused with the modeling of a hardware register. Verilog may be preferred because of it's simplicity.

Design reusability

VHDL. Procedures and functions may be placed in a package so that they are avail able to any design-unit that wishes to use them.

Verilog. There is no concept of packages in Verilog. Functions and procedures used within a model must be defined in the module. To make functions and procedures generally accessible from different module statements the functions and procedures must be placed in a separate system file and included using the `include compiler directive.

2.1.54 Can you tell me some of system tasks and their purpose?

$display, $displayb, $displayh, $displayo, $write, $writeb, $writeh, $writeo.

The most useful of these is $display.This can be used for displaying strings, expression or values of variables.

Here are some examples of usage.

```
$display("Hello oni");

--- output: Hello oni

$display($time) // current simulation time.

--- output: 460

counter = 4'b10;

$display(" The count is %b", counter);

--- output: The count is 0010
```

$reset resets the simulation back to time 0; $stop halts the simulator and puts it in interactive mode where the user can enter commands

$finish exits the simulator back to the operating system

2.1.55 Can you list out some of enhancements in Verilog 2001?

In earlier version of Verilog, we use 'or' to specify more than one element in sensitivity list. In Verilog 2001, we can use comma as shown in the example below.

```
// Verilog 2k example for usage of comma

always @ (i1,i2,i3,i4)
```

Verilog 2001 allows us to use star in sensitive list instead of listing all the variables in RHS of combo logics. This removes typo mistakes and thus avoids simulation and synthesis mismatches, Verilog 2001 allows port direction and data type in the port list of modules as shown in the example below

```
module memory (
input r,
input wr, input [7:0] data_in,
input [3:0] addr,
output [7:0] data_out
);
```

The width elements of ports, wire or reg declarations require a constant in both MSB and LSB. Before Verilog 2001, it is a syntax error to specify a function call to evaluate the value of these widths. For example, the following code is erroneous before Verilog 2001 version.

```
reg [ get_high(val1:vla2) : get_low (val3:val4)] reg1;
```

2.1.56 Write a Verilog code for synchronous and asynchronous reset?

Synchronous reset, synchronous means clock dependent so reset must not be present in sensitivity disk eg:

```
always @ (posedge clk )
begin if (reset)
. . . end
```

Asynchronous means clock independent so reset must be present in sensitivity list.

Eg

```
Always @(posedge clock or posedge reset)
begin
if (reset)
. . . end
```

2.1.57 Can you list out some of synthesizable and non-synthesizable constructs?

not synthesizable->>>>

initial

ignored for synthesis.

delays

ignored for synthesis.

events

not supported.

real

Real data type not supported.

time

Time data type not supported.

force and release

Force and release of data types not supported.

fork join

Use nonblocking assignments to get same effect.

user defined primitives

Only gate level primitives are supported.

synthesizable constructs->>

assign,for loop,Gate Level Primitives,repeat with constant value...

2.1.58 What is meant by inferring latches, how to avoid it?

Consider the following:

```
always @(s1 or s0 or i0 or i1 or i2 or i3)
case ({s1, s0})
2'd0: out = i0;
2'd1: out = i1;
2'd2: out = i2;
endcase
```

In a case statement if all the possible combinations are not compared and default is also not specified like in example above a latch will be inferred, a latch is inferred because to reproduce the previous value when unknown branch is specified.

For example in above case if {s1, s0}=3, the previous stored value is reproduced for this storing a latch is inferred.

The same may be observed in IF statement in case an ELSE IF is not specified.

To avoid inferring latches make sure that all the cases are mentioned if not default condition is provided.

2.1.59 What logic is inferred when there are multiple assign statements targeting the same wire?

It is illegal to specify multiple assign statements to the same wire in a synthesizable code that will become an output port of the module. The synthesis tools give a syntax error that a net is being driven by more than one source.

Illegal Code

```
Wire temp;
Assign temp = in1 & in2;
Assign temp = in3 & in4;
```

However, it is legal to drive a three-state wire by multiple assign statements.

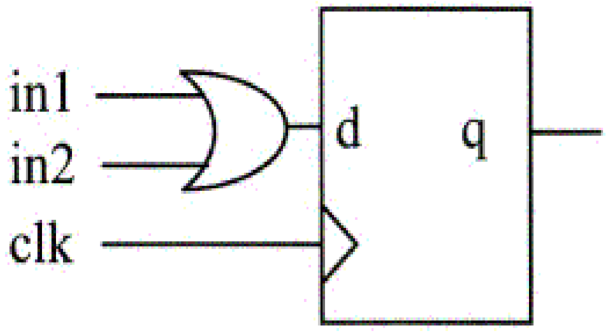
Legal code

```
Wire temp;
Assign temp = enable1? (In1 & in2): 1'bz;
Assign temp = enable2? (In3 & in4): 1'bz
```

2.1.60 What value is inferred when multiple procedural assignments made to the same reg variable in an always block?

When there are multiple nonblocking assignments made to the same reg variable in a sequential always block, then the last assignment is picked up for logic synthesis. For example

```
always @ (posedge clk) begin
out <= in1^in2;
out <= in1 &in2;
out <= in1|in2;
```



In the example just shown, it is the OR logic that is the last assignment. Hence, the logic synthesized was indeed the OR gate. Had the last assignment been the "&" operator, it would have synthesized an AND gate.

2.1.61 What are different types of timing verifications?

Dynamic timing:

a. The design is simulated in full timing mode.

b. Not all possibilities tested, as it is dependent on the input test vectors.

c. Simulations in full timing mode are slow and require a lot of memory.

d. Best method to check asynchronous interfaces or interfaces between different timing domains.

Static timing:

a. The delays over all paths are added up.

b. All possibilities, including false paths, verified without the need for test vectors.

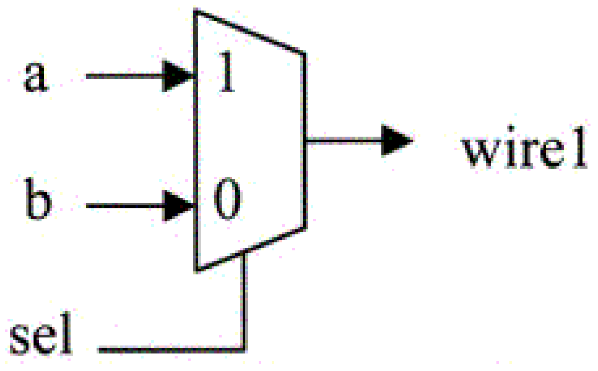
c. Faster than simulations, hours as opposed to days.

d. Not good with asynchronous interfaces or interfaces between different timing domains.

2.1.62 What do conditional assignments get inferred into?

Conditionals in a continuous assignment are specified through the "?:" operator. Conditionals get inferred into a multiplexor. For example, the following is the code for a simple multiplexor

```
assign wire1 = (sel==1'b1)? a : b;
```



2.1.63 Difference between the fork –join and begin-end

The fork - join keywords:

Groups several statements together.Cause the statements to be evaluated in parallel (all at the same time).

-> Timing within parallel group is absolute to the beginning of the group.

-> Block finishes after the last statement completes (Statement with highest delay, it can be the first statement in the block).

Example - "fork-join"

```
module initial_fork_join();
reg clk,reset,enable,data;
initial begin
 $monitor("%g clk=%b reset=%b enable=%b data=%b",
   $time, clk, reset, enable, data);
 fork
    #1   clk = 0;
    #10 reset = 0;
    #5   enable = 0;
    #3   data = 0;
 join
  #1  $display ("%g Terminating simulation", $time);
 $finish;
end
endmodule
```

Fork : clk gets its value after 1 time unit, reset after 10 time units, enable after 5 time units, data after 3 time units. All the statements are executed in parallel.

Simulator Output

```
 0 clk=x reset=x enable=x data=x
 1 clk=0 reset=x enable=x data=x
 3 clk=0 reset=x enable=x data=0
 5 clk=0 reset=x enable=0 data=0
 10 clk=0 reset=0 enable=0 data=0
```

11 Terminating simulations

The begin - end keywords:

Group several statements together. Cause the statements to be evaluated sequentially (one at a time)

-> Any timing within the sequential groups is relative to the previous statement.

-> Delays in the sequence accumulate (each delay is added to the previous delay)

-> Block finishes after the last statement in the block.

Example – begin end

```
module sequential();
reg a;
  initial begin
    $monitor ("%g a = %b", $time, a);
   #10  a = 0;
   #11  a = 1;
   #12  a = 0;
    #13  a = 1;
    #14  $finish;
 end
 endmodule
```

Simulator Output

```
0 a = x
10 a = 0
21 a = 1
33 a = 0
46 a = 1
```

2.1.64 Difference between the testing and verification.

Verification proves conformance with a specification.

Testing tries to find cases where the system does not meet its specification.

2.1.65 What is the difference between the specparam and parameter constructs?

The specparam is a special kind of parameter that is intended to specify only timing and the delay values. The key differences in using the specparam and the parameter constructs are:

Specparam	Parameter
Can be defined within both module and specify block.	Must be defined outside of the specify block and with in module.
A specparam can be assigned using another specparam or parameter or combination of both.	Parameter cannot be assigned the value of a specparam.
Value overridden using SDF annotaion.	Can be overridden during the instantiation or using defparam.

2.1.66 What logic gets synthesized when I use an integer instead of a reg variable as a storage element? Is use of integer recommended?

An integer can take the place of a reg as a storage element. An example to illustrate this is as follows:

```
module int_insteadof_reg (in1, clk, reset, out1);
input  [3:0] in1;
input  clk, reset;
output [3:0] out1;

integer int_tmp;
// reg [3:0] int_tmp;   // Normally we use this reg
// declaration

always @(posedge clk or negedge reset)
begin
  if (!reset)
    int_tmp <= 0;
  else
    int_tmp <= in1;
end

assign out1 = int_tmp;

endmodule
```

In this example, the variable int_tmp is defined as an integer, instead of the reg that it would normally be (the reg declaration is commented in the example for illustration). Note that, although the default width of the integer declaration is 32 bits, the final result of the int_tmp registers synthesis yield is only 4 bits. This is because the optimiser in the synthesis tool removes the unnecessary higher order bits, in order to minimize the area.

Although the use of integer as shown above is a legal construct, it is not recommended for the synthesis of storage elements.

2.1.67 How do I choose between a case statement and a multi-way if-else statement?

Both case and if-else are flow control constructs. Functionally in simulation they yield similar results. While both these constructs get elaborated into combinatorial logic, the usage scenarios for these constructs are different.

A case statement is typically chosen for the following scenarios:

- ❖ When the conditionals are mutually exclusive and only one variable controls the flow in the case statement. The case variable itself could be a concatenation of different signals.
- ❖ To specify the various state transitions of a finite state machine
- ❖ Use of casex and casez allows use of x and z to represent don't-care bits in the control expression

A multi way if statement is typically chosen in the following scenarios:

- ❖ Synthesizing priority encoded logic
- ❖ When the conditionals are not mutually exclusive and more general in using multiple expressions for the condition expression.

The advantages of using the case over if-else is as follows:

- ❖ Case statements are more readable than if-else.
- ❖ When used for state machines, there is a direct mapping between the state machine's "bubble diagram" and the case description.

In a case construct, if all the possible cases are not specified, and the default clause is missing, a latch is inferred. Likewise, for an if-else construct, if a final else clause is missing, a latch is inferred.

2.1.68 What is the difference between full_case and parallel_case synthesis directive?

A "full" case statement is a case statement in which all possible case-expression binary patterns can be matched to a case item or to a case default. If a case statement does not include a case default and if it is possible to find a binary case expression that does not match any of the defined case items, the case statement is not "full."

A "parallel" case statement is a case statement in which it is only possible to match a case expression to one and only one case item. If it is possible to find a case expression that would match more than one case item, the matching case items are called "overlapping" case items and the case statement is not "parallel."

The difference between full case and parallel case synthesis directives is summarized in the table below:

Full case	Parallel case
Indicates the case statement has been fully specified and all unspecified case expressions can be optimized away.	Indicates all case items need to evaluated in parallel and not infer any priority encoding logic.
All control paths are specified explicitly or by using default.	There is no overlap among the case statements.
Helps avoids latches as all cases are fully specified.	Results in multiplexor logic as parallel logic.
Although not recommended, the default clause can be avoided, and still not infers a latch.	A priority encoder is not synthesized as each path is unique.

An example of case statement that is full(and parallel) shown below.
Case(sel)
2’b00: out = a;
2’b01: out = b;
2’b10: out = c;
2’b11: out = d;
endcase
The default clause was not required since all the case expressions are specified.
	An example of case statement that is parallel (not full) is shown below.
	Case(sel)
	2’b00: out = a;
	2’b01: out = b;
	2’b10: out = c;
	endcase
	The default clause was not specified but all the case expressions are uniqe. A latch will be inferred.

2.1.69 What are some reusable coding practices for RTL Design?

A reusable design mainly helps in reducing the design time of larger implementations using IPs. The following key points summarize the main considerations during the implementation phase:

- ❖ Register all the outputs of crucial design blocks. This will make the timing interface easy during system level integration
- ❖ If an IP is being developed in both Verilog and VHDL, try to use the constructs that will be translatable later into VHDL.
- ❖ Avoid snake paths, as it will make both debugging tedious and synthesis inefficient.
- ❖ Partition the design considering the clock domains and the functional goals.

- Follow lexical and naming conventions that are self-descriptive and facilitate future product maintenance.
- Avoid instantiation of technology specific gates
- Use parameters instead of hard-coded values in the design
- Avoid clocks and resets that are generated internal to the design
- Avoid glue logic during top level inter-module instantiations

2.1.70 What are "snake" paths, and why should they be avoided?

A snake path, as the name suggests is a path that traverses through a number of hierarchies, and may eventually return back to the same hierarchy from which it originated.

Snake paths must be avoided in a design for the following reasons:

- It will constitute a long timing path, and hence, be the surprise critical path when static timing analysis is done at the top level. It may not show up during the timing analysis of the unit level blocks if it is poorly constrained.
- The synthesis tools need to put more effort in characterizing the constraints of the path across the hierarchies, and the compile time can get higher.

Some tips that can be followed to avoid the snake paths are:

- Register the outputs of modules with different functional objectives.
- Partition the design functionally, to avoid long paths across different hierarchies.
- Keep checking for the presence of the snake paths by periodically running synthesis on the fully integrated RTL, even if it is not fully verified functionally. This will give early feedback through the timing reports for the presence of a path traversing across multiple hierarchies.

2.1.71 What are a few considerations while partitioning large designs?

A large design needs to be approached in a hierarchical fashion. The following considerations need to be taken while partitioning these designs:

Functionality:

The functional grouping of the logic within a hierarchy is the prime criteria during partitioning the design. Typical partitioning of hierarchies are:

- Address and data paths: This module typically contains the address and data path registers, which drive the address and data buses of the primary outputs.
- Control logic: This module typically contains Finite StateMachines (FSMs), and the module gets the inputs for the FSMs, whose outputs drive the controls for the rest of the logic.

Clock domains:

In a multiple clock design, it is recommended to group the logic connected in the same clock domain in a single module. When signals need to interact with another module with a different clock, it is recommended to go through a synchronizer module, which takes in the input from the source clock domain and synchronizes it to the clock domain of the destination module.

Area:

Having too little logic in a module will create too many hierarchies, and too much logic within a single module will create issue of not being able to do fine tune control during floorplanning later during the backend process.

2.1.72 What will be synthesized of a module with only inputs and no outputs?

A module with only inputs and no outputs will synthesize into a module with no logic, since there is nothing to be synthesized as an output.

2.1.73 Why do I see latches in my synthesized logic?

There is more than one reason why latches could be seen in synthesized logic. This information is typically present in the elaboration log file of the synthesis tool.

- ❖ The if-else clause in the always block to which the latch is associated doesn't have a final else clause.
- ❖ The reg declaration of the variable doesn't have any value assigned upon entry to the combinatorial always block if the variable is used in an if statement without the else clause.
- ❖ There could be no default clause of a case construct that is not complete or the variables assigned within the case were not assigned a default value before entering the case statement.

2.1.74 What are "combinatorial timing loops"? Why should they be avoided?

Combinational timing loops are hardware loops in which the output of either a gate or a long combinatorial path is fed back as an input to the same gate or to another gate earlier in the combinatorial path. These paths are generally created unintentionally when a variable from one combinatorial block is used to drive a signal that is used in the same combinatorial block from which the variable was derived. This typically happens in large size combinatorial blocks, wherein it is difficult to visually track that a loop is getting created.

These combinatorial feedback loops are undesirable for the following reasons:

- ❖ Since there is no clock edge in between to break the path, the combinatorial loops will infinitely keep oscillating and triggering a square waveform, whose duty cycle is dependent upon the sum of ON delays and OFF delays across the combinatorial path.

For example, the following code is a combinatorial loop:

```
Assign out1 = out1 & in1
```

This will cause the out1 to feed in combinatorially back as one of the inputs.

- ❖ These loops cause a problem in testability, since they can inhibit the propagation of the logic forward.

Combinatorial loops can be caught quite early by one of the following means:

- ❖ Periodic use of linting tools throughout the development process. This is by far the best and easiest way to catch and fix loops early in the design cycle.
- ❖ During functional simulation, the desired output behavior doesn't appear in the output, or the simulation doesn't proceed ahead at all, because the simulator is hung.
- ❖ If the loop is undetected during simulation, many synthesis tools have suitable reporting commands, which detect the presence of a loop. Note that synthesis tools proceed with the static timing analysis by breaking the timing arc of the loop for critical path analysis.

2.1.75 Implement a 3:1 multiplexer in verilog

Three data inputs D0, D1 and D2

Two select inputs S0 and S1

One data output Y

The value of output Y is determined as follows:

Y = D0 if S0 = 0 and S1 = 0

Y = D1 if S0 = 1 and S1 = 0

Y = D2 if S1 = 1 (the value S0 doesn't matter)

- ❖ **Write a Verilog module for the 3:1 multiplexer that implements the sum-of-products equation for Y:**

Use a dataflow style for your code (ie, use "assign" to set the values of your signals).

```
module mux31_1(d0,d1,d2,s0,s1,y);
  input d0,d1,d2,s0,s1;
  output y;
  assign y = (~s0 & ~s1 & d0) | (s0 & ~s1 & d1) | (s1 & d2);
endmodule
```

- **Write a Verilog module for the 3:1 multiplexer that uses the "?" operator. Again use a dataflow style for your code (ie, use "assign" to set the values of your signals).**

```
module mux31_2(d0,d1,d2,s0,s1,y);
  input d0,d1,d2,s0,s1;
  output y;
  assign y = s1 ? d2 : (s0 ? d1 : d0);
endmodule
```

- **Write a Verilog module for the 3:1 multiplexer that uses the "case" statement. Remember that a "case" statement can only appear inside of a behavioral block.**

```
module mux31_3(d0,d1,d2,s0,s1,y);
  input d0,d1,d2,s0,s1;
  output y;
  // add reg declarations for variables assigned inside of
  // the behavioral block below
  reg y;
  always @ (d0 or d1 or d2 or s0 or s1)
  begin
    case ({s1,s0})
     2'b00: y <= d0;
     2'b01: y <= d1;
     2'b10, 2'b11: y <= d2;
    endcase
  end
endmodule
```

2.1.76 Behavioral block example

When creating a behavioral block using Verilog's ALWAYS statement, one supplies a sensitivity list -- a list of signals that trigger execution of the block when they change value. For example, here's full adder module which uses an ALWAYS block:

```
module fulladder(a,b,cin,sum,cout);
  input a,b,cin;
  output sum,cout;
  reg sum,cout;
  always @ (a or b or cin)
  begin
    sum <= a ^ b ^ cin;
    cout <= (a & b) | (a & cin) | (b & cin);
  end
endmodule
```

When writing behavioral implementations of combinational logic all the signals appearing on the right-hand side of assignment statements (ie, signals whose values are used in computations) should appear in the sensitivity list.

- **What would be the effect of, say, omitting cin from the sensitivity list above?**

Omitting cin from the sensitivity list would mean that during simulation changes in cin wouldn't trigger execution of the ALWAYS block, so the values of sum and cout wouldn't be recomputed to reflect the new value of cin.

The omission of a variable from the sensitivity list doesn't usually affect what logic is synthesized for the ALWAYS block. Of course, the logic gates will propagate changes in cin to sum and cout.

What's troublesome is that the simulation results are different from the results produced by the actual logic. This isn't so bad when the simulation results are discovered to be wrong and the source code gets fixed. It's much worse when the simulation gives the desired answer, but the synthesized logic (which may have already been delivered to a customer) does the wrong thing, ie, the incorrect simulation masked a bug in logic equations. Nowadays, many synthesizers give warnings when synthesizing combinational logic using values not declared in the sensitivity list of an ALWYS block.

2.1.77 Verilog parameter example

Using the Verilog parameter mechanism it's possible to write modules whose operation depends on parameters specified when the module is instantiated.

- **Write a parameterized parity module that takes as input a vector whose size is set by the parameter. The module has a single output which is 1 if the number of 1's in the input vector is odd and 0 otherwise.**

```
module parity(in,p);
  parameter N = 8;   // width of vector IN, default value is arbitrary
  input [N-1:0] in;
  output p;
  assign p = ^in;    // using XOR as a reduction operator
endmodule
```

❖ **How would one instantiate an instance of your module to compute parity on the 16-bit wide data bus DATA[15:0]?**

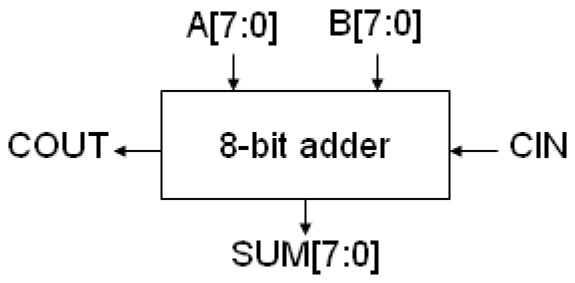
```
parity #(16) xxx(DATA[15:0],p);  // xxx is the name of this instance
```

Some Verilog synthesizers (eg, MAX+plus II, the one we use) don't allow the syntax used above for setting module parameters. One can also write

```
parity xxx(DATA[15:0],p);
defparam xxx.N=16;  // in MAX+plus II, this has to follow the instantiation of xxx
```

2.1.78 Combinational adder

Consider the following combinational circuit that adds to 8-bit numbers. The circuit includes carry-in (CIN) and carry-out (COUT) signals so that it can be cascaded to form wider adders.



❖ **Where possible, it's best to use the built-in Verilog operators for integer arithmetic. This makes it easy for the CAD tools to map your logic onto the special-purpose arithmetic circuitry provided by many of today's FPGAs. Write a Verilog module for the 8-bit adder that uses the Verilog "+" operator. Use a dataflow style for your code (ie, use "assign" to set the values of your signals).**

```
module adder8_1(a,b,cin,sum,cout);
  input [7:0] a,b;
  input cin;
  output [7:0] sum;
  output cout;
  // Verilog knows that adding two N-bit numbers produces a (N+1)-bit
  // result.  By concatenating cout and sum to form a 9-bit destination
  // for the add, we'll capture that high-order result bit.
  assign {cout,sum} = a + b + cin;
endmodule
```

- **A common method for building fast adders is to rewrite the equations for SUM and COUT in terms of P (propagate) and G (generate) signals. P is true if a carry-in will be propagated to the carry-out; G is true if a carry-out will be generated regardless of the value of the carry-in.**

P = A ^ B   (^ is the xor operator)

G = A*B

SUM = P ^ CIN

COUT = G + P*CIN

The advantage of this formulation is that P and G can be computed in parallel for all the bits of the adder since they don't depend on CIN. This will shorten the delay between CIN and COUT.

Write a Verilog module for the 8-bit adder that uses the P and G to compute SUM and COUT. Again use a dataflow style for your code.

```
module adder8_2(a,b,cin,sum,cout);
  input [7:0] a,b;
  input cin;
  output [7:0] sum;
  output cout;

  wire [7:0] p,g,c;   // P, G and COUT for each bit

  // all the lines below involve 8-bit wide operations
  assign p = a ^ b;
  assign g = a & b;
  assign c = g | (p & {c[6:0],cin});  // ripple carry!
  assign sum = p ^ {c[6:0],cin};
  assign cout = c[7];
endmodule
```

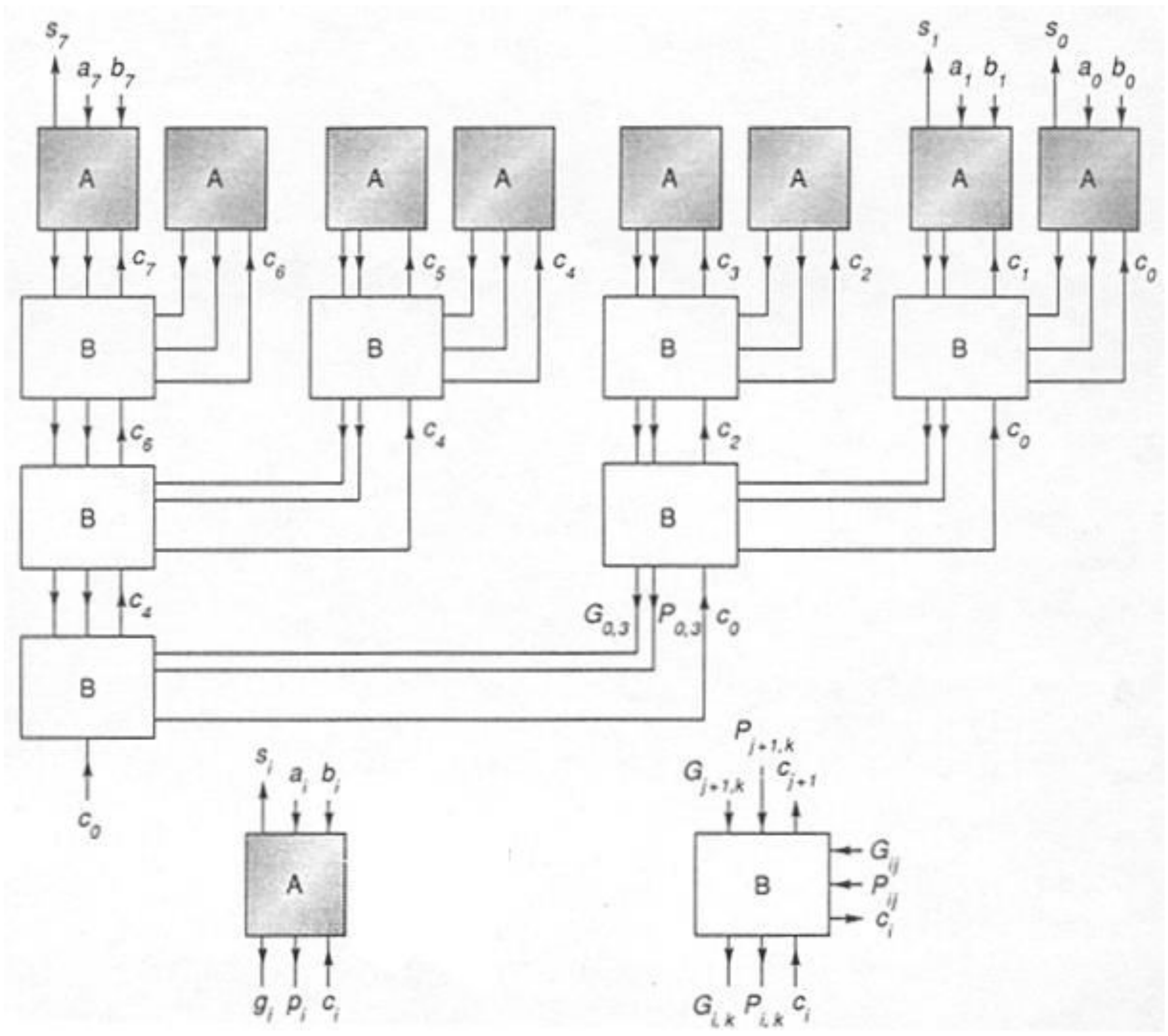
❖ **It can be hard to see how the carry chain is working in the previous example. Sometimes it's easier to express what you want if you use a loop and compute the answer bit-by-bit. Write a Verilog module for the 8-bit adder that uses P and G, but this time use a for-loop inside a behavioral block to compute SUM and COUT bit-by-bit.**

```
module adder8_3(a,b,cin,sum,cout);
  input [7:0] a,b;
  input cin;
  output [7:0] sum;
  output cout;

  // provide reg declarations for signals we'll be assigning
  // inside of the ALWAYS block below
  reg [7:0] sum;
  reg cout;
  reg c,p,g;
  integer i;    // index for for loop

  always @ (a or b or cin)     // we're sensitive to input changes
  begin
    c = cin;                  // carry-in to first stage
    // even though we appear to be reusing the c, p and g signals
    // each time through the loop, the synthesizer will realize it
    // needs to make local copies for each loop iteration.
    for (i = 0; i < 8; i = i + 1)
      begin
        p = a[i] ^ b[i];
        g = a[i] & b[i];
        sum[i] = p ^ c;
        c = g | (p & c);       // carry-in for next stage
      end
    cout = c;               // final carry out
  end
endmodule
```

In the adder implementations above the carry ripples up through the bits starting with the least significant bit. The total propagation delay (tPD) of the circuit is determined by the logic along the path between CIN and COUT (known as the carry chain). In a ripple carry adder tPD is proportional to the number of bits in the adder.

We can improve tPD by reworking the carry logic to use a "lookahead" strategy -- we'll use the P and G signals for individual nodes to compute P and G signals for pairs of nodes. We can apply this strategy again, computing P and G for groups of 4 nodes, and so on. We're building a tree of these lookahead modules; each node in the tree has the same logic. Once CIN for the adder is available it is fed to the top node of the lookahead tree, where it's combined with P and G of the children to produce carry-ins for the children nodes. This process continues until carry-in arrives at the leaf nodes where it can be used to compute SUM. For an N-bit adder, the depth of the lookahead tree is log2(N). It takes two traversals of the lookahead tree to (Ps and Gs up the tree, CINs down the tree) in order to finish the computation of SUM -- a tPD that's proportional to log2(N).

The following figure (from Hennessy and Patterson) shows what we have in mind.



❖ **Here are the logic equations for the outputs of the A module in the figure above:**

P = A ^ B

G = A * B

SUM = P ^ CIN

Write a Verilog implementation of the A module.

```
module A(a,b,cin,p,g,sum);
  input a,b,cin;
  output p,g,sum;

  assign p = a ^ b;
  assign g = a & b;
  assign sum = p ^ cin;
endmodule
```

- **Here are the logic equations for the outputs of the B module in the figure above.**

P1 and G1 are the P and G signals from the right-hand connections to the B block; P2 and G2 from the top connections.

POUT = P1*P2

GOUT = G2 + P2*G1

C1 = CIN

C2 = G1 + P1*CIN

Write a Verilog implementation of the B module.

```
module B(p1,g1,p2,g2,cin,pout,gout,c1,c2);
  input p1,g1,p2,g2,cin;
  output pout,gout,c1,c2;

  // just for variety try a behavioral description
  reg pout,gout,c1,c2;
  always @ (p1 or g1 or p2 or g2 or cin)
  begin
    pout <= p1 & p2;
    gout <= g2 | (p2 & g1);
    c1 <= cin;
    c2 <= g1 | (p1 & cin);
  end
endmodule
```

❖ **Using the A and B modules you created above, write a Verilog module for an 8-bit adder.**

```
module adder8_4(a,b,cin,sum,cout);
  input [7:0] a,b;
  input cin;
  output [7:0] sum;
  output cout;

  // 8 A modules
  wire [7:0] p,g,c;
  A a0(a[0],b[0],c[0],p[0],g[0],sum[0]);
  A a1(a[1],b[1],c[1],p[1],g[1],sum[1]);
  A a2(a[2],b[2],c[2],p[2],g[2],sum[2]);
  A a3(a[3],b[3],c[3],p[3],g[3],sum[3]);
  A a4(a[4],b[4],c[4],p[4],g[4],sum[4]);
  A a5(a[5],b[5],c[5],p[5],g[5],sum[5]);
  A a6(a[6],b[6],c[6],p[6],g[6],sum[6]);
  A a7(a[7],b[7],c[7],p[7],g[7],sum[7]);

  // first layer of B modules
  wire [3:0] pp,gg,cc;
  B b_0(p[0],g[0],p[1],g[1],cc[0],pp[0],gg[0],c[0],c[1]);
  B b_1(p[2],g[2],p[3],g[3],cc[1],pp[1],gg[1],c[2],c[3]);
  B b_2(p[4],g[4],p[5],g[5],cc[2],pp[2],gg[2],c[4],c[5]);
  B b_3(p[6],g[6],p[7],g[7],cc[3],pp[3],gg[3],c[6],c[7]);

  // second layer of B modules
  wire [1:0] ppp,ggg,ccc;
  B bb_0(pp[0],gg[0],pp[1],gg[1],ccc[0],ppp[0],ggg[0],cc[0],cc[1]);
  B bb_1(pp[2],gg[2],pp[3],gg[3],ccc[1],ppp[1],ggg[1],cc[2],cc[3]);

  // final layer of B modules
  wire pppp,gggg;
  B bbb(ppp[0],ggg[0],ppp[1],ggg[1],cin,pppp,gggg,ccc[0],ccc[1]);

  // compute final carry out
  assign cout = gggg + (pppp & cin);
endmodule
```

2.1.79 How many levels can be nested using `include ?

Ans: You can nest the`include compiler directive to at least 16 levels.

2.1.80 What is the use of $countdrivers?

Ans: The $countdrivers system function is provided to count the number of drivers on a specified net so that bus contention can be identified.

2.1.81 What is the use of $getpattern?

Ans:The system function $getpattern provides for fast processing of stimulus patterns that have to be propagated to a large number of scalar inputs.The function reads stimulus patterns that have been loaded into a memory using the $readmemb or $readmemh system tasks.

```
regi[1:in_width] in_mem[1:patterns];
integer index;
assign {i1,i2,i3,i4,i5,i6,i7,i8,i9,i10} = $getpattern(in mem[index]);
```

2.1.82 What is the reason that Verilog is usually considered better at low level modeling than VHDL? Why is VHDL usually considered better than Verilog for high level modeling?

Verilog has built-in types for gates and transistors, can also handle true bidirectional signals (VHDL has none of these things).

VHDL allows users to define their own data types which allows users to extend the language. Also, support for libraries and packages lends itself to more complex models.

2.1.83 What are the ways to create a race condition and how can these race conditions can be avoided?

The IEEE Verilog Standard defines which statements have a guaranteed order of execution and which statements don' t have a guaranteed order of execution.

A Verilog race condition occurs when two or more statements that are scheduled to execute in the same simulation time-step, would give different results when the order of statement execution is changed.

```
module race (out1, out2, clk, rst);
output out1, out2;
input clk, rst;
reg out1, out2;
always @(posedge clk or posedge rst)
if (rst) out1 = 0;
else out1 = out2;

always @(posedge clk or posedge rst)
if (rst) out2 = 1;
else out2 = out1;
endmodule
```

If the first always block executes first after a reset, both out1 and out2 will take on the value of 1. If the second always block executes first after a reset, both out1 and out2 will take on the value 0. This clearly represents a Verilog race condition.

Making multiple assignments to the same variable from more than one always block is a Verilog race condition, even when using nonblocking assignments.One of the recommendations is to avoid driving variables from multiple sources.

Illustrate example of how unintentional deadlocked situations can happen during simulation.

The deadlock situation is one in which one process is waiting for the other process to enable it, which in turn will enable the source process. The code could be a syntactically correct implementation, and still have a deadlock situation. The scenario can happen in both synchronous and asynchronous designs. A simple asynchronous example has been illustrated in the following, to demonstrate how deadlock occurs.

```
module deadlock;
reg reg1, reg2;

initial
begin
reg1 = 1'b0;
wait @ (reg2==1'b1)
end

always @(reg1)
begin
```

```
if (reg1==1'b1)
reg2 = 1'b1;
end
endmodule
```

The above example is an illustration of the deadlock scenario, which can be difficult to capture in a larger implementation.

2.1.84 What is the difference between a vectored and a scalared net?

Both scalared and vectored are Verilog constructs used on multi-bit nets to specify whether or not specifying bit and part select of the nets is permitted. For example,

```
wire scalared [3:0] a;
wire vectored [3:0] b;
wire c, d;
// Syntax error to use a bit select of vectored net
assign b[1] = 1'b1;
// OK
assign a[1] = 1'b0;
```

2.1.85 Difference b/w assign, de-assign and force, release.

The assign-deassign and force-release constructs in Verilog have similar effects, but differ in the fact that force-release can be applicable to nets, whereas assign-deassign is applicable only to registers.

The procedural assign-deassign construct is intended to be used for modeling hardware behavior, but the construct is not synthesizable by most logic synthesis tools. The force-release construct is intended for design verification, and is not synthesizable.

2.1.86 What does it mean to "short-circuit" the evaluation of an expression?

Verilog supports numerous operators that have rules of associativity and precedence. In some of the expressions, the result of the expression can be evaluated early on, due to the precedence and influence to override the rest of the expression. In that case, the entire expression need not be evaluated. This is called short-circuiting and expression evaluation.

For example,

```
assign out = ((a>b) & (c|d));
```

If the result of (a>b) is false (1'b0), then tools can already determine that the result of the AND operation will be 0. Thus, there is no need to evaluate (c|d) and rest of the equation is short-circuited.

assign out = (a>b) | (cld));

If the result of (a>b) is true (1'b0), then tools can already determine that the result of the OR operation will be 1. Thus, there is no need to evaluate (cld) and rest of the equation is short-circuited.

2.1.87 What are the pros and cons of using hierarchical names to refer to Verilog objects?

The top-level module is called the root module because it is not instantiated anywhere. It is the starting point. To assign a unique name to an identifier, start from the top-level module and trace the path along the design hierarchy to the desired identifier.

assign status = top.hub_top.hpie.status_reg;

Adv:

It is easy to debug the internal signals of a design, especially if they are not a part of the top level pin out.

Disadv:

- ❖ Sometimes, during synthesis, these hierarchical names get renamed, depending upon the synthesis strategy and switches used, and hence, will cease to exist. In that case, special switches need to be added to the synthesis compiler commands, in order to maintain the hierarchical naming.
- ❖ If the Verilog code needs to be translated into VHDL, the hierarchical names are not translatable.

2.1.88 Does Verilog support an "a to the power b" operator?

Yes. Verilog supports the operation by using two astrices, back to back like,

assign out = (in ** 5);

2.1.89 How to get copy of all the text that is printed to the standard output in a log file?

Ans:Using $log("filename");

2.1.90 What is the use of PATHPULSE$?

Ans: PATHPULSE$secparam is used to control pulse handling on a module path.

2.1.91 Difference between Reduction and Bitwise operators?

Ans: The difference is that bitwise operations are on bits from two different operands, where as reduction operations are on the bits of the same operand. Reduction operators works bit by bit from right to left.Reduction operators perform a bitwise operation on a single vector operand and yield a 1-bit result. Bitwise operators perform a bit-by-bit operation on two operands. They take each bit in one operand and perform the operation with the corresponding bit in the other operand.

2.1.92 How is Verilog implementation independent and why is this an advantage?

2.1.93 What level of Verilog is use in:

a.Test benches

b.Synthesized design

c.Net list

2.1.94 What is the difference between $fopen("filename"); and $fopen("filename","w");

Ans:If type is mitted,the file is opened for writing, and a multi channel descriptor mcd is returned.If type is supplied, the file is opened as specified by the value of type, and a file descriptor fd is returned. So in first statements , type is omitted and mcd is returned and in the second statement, fd is returned.

In the first statement, the file is opened for read and write.But in second statement,"w" is specified, so the file is opened for only writing.

2.1.95 Whate is the difference between multi channel descriptors(mcd) and file descriptors(fd)?

Ans: The multi channel descriptor mcd is a 32obit reg in which a single bit is set indicating which file is opened. Unlike multi channel descriptors, file descriptors can not be combined via bitwise or in order to direct output to multiple files.

Instead, files are opened via file descriptor for input, output, input and output, as well as for append operations, based on the value of type.

2.1.96 What is the difference between $sformat and $swrite?

Ans: The system task $sformat is similar to the system task $swrite, with a one major difference Unlike the display and write family of output system tasks, $sformat always interprets its second argument,and only its second argument as a format string. This format argument can be a static string, such as "data is 0" , or can be a reg variable whose content is interpreted as the format string. No other arguments are interpreted as format strings. $sformat supports all the format specifiers supported by $display,

2.1.97 What happens if a port is unconnected?

Ans:Unconnected input ports initialize to z and feed that value into the component, which can cause problems.More common are redundant or unwanted outputs which are left unconnected to be optimized away in synthesis.

2.1.98 What is compilation ?

Ans: To simulate a Verilog model, we must first convert our sourcerfiles into a binary formot that can be recognized by the simulator.The process of checking the syntax and producing the binary file is known as compilation.

2.1.99 What is the difference between static function and automatic function?

Ans: Automatic function local variables Can not seen in wave form viewer. We cannot use $Monitor and $strobe on local variables also.

2.1.100 What is advantage of wand and wor over wire?

Ans: It support Technology-dependent logic conflict resolution.

wired-AND for openocollector

wired-OR for ECL

2.1.101Is it possible to synthesize for loop?

Ans: for loop with fixed limits can be Synthesized

2.1.102Whateis the difference between posedge and negedge?

Ans: A negedge shall be detected on the transition from 1 to x,ez, or0, and from x or z to 0 where as posedge shall be detected on the transition from 0 t x, z,z orx, and from x or z to 1.

2.1.103What is the difference between initial and always block?

Ans: NOTE:Initial block can also be synthesized. Ref to IEEE Verilog Synthesis slandered.

2.1.104Whateis the difference between 0 and %z format specification?

Ans:

o or O Unformatted 2 value data

%z or %Z Unformatted 4 value data

2.1.105What is the difference between 0.000000e+00,0.000000 and 0?

Ans:

0.000000e+00 or 0.000000E+00 Display real in an exponential format

0.000000 or 0.000000 Display real in a decimaloformat

0 or 0 Display real in exponential or decimal format, which ever format results in the shorter printed output

2.1.106What is the difference between $finish ad $stop?

The $finish system task simply makes the simulator exit and pass control back to the host operating system.

The $stop system task causes simulation to be suspended.

2.1.107What is the difference between PLI and VPI?

Ans: Verilog Procedural Interface routines, called VPI routines, are the third generation of the PLI.

2.1.108What is the difference between $finish(0),$finish(1) and $finish(2)?

Ans: The $finish system task simply makes the simulator exit and passrcontrol back to the host operating system.

If an expression is supplied to this task,then its value (0,1,or 2) determines the diagnostic messages that are printed before the prompt is issued. If no argument is supplied, then a value of 1 is taken as the default.

$finish(0)Prints nothing

$finish(1)Prints simulation time and location

$finish(2)Prints simulation time,location, and statistics about the memory and CPU time used in simulation

2.1.109What is the difference b/w $time , $stime and $realtime ?

Ans: The $time system function returns an integer that is a 64-bit time,scaled to the timescale unit of the module that invoked it.

The $stime system function returns an unsigned integer that is a 32-bitetime, scaledito the timescale unit of the module that invoked it. If the actual simulation time does not fit in 32 bits, the low order 32 bits of the current simulation time are returned.

The $realtime system function returns a real number time that, like $time, is scaled to the time unit of the module that invoked it.

2.1.110What is the difference between the following two programs?

```
a)initial

#10a=0;

always@(a)

a<=~a;

b)initial

#10a =0;


always@(a)

a= ~a;
```

Ans:

When "a=~a" is evaluated and 'a'is updated,clearly you must agree that execution is *not* stalled at the @a event control.Â When execution reaches the @a event control, 'a' has already changed. Â It will not change again. Â So the event control will stall forever; its event of interest has already occurred, earlier in the same time slot, and can no longer have any effect.

2.1.111 What is the functionality of $input?

Ans: The $input system task allows command input text to come from a named file instead of from the terminal. At the end of the command file, the input is switched back to the terminal.

2.1.112 What is the MCDivalue of STD OUTPUT ?

ANS :0000000000000000000000000000001

2.1.113 To modify a behavioral Verilogowaitqstatement to makeeit synthesized

Originalcode:

```
command1;
wait(x !=0);
command3;
```

Ans:

Synthesized Verilog:

```
case(state)
0:begin
 command1;
if(x !=0)command3;
else state <=1;
end

1:if (x!= 0)//wait until this is true
command3;
endcase
```

2.1.114 What is configuration block?

Ans: Verilog-2000 adds configuration blocks,which allow the exact version and sourcerlocation of each Verilog module to be specified as part of the Verilog language. For portability,virtual model libraries are used in configuration blocks,and separate library map files associate virtual libraries with physical locations. Configuration blocks are specified outside of module definitions.

2.1.115 How many files can be opened(without closing) using multichannel descriptor ?

Ans:31ifiles.

2.1.116Why only 31ifiles can beqopened(without closing) using multichannel descriptor while integer can hold 32 bits?

Ans: The most significant bit (bit 32)of a multi channel descriptor is reserved, and shall always be cleared,limiting an implementation to atimost 31 files opened for output via multiychannel descriptors.

2.1.117 If mcd(multichannel descriptor)is 0000000000000000000000000000001,then what does it mean?

Ans: The least significant bit (bit 0) of a mcd always refers to the standard output.

2.1.118 What are the typical tasks you perform inside a specify block?

Ans: -Describe various paths across the module and assign delays to those paths.

-Describe timing checks to ensure that the timing constraints of the device are met.

-Define the pulse filtering limits for a specific module or for particular paths within a module.

2.1.119 Find the bug in the following code.

```
if(a=b)
match= 1;
else
match= 0;
```

Ans:

if(a=b) assigns b to a, the if a is non-zero sets match. The correct code is

```
if (a==b)
matchi= 1;
else
matchi= 0;
```

2.1.120 Find the bug in the followingqcode.

```
for(............);
begin
.....
end
```

Ans:

Misplaced semicolons in for-loops

2.1.121 Find the bug in the following code.

```
automatic task intra_assign();
begin
a<= #10b;
end
```

ANS: Intra assignment nonblocking statements are not allowed in automatic tasks.

2.1.122 Find the bug in the following code.

```
always@ (in )
if(ena)
out= in ;
else
out= 1;
```

Ans: Synthesis and simulation mismatch might occur.

To assure the simulation will match the synthesized logic, add "ena" to the event list so the event list reads:alwayse@ (in or ena)

2.1.123Find the bug in the followingqcode.

```
alwaysi@ (in1 or in2 or sel)
begin
out= in1 ;
if(sel)
out<= in2 ;
end
```

Ans: Not supported, cannot mix blocking and nonblocking assignments in an always statement.

2.1.124Find the bug in the following code.

```
reg[1:0] select;
always@(select)
begin
case(select)
00:y =1;
01:y =2;
```

10:y =3;

11:y =4;

endcase

end

Ans: branches 01 and 11 are considered as integers and they will never be selected.

2.1.125What is calltf in Verilog PLI?

Ans: Calltf is similar to main() function in C.calltf can call other sub methods to different jobs.

2.1.126 What is PLI?

Ans: PLIiis a mechanism to invoke c subroutines to Verilog.

2.1.127 What is the difference between Parallel and Full Connection Module Path delays?

Ans: A parallel connection establishes a connection between each bit in the source to each corresponding bit in the destination.
Parallel module paths can be created only between one source and one destination where each signal contains the same number of bits.

A full connectiono stablishes a connection between every bit in the source and every bit in the destination.The module path source does not need to have the same number of bit as the module path destination.

2.1.128 What is misctf ?

Ans: Misctf is to do housekeeping jobs.This is can be called many times unlike other predefined functions which are called once per instance.

2.1.129 What construct in Verilog can be used to simulate a capacitive storage node in a circuit?

Ans: The trireg statement is used to simulate a wire with a capacitive hold value.

2.1.130 Describe the basic strength system in Verilog.

Ans: class="aaa" The strength system has 8 values 0 through 7, with the strongest strength being known as supply and the weakest strength as high impedance.

2.1.131 #define cat(x,y) x y concatenates x to y.But cat(cat(1,2),3) does not expand but gives preprocessor warning.Why?

Ans: Because parameterized macros are not recursive.

2.1.132 What are the types of strengths that can be specified on a net ??

Ans: There are two types of strengths that can be specified in a net declaration.They are as follows:

charge strength shall only be use when declaring a net of type trireg drive strength shall only be use when placing a continuous assignment on a net in the same statement that declares the net

2.1.133 How to resolve a tristate driver in Verilog?

2.1.134 WHAT ARE THE TYPES OF CHARGE STRENGTHS?

Ans: SMALL,MEDIUM, LARGE

2.1.135 How to model power supply strengths in verilog?

Ans: The supply0 and supply1 nets may be used to model the power supplies in a circuit. These nets shall have supply strengths.

2.1.136 How to modify a localparam?

Ans: Localiparameters can be assigned to a constant expression containing a parameter which can be modified with the defparam statement or by the ordered or named parameter value assignment.

```
Parameter WIDo=3;
LocalparamiWIDTH =2*WID;
```

2.1.137 WHAT IS specparam?

Ans: specparam declares a special type of parameter which is intended only for providing timing and delay values, but can appear in any expression that is not assigned to a parameter and isnot part of the range specification of a declaration. Unlike a module parameter, a specify parameter cannot be modified from within the language, but it may be modified through SDF annotation

2.1.138 What does the followingicode mean?

```
Reg[22:0] sig;
always@(|sig)
begin
......
end
```

2.1.139 What is the function of force & release?

Ans: Force and release statements are used to override assignments on both registers and nets.They are typically used in the interactive debugging process,where certain registers or nets are forced to a value and the effect on other registers and nets is noted.

They should occur only in simulation block.

2.1.140 What is the purpose of declaring tasks or functions as automatic?

Ans: Declaration of tasks and functions as Automatic will create dynamic storage for each task or function call.

2.1.141 Weather initiale block can be synthesized?

Ans:YES According to IEEE

2.1.142 How to read data from a file?

Ans: Using $readmemh and $readmem0.File io

2.1.143 Illustrate with example the declaration of local variable inside abegin ...end block?

Ans: TO declare a local variable inside begin...end block,the block should be named.

```
begin:name_
integer varb;
....
....
end
```

2.1.144 If A and B are two clk pulses which are out of phase and having sameqfrequency, how to find which input clk signal is leading? Write verilog code for this.

Ans:

Here is the simple solution forithis.

All it needs is a flipiflop.

if we have 2 clks, clk1 and clk2 give clk1 to D-input of flipiflop and other CLK input of FF.

if clk1 is leading the output is high.

if clk2 is leading the output is low.

2.1.145 Verilog code to detect if a 64bit pattern can be expressed using power of 2

Ans:

```
module pat_det (data_in,patDetected );
inputi[31:0] data_in;
output patDetected;

wire [4:0] patSum =data_in[0] +data_in[1] +data_in[2]+
data_in[3]+ data_in[4]+edata_in[5] +
data_in[6]+ data_in[7]+edata_in[8] +
data_in[9]+ data_in[10]+edata_in[11] +
data_in[12]+ data_in[13]+edata_in[14] +
data_in[15]+ data_in[16]+edata_in[17] +
data_in[18]+ data_in[19]+edata_in[20] +
data_in[20]+ data_in[21]+edata_in[22] +
data_in[23]+ data_in[24]+edata_in[25] +
data_in[26]+ data_in[27]+edata_in[28] +
data_in[29]+ data_in[30]+edata_in[31] ;

wire patDetected =(patSume== 1)?1'b1: 1'b0;
endmodule
```

2.1.146 What is the use of $printtimescale?

Ans: The $printtimescale system task displays the time unit and precision for a particular module. When no argument is specified, $printtimescale displays the time unit and precision of the module that is the current scope.

When an argumentois specified, $printtimescale displays the time unit and precision of the module passed to it.

2.1.147 What will be the syntheses output of the following verilog code?

```
alwaysi@ (*)
if(enable)
q<= d;
```

Ans:

Alevel-sensitive storageodevice eis inferred for q. If enable is deasserted,q will hold its value.

2.1.148What will be the syntheses output of the following verilog code?

```
always@ (enable or d)
if(enable)
q<= d;
else
q<= 'b0;
```

Ans:

A latch is not inferred because the assignment to q is complete,

2.1.149 How to model a sequential circuit?

Ans:

Sequential logic shall be modeled using an always statement that has one or more edge events in the event list.

2.1.150 What is the use of $timeformat?

Ans:

The $timeformat system task performs the following two operations:

It sets the time unit for all later-entered delays entered interactively.

It sets the time unit, precision number, suffix string, and minimum field width for all %t formats specified in all modules that following the source description until another $timeformat system task is invoked.

2.1.151 How to describe delays for structural models such as ASIC cells?

Ans:

Two types of HDL constructs are often use to describe delays for structural models such as ASIC cells.

They are as follows:

Distributed delays, which specify the time it takes events to propagate through gates and nets inside the module

Module path delays, which describe the time it takes an event at a source (input port or inout port) to propagate to a destination (output port or inout port)

2.1.152 How to model a tri state driver?

Ans: Three-state logic shall be modeled when a variable is assigned the value z. The assignment of z can be conditional or unconditional. If any driver of a signal contains an assignment to the value z, then all the drivers shall contain such an assignment.

```
module ztest (test2,test1,test3, ena);

input[0:1] ena;

input[7:0] test1,test3;

output[7:0] test2;

wire[7:0] test2;

assign test2 =(ena== 2'b01) ?test1:8'bz;

assignitest2 =o(ena== 2'b10) ?test3: 8'bz;


Endmodule
```

2.1.153 How to model a ROM?

Ans:

An asynchronous ROM shall be modeled as combinational logic using one of the following styles:

a)One-dimensional array with data in case statement.

b)Two-dimensional array with data in initial statement.

c)Two-dimensional array with data in text file.

2.1.154 How to model a RAM?

Ans:

A RAM shall be modeled using a Verilog memory (a two-dimensional reg array) that has the attribute ram_block associated with it.A RAM element may either be modeled as an edge-sensitive storage element or as a level-sensitive storage element. A RAM data value may be read synchronously or asynchronously.

```
// RAMoelementeis an edge-sensitive storage oelement:
module ram_test(
output wire [7:0]q,
input wire [7:0]d,
input wire [6:0]a,
input wire clk,we);
(*is nthesis, ram_blocko*)reg [7:0]mem [127:0];
always@(posedge clk)if(we ) mem[a]<= d;
assign q =mem[a];
endmodule
```

2.1.155 Is it possible to synthesize power operator(**)?

Ans: The power operatoro(**) shall be supported only when both operands are constants or if the first operand is 2.

2.1.156 What is use of Escape sequences for special character\ddd?

\ddd A character specified by 1 to 3 octal digits

```
module disp;
initialbegin
$display("\\\t\\\n\"\123");
end
endmodule
```

Simulating this example shall display the following:

```
\\
"S
```

2.1.157What is the format specification to Display in ASCII character format

Ans:

^@ or %C

2.1.158 What is the format specification to Display library binding information

Ans:

%l or %L

2.1.159 What is the format specification to Display net signal strength

Ans:

%v or %V

2.1.160 What is the format specification to Display hierarchical name

Ans:

%m or %M

2.1.161 What is the format specification to Display in current time format

Ans:

%t or %T

2.1.162 What is VCD ?

Ans: A value changeodumpe(VCD) fileicontains in ormationoaboutqvalue changesroneselected variablesiin the designjstored by valuerchange dump system tasks.

a)Four state:to represent variable changes in 0,q1, x,and z with no strength information.

b)Extended: to represent variable changes in all states and strength information.

2.1.163 What are the ways to model a combinational circuit?

Ans: Combinational logic shall be modeled using a continuousoassignment or a netedeclaration assignment or an always statement.

2.1.164 What are the rules to be fallowed while using an always statement to model combinational circuit?

Ans: When using an always statement, the event list shall not contain an edge eventi(posedge or negedge). The event list does not affect the synthesized net list However, it may be necessary to include in the event list all the variables read in the always statement to avoid mismatches between simulation and synthesized logic.

A variable assigned in an always statement shall not be assigned using both a blocking assignmento =) and a nonblocking assignment (<=) in the same always statement.

2.1.165 How to get the system time in to verilog?

Using PLI or VPI.

2.1.166 What is value of a ?

```
reg[2:0] d;

rega;

d= 3'b1xx;

a = |d;
```

Ans: a is 1;

2.1.167 I want to return 2 values by a function. How do i do ?

Ans: Concatenate the results and return.

2.1.168 How many times this loop will get executed?

```
reg[3:0] i;

for(i=0; i<=15;oi=i+1)

begin

.......

end
```

Ans: infinite times.

i<=15 will always be true as i is 4 bit only.

2.1.169 What is the result of the following code?

```
Integeria,b,c;

A= 10;

b= 5;
```

```
c= 7;
if(i> b>c)
$display(true);
else
$display(False);
```

Ans: TRUE.

A> b>c is interpreted by verilog simulator as (a >(b >c))

10> (5>7) i-> 10 >0 i ->0 1

2.1.170 How to print line and file name from where the $display message is coming?

Ans: Using the following plie, we can print the file name and line number by overridden the $display task.

2.1.171 Is there a race condition in the following programr?

```
begin
a=0;
b<= a;
end
```

Ans:NO. There is no race condition .

2.1.172 How many times the following repeat loop executes?

```
begin
a = 1;
repeat (a)
a= a + 1;
end
```

Ans: Only once. Loop execution for a specific number of times. This construct is associated with a constant or a variable. If a variable is use , it is evaluated once when the loop starts. When the loop is started, the value of a is 1. So the loop is executed once.

2.1.173 At what time a,b,c,d,e,f will get 1 in the following program?

```
module block_nonblock();
reg a,b,c, d, e,f ;

// Blockingoassignments
initial begin
a =#10 1'b1;
b =#20 1'b1;
c =#40 1'b1;
end

// Nonblockingoassignments
iinitial begin
d <=#10 1'b1;
e <=#20 1'b1;
f <=#40 1'b1;
end

endmodule
```

Ans:

```
// Blockingoassignments
initial begin
a =#10 1'b1; //The simulator assigns 1 to a at time 10
b =#20 1'b1; //The simulator assigns 1 to b at time 30
c =#40 1'b1; //The simulator assigns 1 to c at time 70
end

// Nonblockingoassignments
initial begin
d <=#10 1'b1; //The simulator assigns 1 to d at time 10
e <=#20 1'b1; //The simulator assigns 1 to e at time 20
f <=#40 1'b1; //The simulator assigns 1 to f at time 40
end
```

2.1.174How to relize "always@(posedge clock)" with out using always block?

```
Initial
forever
begin
@(posedge clock);
.......code ............
end
or
initial
while(1)
begin
@(posedgeiclock);
.......code............
end
```

2.1.175 Is it possible to use negative numbers while specifying vector indexes?

Ans: Yes.

reg [-1:4] b; // a 6-bit vector reg

2.1.176 How many bits are there in integer?

Ans: It is implementation dependent.but the shall at least be 32 bits.

2.1.177 How many bits are there in time variable?

Ans: 64

2.1.178 What are the different phases of execution?

2.1.179 What is the valuei f a?

Integer i =3.5

Ans:4.

2.1.180 What is the value of a?

integer a =-3.5

Ans:it is -4

If the fractional parteof the real number is exactly 0.5, it shall be rounded away from zero.

2.1.181 Where the operator "or" is used?

Ans:Used in events.

2.1.182 What is difference between define and parameter? Which do you prefer and why?

2.1.183 What is the valueiof a?

Integer a =-12/3;

Ans: The result is -4.

2.1.184 IN the following code,what message will be displayed(specifically about the number of bits it prints)?

```
module bitlength();
reg[3:0] a,b,c;
reg[4:0] d;
initial begin
a= 9;
b= 8;
c= 1;
$display("answer= 0",c? (a&b)i: d);
end
endmodule
```

Ans: The $display statement will display:

```
answer= 01000
```

By itself, the expression a&b would have the bit length 4,but since it is in the context of the conditionalexpression, which uses the maximum bit-length,the expression a&b actually has length 5,the length of d.

2.1.185 IN following program, at what time the statements are scheduled and executed?

```
module multiple2 ;
reg a;
initial #8 a<=#8 1;
initial #12 a<=#4 0;
```

endmodule

Ans:

initial #8 a<=#8 1;//executed at time 8; schedules

//an update of 1 at time 16

initial #12 a<=#4 0;//executed at timeq1; schedules

//an update ofe0 at time 16

2.1.186 Is there a race condition in the above code?

Ans:

NO.Because it is determinate that the update of a to the value 1 is scheduled before the update of a to the value 0,then it is determinate that a will have the value 0 at the end of time slot 16.

2.1.187 What is the value of answer?

reg [15:0] a,b,answer; //16-bit regs

a= 16'hf000;

b= 16'hf000;

answer= (a+b) >>i;

Ans: will not workoproperly . where a and b are to be added, which may result in an overflow, and then shifted righ iby 1bit to preserve the carry bit in the 16-bit answer.

A problem arises,however,because all operands in the expression are of a 16-bit width. Therefore,the expression (a +b) produces an interim result that is only 165bits wide,thus losing the carry bit before the evaluation performs the 1-bit right shift operation.

To solve the above problem

The solution is to force the expression (a + b) to evaluate using atleast 17 bits.Forqexample, adding an integer value of 0 to the expression will cause the evaluation to be performed using the bit size of integers. The following example will produce the intended result:

answer i= (a+b +0) >>1;

2.1.188 How many times the begin..end block will get executed?

integer a;

a= 'Z;

repeat(a)

```
begin
....
end
```

Ans: Repeat Executes a statement a fixed number of times.If the expression evaluates to unknown or high impedance,it shall be treated as zero, and no statement shall be executed.

2.1.189How many times the begin..end block will get executed?

```
integer a;
a= 'bx;
repeat(a)
begin
....
end
```

Ans: repeat Executes a statement a fixedinumber of times. If the expression evaluates to unknown or high impedance,it shall be treated as zero, and no statement shall be executed.

2.1.190What is the equivalent always@(*) in the following program?

```
always @*
begin
y= 8'hff;
y[a]= !en;
end
```

Ans: equivalent to as @(a or en)

2.1.191 Weather non bocking statements are allowed in function?

Ans: No.Non Blocking statements are not allowed in function.

2.1.192 Maximum number files can be opened using fopen?

2.1.193 Weather noneblocking statements are allowed in automatic task?

Because variables declared in automatic tasks are deallocated at the end of the task invocation, shall not be used inecertain constructs like nonblocking that might refer to them after that point.

2.1.194 Is it possible to see the automatic task local variables in waveform debugger?

Ans: No it is not possible to see. These variables are automatically deallocated at the end of task invocation.

2.1.195 Is it possible to use automatic task local variables in $monitor?

Ans: No. These variables are automatically deallocated at the end of task invocation.

2.1.196 Is it possible to use procedural continuous assignments or procedural force statements on automatic task local variables?

Ans:No.

2.1.197 Is it possible to disable a function?

Ans: A function canot be disabled . The disable statement can be used to disable named blocks within a function. In cases where a disable statement within a function disables a block or a task that called the function,the behavior is undefined.

2.1.198 Between the if-else and case statements which is usually preferred?

Ans:

Case is better from synthesis point of view.

if else will be synthesized to a priority encoder.

Where case will be synthesized to a normal encoder.

Priority encoder has more gates and also timing is affected.

So,case is usuallyopreferred.

There are switches that design compiler(synopses synthesis to l)provides to synthesizer caseestatement either way.

2.1.199 How to display the messages in colorful?

Ans: The following program shows how to display messages in colorful.

Thisiworks only in ASIC Terminals.

Simulate the following code in Linux or Unix and see the outputs.

```
module asdsadf();
```

```
initial
begin
$write("^@[1;34m",27);
$display("***********Â 0Â Â Â Â Â Â Â Â  This is in blue***********", 1);
$write("^@[0m",27);

$display("^@[1;31m",27);
$display("***********Â 0Â Â Â Â Â Â Â Â  This is in red***********", 2);
$display("^@[0m",27);

$display("^@[0;33m",27);
$display("***********Â 0Â Â Â Â Â Â Â Â  This is in penda color ***********",o3);
$display("^@[0m",27);

$display("^@[5;34m",27);
$display("***********Â 0Â Â Â Â Â Â Â Â  This is in Blink***********", 4);
$display("^@[0m",27);

$display("^@[7;34m",27);
$display("***********Â 0Â Â Â Â Â Â Â Â  This is in Back ground color***********",q1);
$display("^@[0m",27);



end
Endmodule
```

2.1.200 What is TOP module?

Ans: Top-level modules are modules that are included in the source text but are not instantiated. In verification environment,the highest module in the hirarchi is generally named as top.

2.1.201 How to declare real numbers as ports?

Ans: Real number are allowed to be declared as ports. To use realenumbers as ports, use $bitstoreal and $realtobits.

2.1.202 Explain about `resetall.

Ans: When `resetall compiler directive is encountered during compilation, all compiler directives are set to the default values.

This is useful for ensuring that only those directive that are desired in compiling a particular source file are active.

2.1.203 How do you implement the bi-directional ports in Verilog HDL?

```
module bidirec (oe,clk,inp, outp,bidir);

//Port Declaration
input oe;
input clk;
input[7:0] inp;
output[7:0] outp;
in ut[7:0] bidir;
reg[7:0] a;
reg[7:0] b;
assign bidir =oe? a: 8'bZo;
assign outp =b;

//Always Construct
alwaysi@ (posedgeoclk)
begin
b<= bidir;
a<= in ;
end
endmodule
```

(Qi290) If the part-select is out of the address bounds or the part-select is x or z, then the value is returned by the

reference ?

Ans: Its 'bxo.

2.1.204 What is the difference between constant function and ordinary function ?

Ans: Constant function calls are used to support the building of complex calculations of values at elaboration time .A constant function call shall be a function in ocation of a constant function local to the calling module where the arguments to the function are constant expressions. Constant functions are a subset of normal Verilog functions that shall meet the

following constraints:

--They shall contain no hierarchical references.

--Any function invoked within a constant function shall be a constant function local to the current module. System functions

shall not be invoked.

--All system tasks within a constant function shall be ignored.

--All system functions within a constant function shall be illegal.

--The only systemetask that may be invoked is $display, and it shall be ignored when invoked at elaborationrtime.

2.1.205 What is genvar ?

Ans: An index variable that shall only be declared for use in generate statements shall be declared as a genvar.

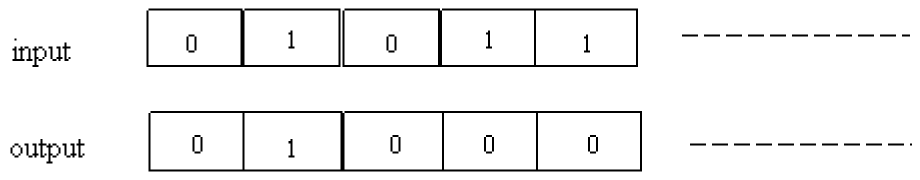
2.1.206 In the case of multiple defparams for a single parameter,what value it will take?

Ans: The parameter takes the value of the last defparam statement encountered in the source text. When defparams are encountered in multiple source files,ie.g., found by library searching, the defparam from which the parameter takes its value is undefined.

2.2 Questions without answers

2.2.1 How to find out number of ones in a 20 bit register is 10.

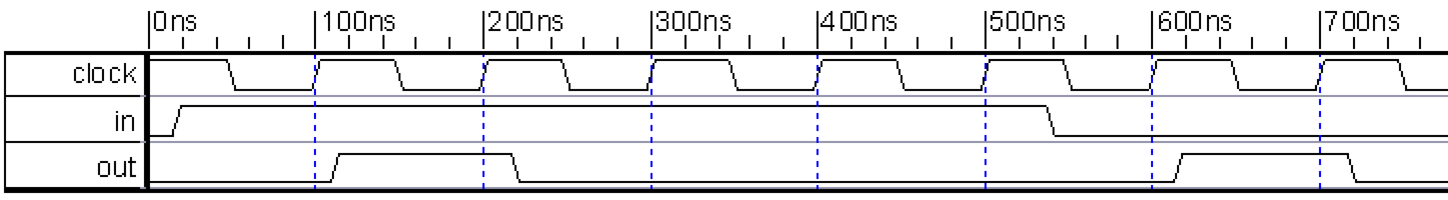
2.2.2 Design a "Stackable First One" finder.



Design a small unit which processes just 1 bit. Design in such a way that it can be stacked to accept make input of any length.

2.2.3 Design Gray counter to count 6

2.2.4 Design a block to generate output according to the following timing diagram.



2.2.5 Design a hardware to implement following equations without using multipliers or dividers.

out = 7x + 8y;

out = .78x + .17y;

2.2.6 Which one is preferred? 1's complement or 2's complement and why?

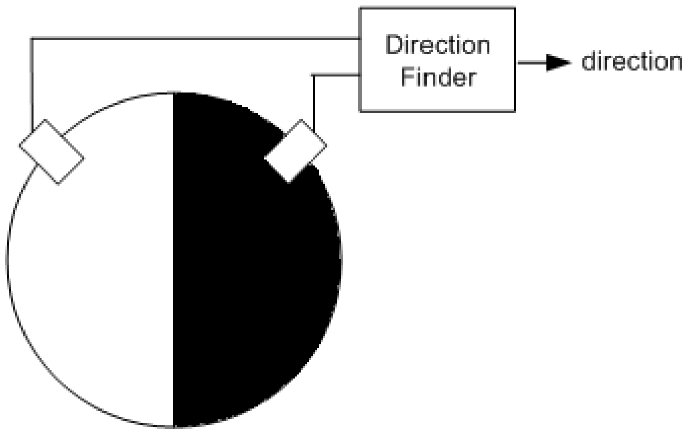
2.2.7 Which one is preferred in FSM design? Mealy or Moore? Why?

2.2.8 Which one is preferred in design entry? RTL coding or Schematic? Why?

2.2.9 Design a 2 input AND gate using a 2 input XOR gate.

2.2.10 Design a Pattern matching block Output is asserted if pattern "101" is detected in last 4 inputs. How will you modify this design if it is required to detect same "101" pattern anywhere in last 8 samples?

2.2.11 Design Direction finder block using digital components (flip flops and gates) to indicate speed. Logic 0 for clockwise and Logic 1 for counter clockwise.



Optical sensors A and B are positioned at 90 degrees to each other as shown in Figure. Half od the disc is white and remaining is black. When black portion is under sensor it generates logic 0 and logic 1 when white portion is under sensor.

2.2.12 What is the difference between code-compiled simulator and normal simulator?

2.2.13 What is the difference between casex, casez and case statements?

2.2.14 Which one preferred-casex or casez?

2.2.15 What is delta simulation time?

2.2.16 Create 4 bit multiplier using a ROM and what will be the size of the ROM. How can you realize it when the outputs are specified.

2.2.17 What is a compiler directive like 'include' and 'ifdef'?

2.2.18 What is the difference between inter statement and intra statement delay?

2.2.19 Why latches are not preferred in synthesized design?

2.2.20 What is file I/O?

2.2.21 What is difference between freeze deposit and force?

2.2.22 Will case always infer priority register? If yes how? Give an example.

2.2.23 How can you model a SRAM at RTL Level?

2.2.24 What are different styles of Verilog coding I mean gate-level,continuous level and others explain in detail?

2.2.25 Identify the error in the following code. b[7:0]= {2{5}};

2.2.26 When are instance names optional?

2.2.27 In the following program, what is the problem and how to avoid it?

```
Task driver;
input read;
input [7:0] write_d;
begin
#30 date_valid =1'b1;
wait(read== 1'b1);
#20cpu_data =write_data;
$display("End of task");
end
endtask
```

2.2.28 What is the functionality of &&& (not &&,not &)?

2.2.29 In statement ((a==b) &&(c== d)), what is the expression coverage if always a=0,b=0,c=0,d=0?

2.2.30 How to generate a random number?

2.2.31 How to generate a random number which is less the 100?

2.2.32 How to generate a random number which is between 0 100 ?

2.2.33 What is the advantage of Named Port Connection overe ordered Port Connection ?

2.2.34 How to generate a random number between 44 to 55?

2.2.35 How to get different random numbers in different simulations?

2.2.36 What is the difference between @(a or b) and @(a | b)

2.2.37 What data types can be used for input port,outut port and inout porto?

2.2.38 What is the functionality of trireg?

2.2.39 What is the functionality of tri1 and tri0 ?

2.2.40 Difference between conditional compilation and $plusargs??

2.2.41 What is the benefit of using Behavior modeling styler over RTL modeling?

2.2.42 What is the difference between task and function?

2.2.43 Identify the error in the followingocode.

a[7:0]i= {4{'b10}};

2.2.44 What is the difference between && and &, if any?

2.2.45 What is the difference between static task and automatic task?

2.2.46 Is it synthesysable ? always @(negedge clk or rst)

2.2.47 What is the difference between $stop and $finish ?

2.2.48 What is the difference between the followingqtwo statements? @(val ==2) wait(val== 2)

2.2.49 Difference between Vectored and scalared nets?

2.2.50 Difference b/w real and realtime?

2.2.51 What is the difference between arthamatic and logical shift register?

2.2.52 What is the difference b/w following two registers?

reg[1:n] rega;//An n-bitiregister is not the same

reg mema [1:n];//as a memory of n 1-bit registers

2.2.53 What is the difference between parameters and specparams?

2.2.54 How is time advanced in a simulation?

2.2.55 Name three methods of timing control?

2.2.56 What is behavioral modeling used for?

2.2.57 How do you define the states for an FSM?

2.2.58 What is the difference between force release and assign deassign?

2.2.59 What sort of hardware structure are inferred by both case and if statements, by default,in Verilog?

2.2.60 How could you change a case statement in order that its implementation does not result in a priority structure?

2.2.61 If you are not using a synthesis attribute "full case",how can you assure coverage of all conditions for a case statement ?

2.2.62 How do you infer tristate gates for synthesis?

2.2.63 Differenceebetween ! And ~?

2.2.64 What is the difference between $test$plusargs and $value$plusargs?

2.2.65 Whateis the difference Difference between the two statement? Weather a and b values are equal?

reg[1:0] data;

a= data[0]||data[1];

b= |data;

2.2.66 Why is it recommended not to mix blocking and non-blocking assignments in the same block?

2.2.67 Declare parameters for representing the stateqmachine states using one hot encoding.

2.2.68 What do sea functionsynthesize to ?

2.2.69 How to change the value of width to 3 in the following code ? `defineiwidth 7

2.2.70 What are the types of race conditions?

2.2.71 How to avoide race condition between dut and testbench?

2.2.72 Give the guideilines which avoids race condition.

2.2.73 What is the use of linting tool?

2.2.74 Write the code to instantiated 1k "and gates" in a module.

2.2.75 Which is better to use when creating test vectors? $display or $strobe?

2.2.76 How would you cater with opening 35 files?

2.2.77 Find the bug in the following code.

```
always@(posedge clk)
a= b;
always@(posedge clk)
b= a;
```

2.2.78 Fill the ????

```
fd = $fopen("filename",r);
if(????)
$display("file cannot be opened");
```

2.2.79 How to model a perfect buffer of 10units delay?

a)#10 a=b;

b)a =#10b;

c)#10a <=b;

d)a <=#10b;

2.2.80 What is verilog configuration?

2.2.81 Write a code for clock generator.

2.2.82 Write a code for clock generator which can generate clock frequency of 156MHZ.

2.2.83 Write a verilog code to generate 40MHz clock with 60% duty cycle

2.2.84 What points need to be considered while writing a clock generator??

2.2.85 How the scope of a variable is realized in verilog.Illustrate withiexample.?

2.2.86 What is incremental compilation?

2.2.87 In what region of the event queue , PLI calls are executed?

2.2.88 Can $setup and $hold check report a violation for a limit of zero?

2.2.89 Explain about $recovery and $removal ?

2.2.90 Which timingecheck(s) acceptia negative limit?

2.2.91 Can you qualify all events in all timing checks with edge specifiers such as edge 01?

2.2.92 For which timing check(s) must you always qualify events?

2.2.93 When does $skew report a violation?

2.2.94 What is UDP? Can we write UDP including clock also?

2.2.95 What are >>> and <<<operators?

2.2.96 Write code for parallel encoder and priority encoder?

2.2.97 What is full case?

2.2.98 How to model a full case block?

2.2.99 What are rules need to be fallowed while using case statement?

2.2.100 How to Model a capacitor ?

2.2.101 what is the use of $timeformat();

2.2.102 How to declare strings in verilog?

2.2.103 Write code for clockidivider and clock multiplier?

2.2.104 List out the simulation and synthesis mismatches.

2.2.105 Model a 3 bitishift register?

2.2.106 how to overcome racing condition?

2.2.107 What is the use Always@(*) ?

2.2.108 What is code coverage?

2.2.109 List outethe types of codeocoverage.

2.2.110 List out some points to speed up simulation.

2.2.111 At what time the simulation stops??

```
initial
while(1)
$display("ajkdkjs");


initial
#10$finish;
```

2.2.112 What is value of a?

```
reg[2:0] a,b,c;
c= 3'b110;
a= b=c ;
```

2.2.113 Given the following code, draw the waveforms for 'a':

```
reg clk;
reg a;
always #10 clk = ~clk;
always @(clk) a=#15 clk;
```

2.2.114 By default Numbers that are specified without a base format specification are

Options :

a)decimal number

b)hexadecimal number

c)binay

d)octal

2.2.115Default value of a net,trireg is

a)logic 0

b)logic1

c)unknown

d)hi-impedence

2.2.116 How do you make out whether always block is a combinational or sequential?

2.2.117 What will be displayed?

```
reg[8*10:1] s1,os2;
intitial begin
s1= "Hello";
s2= "world!";
if({s1,s2} =="Hello world!")
$display("strings are equal");
else
$display("strings are not equal");
end
```

Ans:The comparison in this example fails because during the assignment the string variables are padded as illustrated in the next

example:

s1= 000000000048656c6c6f

s2= 00000020776f726c6421

The concatenation of s1 and s2 includes the zero padding, resulting in the followingivalue:

000000000048656c6c6f00000020776f726c6421

and "hello world"oise48656c6c6f20776f726c6421

2.2.118 How many times the begin..end block will get executed?

```
repeat(-3)
begin
....
end
```

2.2.119 How many times the begin..end block will get executed?

```
repeat(3.5)
begin
....
end
```

2.2.120 What time is displayed?

```
initial
begin
a=3;
#a a= a*2;
$display($time);
end
```

2.2.121 What is the message displayed?

```
initial
begin
#10;
a= 0;
a= 1;
end
always@(a)
$display("a is 0",a);
```

2.2.122 What message is displayed?

```
initial
begin
a= x;
#1a =z;
```

```
end
always@(a)
$display("a is 0",a);
```

2.2.123What message is displayed?

```
initial
begin
a= x;
#1 a =1;
end
always@(posedge a)
$display("posedge on a is seen");
```

2.2.124 What is the equivalent always@(*) in the following program?

```
always@(*)
y= (a&b) |(c &d)| myfunction(f);
```

Ans: equivalentito @(a or b or c or d or f)

2.2.125 What is the equivalent always@(*) in the following program?

```
always@*
begin
tmp1= a&b;
tmp2= c&d;
y= tmp1|tmp2;
end
```

Ans: equivalent to @(a or b or c or d or tmp1 or tmp2)

2.2.126 What is the equivalent @(*) in the following program?

```
always@*
begin
xi= a^b;
@(*)
xi= c^d;
```

end

2.2.127 What is mutex?

2.2.128 How to model a mutex in verilog?

2.2.129 What is semaphore?

2.2.130 How to model a semaphore in verilog?

2.2.131 What is the difference between mutex and semaphore?

2.2.132 When is fork-join use ?

2.2.133Which procedural assignment should be use to model a combinational logic buffer?

1)
```
always@(in )
#5out =in;
```

2)
```
always@(in )
#5out <=in;
```

3)
```
always@(in )
out= #5in;
```

4)
```
always@(in )
out<= #5in;
```

2.2.134Which procedural assignment should be use ot model a sequential logic flip-flop?

1)
```
always @(posedge clk)
#5q =d;
```

2)
```
always@(posedge clk)
```

```
#5q <=d;
```

3)

```
always@(posedge clk)
q= #5d;
```

4)

```
always@(posedge clk)
q<= #5d;
```

2.2.135 Explore and explain what happens if you write this:

```
always @(a or b or c) e =(a|b)&(c|d);
```

2.2.136 Explain the following:

```
integer IntA;
Int A= -12o/e3; //result is -4
Int A= -'do 2e/ 3;// result is 1431655761
```

2.2.137 What is the difference in the following sum statements?

```
reg [7:0] a, b,sum;
sum= (a+b) >>1;
sum= (a+b +0) >>1;
sum= {0,a}+{0,b} >>1;
```

2.2.138 How can you swap 2 integers a and b,without using a 3rd variable?

Ans: There are many solutions.

One of the solution is

```
bit a,b;
a=a XOR b;
b=a XOR b;
a=a XOR b;
```

2.2.139 What you mean by inferring latches?

2.2.140 How to avoid latches in yourdesign?

2.2.141 What is sensitivityilist?

2.2.142 How to do a variable part select of a vector?

2.2.143 Find the bug in the following code"

```
module backdrive(in ut wire a);
wire b;
assign a =b;
endmodule
```

2.2.144How to use generate for loop to instantiate a module?

2.3 Examples with code

2.3.1 Following is the Verilog code for flip-flop with a positive-edge clock.

```
    module flop (clk, d, q);
    input  clk, d;
    output q;
    reg    q;

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

2.3.2 Following is Verilog code for a flip-flop with a negative-edge clock and asynchronous clear.

```
    module flop (clk, d, clr, q);
    input  clk, d, clr;
    output q;
    reg    q;
```

```
   always @(negedge clk or posedge clr)
begin
     if (clr)
        q <= 1'b0;
     else
        q <= d;
   end
endmodule
```

2.3.3 Following is Verilog code for the flip-flop with a positive-edge clock and synchronous set.

```
module flop (clk, d, s, q);
input  clk, d, s;
output q;
reg    q;
always @(posedge clk)
begin
  if (s)
     q <= 1'b1;
  else
     q <= d;
end
endmodule
```

2.3.4 Following is Verilog code for the flip-flop with a positive-edge clock and clock enable.

```
module flop (clk, d, ce, q);
input  clk, d, ce;
output q;
reg    q;
always @(posedge clk)
begin
   if (ce)
  q <= d;
```

```
    end
endmodule
```

2.3.5 Following is Verilog code for a 4-bit register with a positive-edge clock, asynchronous set and clock enable.

```
    module flop (clk, d, ce, pre, q);
    input       clk, ce, pre;
    input  [3:0] d;
    output [3:0] q;
    reg    [3:0] q;
    always @(posedge clk or posedge pre)
       begin
      if (pre)
        q <= 4'b1111;
      else if (ce)
        q <= d;
        end
endmodule
```

2.3.6 Following is the Verilog code for a latch with a positive gate.

```
    module latch (g, d, q);
input  g, d;
output q;
    reg    q;
    always @(g or d)
begin
  if (g)
    q <= d;
end
endmodule
```

2.3.7 Following is the Verilog code for a latch with a positive gate and an asynchronous clear.

```
module latch (g, d, clr, q);
input  g, d, clr;
output q;
reg    q;
   always @(g or d or clr)
begin
  if (clr)
    q <= 1'b0;
  else if (g)
    q <= d;
end
endmodule
```

2.3.8 Following is Verilog code for a 4-bit latch with an inverted gate and an asynchronous preset.

```
module latch (g, d, pre, q);
input        g, pre;
input  [3:0] d;
output [3:0] q;
reg    [3:0] q;
always @(g or d or pre)
begin
  if (pre)
    q <= 4'b1111;
  else if (~g)
    q <= d;
end
endmodule
```

2.3.9 Following is Verilog code for a tristate element using a combinatorial process and always block.

```
module three_st (t, i, o);
input  t, i;
output o;
reg    o;
always @(t or i)
begin
  if (~t)
    o = i;
  else
    o = 1'bZ;
end
endmodule
```

2.3.10 Following is the Verilog code for a tristate element using a concurrent assignment.

```
   module three_st (t, i, o);
   input  t, i;
   output o;
     assign o = (~t) ? i: 1'bZ;
endmodule
```

2.3.11 Following is the Verilog code for a 4-bit unsigned up counter with asynchronous clear.

```
module counter (clk, clr, q);
input       clk, clr;
output [3:0] q;
```

```
reg   [3:0] tmp;
always @(posedge clk or posedge clr)
begin
  if (clr)
    tmp <= 4'b0000;
  else
    tmp <= tmp + 1'b1;
end
  assign q = tmp;
endmodule
```

2.3.12 Following is the Verilog code for a 4-bit unsigned down counter with synchronous set.

```
module counter (clk, s, q);
input      clk, s;
output [3:0] q;
reg   [3:0] tmp;
always @(posedge clk)
begin
  if (s)
    tmp <= 4'b1111;
  else
    tmp <= tmp - 1'b1;
end
  assign q = tmp;
endmodule
```

2.3.13 Following is the Verilog code for a 4-bit unsigned up counter with an asynchronous load from the primary input.

```
module counter (clk, load, d, q);
```

```
input       clk, load;
input  [3:0] d;
output [3:0] q;
reg    [3:0] tmp;
always @(posedge clk or posedge load)
begin
  if (load)
    tmp <= d;
  else
    tmp <= tmp + 1'b1;
end
  assign q = tmp;
endmodule
```

2.3.14 Following is the Verilog code for a 4-bit unsigned up counter with a synchronous load with a constant.

```
    module counter (clk, sload, q);
    input       clk, sload;
    output [3:0] q;
    reg    [3:0] tmp;
    always @(posedge clk)
    begin
      if (sload)
     tmp <= 4'b1010;
      else
        tmp <= tmp + 1'b1;
    end
      assign q = tmp;
endmodule
```

2.3.15 Following is the Verilog code for a 4-bit unsigned up counter with an asynchronous clear and a clock enable.

```
module counter (clk, clr, ce, q);
input      clk, clr, ce;
output [3:0] q;
reg    [3:0] tmp;
always @(posedge clk or posedge clr)
begin
  if (clr)
    tmp <= 4'b0000;
  else if (ce)
    tmp <= tmp + 1'b1;
end
  assign q = tmp;
endmodule
```

2.3.16 Following is the Verilog code for a 4-bit unsigned up/down counter with an asynchronous clear.

```
module counter (clk, clr, up_down, q);
input      clk, clr, up_down;
output [3:0] q;
reg    [3:0] tmp;
always @(posedge clk or posedge clr)
begin
  if (clr)
    tmp <= 4'b0000;
  else if (up_down)
    tmp <= tmp + 1'b1;
  else
    tmp <= tmp - 1'b1;
end
  assign q = tmp;
```

```
endmodule
```

2.3.17 Following is the Verilog code for a 4-bit signed up counter with an asynchronous reset.

```
module counter (clk, clr, q);
input              clk, clr;
output signed [3:0] q;
reg    signed [3:0] tmp;
always @ (posedge clk or posedge clr)
begin
  if (clr)
    tmp <= 4'b0000;
  else
    tmp <= tmp + 1'b1;
end
  assign q = tmp;
endmodule
```

2.3.18 Following is the Verilog code for a 4-bit signed up counter with an asynchronous reset and a modulo maximum.

```
module counter (clk, clr, q);
parameter MAX_SQRT = 4, MAX = (MAX_SQRT*MAX_SQRT);
input               clk, clr;
output [MAX_SQRT-1:0] q;
reg    [MAX_SQRT-1:0] cnt;
always @ (posedge clk or posedge clr)
begin
  if (clr)
    cnt <= 0;
  else
```

```
    cnt <= (cnt + 1) %MAX;
end
  assign q = cnt;
endmodule
```

2.3.19 Following is the Verilog code for a 4-bit unsigned up accumulator with an asynchronous clear.

```
module accum (clk, clr, d, q);
input        clk, clr;
input  [3:0] d;
output [3:0] q;
reg    [3:0] tmp;
always @(posedge clk or posedge clr)
begin
  if (clr)
    tmp <= 4'b0000;
  else
    tmp <= tmp + d;
end
  assign q = tmp;
endmodule
```

2.3.20 Following is the Verilog code for an 8-bit shift-left register with a positive-edge clock, serial in and serial out.

```
module shift (clk, si, so);
input        clk,si;
output       so;
reg    [7:0] tmp;
always @(posedge clk)
begin
```

```
        tmp    <= tmp << 1;
        tmp[0] <= si;
      end
        assign so = tmp[7];
  endmodule
```

2.3.21 Following is the Verilog code for an 8-bit shift-left register with a negative-edge clock, a clock enable, a serial in and a serial out.

```
    module shift (clk, ce, si, so);
    input       clk, si, ce;
    output       so;
    reg    [7:0] tmp;
    always @(negedge clk)
    begin
      if (ce) begin
        tmp    <= tmp << 1;
        tmp[0] <= si;
      end
    end
      assign so = tmp[7];
  endmodule
```

2.3.22 Following is the Verilog code for an 8-bit shift-left register with a positive-edge clock, asynchronous clear, serial in and serial out.

```
    module shift (clk, clr, si, so);
    input       clk, si, clr;
    output       so;
    reg    [7:0] tmp;
    always @(posedge clk or posedge clr)
    begin
```

```
      if (clr)
        tmp <= 8'b00000000;
      else
        tmp <= {tmp[6:0], si};
    end
      assign so = tmp[7];
endmodule
```

2.3.23 Following is the Verilog code for an 8-bit shift-left register with a positive-edge clock, a synchronous set, a serial in and a serial out.

```
    module shift (clk, s, si, so);
    input       clk, si, s;
    output       so;
    reg    [7:0] tmp;
    always @(posedge clk)
    begin
      if (s)
        tmp <= 8'b11111111;
      else
        tmp <= {tmp[6:0], si};
    end
      assign so = tmp[7];
endmodule
```

2.3.24 Following is the Verilog code for an 8-bit shift-left register with a positive-edge clock, a serial in and a parallel out.

```
module shift (clk, si, po);
input       clk, si;
output [7:0] po;
reg    [7:0] tmp;
```

```
always @(posedge clk)
begin
  tmp <= {tmp[6:0], si};
end
  assign po = tmp;
endmodule
```

2.3.25 Following is the Verilog code for an 8-bit shift-left register with a positive-edge clock, an asynchronous parallel load, a serial in and a serial out.

```
   module shift (clk, load, si, d, so);
   input       clk, si, load;
   input  [7:0] d;
   output      so;
   reg    [7:0] tmp;
   always @(posedge clk or posedge load)
   begin
     if (load)
    tmp <= d;
     else
       tmp <= {tmp[6:0], si};
   end
     assign so = tmp[7];
endmodule
```

2.3.26 Following is the Verilog code for an 8-bit shift-left register with a positive-edge clock, a synchronous parallel load, a serial in and a serial out.

```
module shift (clk, sload, si, d, so);
input       clk, si, sload;
input  [7:0] d;
output      so;
```

```
reg    [7:0] tmp;
always @(posedge clk)
begin
  if (sload)
    tmp <= d;
  else
    tmp <= {tmp[6:0], si};
end
  assign so = tmp[7];
endmodule
```

2.3.27 Following is the Verilog code for an 8-bit shift-left/shift-right register with a positive-edge clock, a serial in and a serial out.

```
    module shift (clk, si, left_right, po);
    input        clk, si, left_right;
    output       po;
    reg    [7:0] tmp;
    always @(posedge clk)
    begin
      if (left_right == 1'b0)
        tmp <= {tmp[6:0], si};
      else
        tmp <= {si, tmp[7:1]};
    end
      assign po = tmp;
endmodule
```

2.3.28 Following is the Verilog code for a 4-to-1 1-bit MUX using an If statement.

```
module mux (a, b, c, d, s, o);
```

```
    input       a,b,c,d;
    input  [1:0] s;
    output      o;
    reg         o;
    always @(a or b or c or d or s)
    begin
      if (s == 2'b00)
        o = a;
      else if (s == 2'b01)
        o = b;
      else if (s == 2'b10)
        o = c;
      else
        o = d;
    end
  endmodule
```

2.3.29 Following is the Verilog Code for a 4-to-1 1-bit MUX using a Case statement.

```
    module mux (a, b, c, d, s, o);
    input       a, b, c, d;
    input  [1:0] s;
    output      o;
    reg         o;
    always @(a or b or c or d or s)
    begin
      case (s)
        2'b00   : o = a;
        2'b01   : o = b;
        2'b10   : o = c;
        default : o = d;
      endcase
```

```
   end
endmodule
```

2.3.30 Following is the Verilog code for a 3-to-1 1-bit MUX with a 1-bit latch.

```
module mux (a, b, c, d, s, o);
input        a, b, c, d;
input  [1:0] s;
output       o;
reg          o;
always @(a or b or c or d or s)
begin
  if (s == 2'b00)
    o = a;
  else if (s == 2'b01)
    o = b;
  else if (s == 2'b10)
    o = c;
end
endmodule
```

2.3.31 Following is the Verilog code for a 1-of-8 decoder.

```
module mux (sel, res);
input  [2:0] sel;
output [7:0] res;
reg    [7:0] res;
always @(sel or res)
begin
  case (sel)
    3'b000  : res = 8'b00000001;
```

```
        3'b001  : res = 8'b00000010;
        3'b010  : res = 8'b00000100;
        3'b011  : res = 8'b00001000;
        3'b100  : res = 8'b00010000;
        3'b101  : res = 8'b00100000;
        3'b110  : res = 8'b01000000;
        default : res = 8'b10000000;
      endcase
    end
    endmodule
```

2.3.32 Following Verilog code leads to the inference of a 1-of-8 decoder.

```
      module mux (sel, res);
      input  [2:0] sel;
      output [7:0] res;
      reg    [7:0] res;
      always @(sel or res) begin
        case (sel)
          3'b000  : res = 8'b00000001;
          3'b001  : res = 8'b00000010;
          3'b010  : res = 8'b00000100;
          3'b011  : res = 8'b00001000;
          3'b100  : res = 8'b00010000;
          3'b101  : res = 8'b00100000;
          // 110 and 111 selector values are unused
          default : res = 8'bxxxxxxxx;
        endcase
      end
    endmodule
```

2.3.33 Following is the Verilog code for a 3-bit 1-of-9 Priority Encoder.

```
module priority (sel, code);
input  [7:0] sel;
output [2:0] code;
reg    [2:0] code;
always @(sel)
begin
  if (sel[0])
    code = 3'b000;
  else if (sel[1])
    code = 3'b001;
  else if (sel[2])
    code = 3'b010;
  else if (sel[3])
    code = 3'b011;
  else if (sel[4])
    code = 3'b100;
  else if (sel[5])
    code = 3'b101;
  else if (sel[6])
    code = 3'b110;
  else if (sel[7])
    code = 3'b111;
  else
    code = 3'bxxx;
end
endmodule
```

2.3.34 Following is the Verilog code for a logical shifter.

```
module lshift (di, sel, so);
```

```
input  [7:0] di;
   input  [1:0] sel;
   output [7:0] so;
   reg    [7:0] so;
   always @(di or sel)
   begin
     case (sel)
       2’b00   : so = di;
       2’b01   : so = di << 1;
       2’b10   : so = di << 2;
       default : so = di << 3;
     endcase
   end
endmodule
```

2.3.35 Following is the Verilog code for an unsigned 8-bit adder with carry in.

```
   module adder(a, b, ci, sum);
   input  [7:0] a;
   input  [7:0] b;
   input        ci;
   output [7:0] sum;

     assign sum = a + b + ci;

endmodule
```

2.3.36 Following is the Verilog code for an unsigned 8-bit adder with carry out.

```
   module adder(a, b, sum, co);
   input  [7:0] a;
   input  [7:0] b;
   output [7:0] sum;
```

```
    output       co;
    wire   [8:0] tmp;

      assign tmp = a + b;
      assign sum = tmp [7:0];
      assign co  = tmp [8];

endmodule
```

2.3.37 Following is the Verilog code for an unsigned 8-bit adder with carry in and carry out.

```
module adder(a, b, ci, sum, co);
input        ci;
input  [7:0] a;
input  [7:0] b;
output [7:0] sum;
output       co;
wire   [8:0] tmp;

  assign tmp = a + b + ci;
  assign sum = tmp [7:0];
  assign co  = tmp [8];

endmodule
```

2.3.38 Following is the Verilog code for an unsigned 8-bit adder/subtractor.

```
    module addsub(a, b, oper, res);
    input        oper;
    input  [7:0] a;
    input  [7:0] b;
    output [7:0] res;
    reg    [7:0] res;
    always @(a or b or oper)
    begin
      if (oper == 1'b0)
        res = a + b;
      else
        res = a - b;
  end
  endmodule
```

2.3.39 Following is the Verilog code for an unsigned 8-bit greater or equal comparator.

```
    module compar(a, b, cmp);
    input  [7:0] a;
    input  [7:0] b;
    output       cmp;

      assign cmp = (a >= b) ?  1'b1 : 1'b0;

  endmodule
```

2.3.40 Following is the Verilog code for an unsigned 8x4-bit multiplier.

```
  module compar(a, b, res);
  input  [7:0]  a;
  input  [3:0]  b;
  output [11:0] res;

    assign res = a * b;
```

```
endmodule
```

2.3.41 Following Verilog template shows the multiplication operation placed outside the always block and the pipeline stages represented as single registers.

```
module mult(clk, a, b, mult);
input       clk;
input  [17:0] a;
input  [17:0] b;
output [35:0] mult;
reg    [35:0] mult;
reg    [17:0] a_in, b_in;
wire   [35:0] mult_res;
reg    [35:0] pipe_1, pipe_2, pipe_3;

  assign mult_res = a_in * b_in;

always @(posedge clk)
begin
     a_in   <= a;
  b_in   <= b;
  pipe_1 <= mult_res;
  pipe_2 <= pipe_1;
  pipe_3 <= pipe_2;
  mult   <= pipe_3;
end
endmodule
```

2.3.42 Following Verilog template shows the multiplication operation placed inside the always block and the pipeline stages are represented as single registers.

```
module mult(clk, a, b, mult);
```

```
input        clk;
input  [17:0] a;
input  [17:0] b;
output [35:0] mult;
reg    [35:0] mult;
reg    [17:0] a_in, b_in;
reg    [35:0] mult_res;
reg    [35:0] pipe_2, pipe_3;
always @(posedge clk)
begin
     a_in     <= a;
  b_in     <= b;
  mult_res <= a_in * b_in;
  pipe_2   <= mult_res;
  pipe_3   <= pipe_2;
  mult     <= pipe_3;
end
endmodule
```

2.3.43 Following Verilog template shows the multiplication operation placed outside the always block and the pipeline stages represented as single registers.

```
module mult(clk, a, b, mult);
input        clk;
input  [17:0] a;
input  [17:0] b;
output [35:0] mult;
reg    [35:0] mult;
reg    [17:0] a_in, b_in;
wire   [35:0] mult_res;
reg    [35:0] pipe_1, pipe_2, pipe_3;

  assign mult_res = a_in * b_in;
  always @(posedge clk)
  begin
    a_in   <= a;
    b_in   <= b;
    pipe_1 <= mult_res;
    pipe_2 <= pipe_1;
    pipe_3 <= pipe_2;
    mult   <= pipe_3;
  end
endmodule
```

2.3.44 Following Verilog template shows the multiplication operation placed inside the always block and the pipeline stages are represented as single registers.

```
module mult(clk, a, b, mult);
input         clk;
input  [17:0] a;
input  [17:0] b;
output [35:0] mult;
reg    [35:0] mult;
reg    [17:0] a_in, b_in;
reg    [35:0] mult_res;
reg    [35:0] pipe_2, pipe_3;
always @(posedge clk)
begin
  a_in     <= a;
  b_in     <= b;
  mult_res <= a_in * b_in;
  pipe_2   <= mult_res;
  pipe_3   <= pipe_2;
  mult     <= pipe_3;
end
```

```
endmodule
```

2.3.45 Following Verilog template shows the multiplication operation placed outside the always block and the pipeline stages represented as shift registers.

```
module mult3(clk, a, b, mult);
input        clk;
input  [17:0] a;
input  [17:0] b;
output [35:0] mult;
reg    [35:0] mult;
reg    [17:0] a_in, b_in;
wire   [35:0] mult_res;
reg    [35:0] pipe_regs [3:0];

  assign mult_res = a_in * b_in;

always @(posedge clk)
begin
  a_in <= a;
  b_in <= b;
  {pipe_regs[3],pipe_regs[2],pipe_regs[1],pipe_regs[0]} <=
{mult, pipe_regs[3],pipe_regs[2],pipe_regs[1]};
end
endmodule
```

2.3.46 Following templates to implement Multiplier Adder with 2 Register Levels on Multiplier Inputs in Verilog.

```
module mvl_multaddsub1(clk, a, b, c, res);
input        clk;
input  [07:0] a;
input  [07:0] b;
input  [07:0] c;
output [15:0] res;
reg    [07:0] a_reg1, a_reg2, b_reg1, b_reg2;
wire   [15:0] multaddsub;
always @(posedge clk)
begin
  a_reg1 <= a;
  a_reg2 <= a_reg1;
  b_reg1 <= b;
  b_reg2 <= b_reg1;
end
  assign multaddsub = a_reg2 * b_reg2 + c;
  assign res = multaddsub;
endmodule
```

2.3.47 Following is the Verilog code for resource sharing.

```
module addsub(a, b, c, oper, res);
input        oper;
input  [7:0] a;
input  [7:0] b;
input  [7:0] c;
output [7:0] res;
reg    [7:0] res;
always @(a or b or c or oper)
begin
  if (oper == 1'b0)
    res = a + b;
  else
    res = a - c;
end
```

```
endmodule
```

2.3.48 Following templates show a single-port RAM in read-first mode.

```
module raminfr (clk, en, we, addr, di, do);
input       clk;
input       we;
input       en;
input  [4:0] addr;
input  [3:0] di;
output [3:0] do;
reg    [3:0] RAM [31:0];
reg    [3:0] do;
always @(posedge clk)
begin
  if (en)
    begin
    if (we)
         RAM[addr] <= di;

      do <= RAM[addr];
  end
end
endmodule
```

2.3.49 Following templates show a single-port RAM in write-first mode.

```
module raminfr (clk, we, en, addr, di, do);
input       clk;
input       we;
input       en;
```

```
    input  [4:0] addr;
    input  [3:0] di;
    output [3:0] do;
    reg    [3:0] RAM [31:0];
    reg    [4:0] read_addr;
    always @(posedge clk)
    begin
      if (en) begin
        if (we)
            RAM[addr] <= di;
      read_addr <= addr;
      end
    end
      assign do = RAM[read_addr];
  endmodule
```

2.3.50 Following templates show a single-port RAM in no-change mode.

```
    module raminfr (clk, we, en, addr, di, do);
    input        clk;
    input        we;
    input        en;
    input  [4:0] addr;
    input  [3:0] di;
    output [3:0] do;
    reg    [3:0] RAM [31:0];
    reg    [3:0] do;
    always @(posedge clk)
    begin
      if (en) begin
        if (we)
            RAM[addr] <= di;
            else
                    do <= RAM[addr];
          end
       end
endmodule
```

2.3.51 Following is the Verilog code for a single-port RAM with asynchronous read.

```
module raminfr (clk, we, a, di, do);
input        clk;
input        we;
input  [4:0] a;
input  [3:0] di;
output [3:0] do;
reg    [3:0] ram [31:0];
always @(posedge clk)
begin
       if (we)
      ram[a] <= di;
end
   assign do = ram[a];
endmodule
```

2.3.52 Following is the Verilog code for a single-port RAM with "false" synchronous read.

```
module raminfr (clk, we, a, di, do);
input        clk;
input        we;
input  [4:0] a;
input  [3:0] di;
output [3:0] do;
```

```
    reg    [3:0] ram [31:0];
    reg    [3:0] do;
    always @(posedge clk)
    begin
      if (we)
        ram[a] <= di;
      do <= ram[a];
    end
endmodule
```

2.3.53 Following is the Verilog code for a single-port RAM with synchronous read (read through).

```
    module raminfr (clk, we, a, di, do);
    input        clk;
    input        we;
    input  [4:0] a;
    input  [3:0] di;
    output [3:0] do;
    reg    [3:0] ram [31:0];
    reg    [4:0] read_a;
    always @(posedge clk)
    begin
      if (we)
        ram[a] <= di;
      read_a <= a;
    end
      assign do = ram[read_a];
endmodule
```

2.3.54 Following is the Verilog code for a single-port block RAM with enable.

```
module raminfr (clk, en, we, a, di, do);
input       clk;
input       en;
input       we;
input  [4:0] a;
input  [3:0] di;
output [3:0] do;
reg    [3:0] ram [31:0];
reg    [4:0] read_a;
always @(posedge clk)
begin
  if (en) begin
    if (we)
      ram[a] <= di;
    read_a <= a;
  end
end
  assign do = ram[read_a];
endmodule
```

2.3.55 Following is the Verilog code for a dual-port RAM with asynchronous read.

```
module raminfr (clk, we, a, dpra, di, spo, dpo);
input       clk;
input       we;
input  [4:0] a;
input  [4:0] dpra;
input  [3:0] di;
output [3:0] spo;
output [3:0] dpo;
reg    [3:0] ram [31:0];
```

```
    always @(posedge clk)
    begin
      if (we)
        ram[a] <= di;
    end
      assign spo = ram[a];
      assign dpo = ram[dpra];
endmodule
```

2.3.56 Following is the Verilog code for a dual-port RAM with false synchronous read.

```
module raminfr (clk, we, a, dpra, di, spo, dpo);
input       clk;
input       we;
input  [4:0] a;
input  [4:0] dpra;
input  [3:0] di;
output [3:0] spo;
output [3:0] dpo;
reg    [3:0] ram [31:0];
reg    [3:0] spo;
reg    [3:0] dpo;
    always @(posedge clk)
begin
  if (we)
    ram[a] <= di;

  spo = ram[a];
  dpo = ram[dpra];
end
endmodule
```

2.3.57 Following is the Verilog code for a dual-port RAM with synchronous read (read through).

```
module raminfr (clk, we, a, dpra, di, spo, dpo);
input       clk;
input       we;
input  [4:0] a;
input  [4:0] dpra;
input  [3:0] di;
output [3:0] spo;
output [3:0] dpo;
reg    [3:0] ram [31:0];
reg    [4:0] read_a;
reg    [4:0] read_dpra;
always @(posedge clk)
begin
  if (we)
    ram[a] <= di;
  read_a <= a;
  read_dpra <= dpra;
end
  assign spo = ram[read_a];
  assign dpo = ram[read_dpra];
endmodule
```

2.3.58 Following is the Verilog code for a dual-port RAM with enable on each port.

```
module raminfr (clk, ena, enb, wea, addra, addrb, dia, doa, dob);
input       clk, ena, enb, wea;
input  [4:0] addra, addrb;
input  [3:0] dia;
```

```
  output [3:0] doa, dob;
  reg    [3:0] ram [31:0];
  reg    [4:0] read_addra, read_addrb;
  always @(posedge clk)
  begin
    if (ena) begin
      if (wea) begin
           ram[addra] <= dia;
      end
    end
  end

  always @(posedge clk)
  begin
    if (enb) begin
      read_addrb <= addrb;
    end
  end
    assign doa = ram[read_addra];
    assign dob = ram[read_addrb];
endmodule
```

2.3.59 Following is Verilog code for a ROM with registered output.

```
module rominfr (clk, en, addr, data);
input       clk;
input       en;
input [4:0] addr;
output reg [3:0] data;
  always @(posedge clk)
begin
  if (en)
```

```
    case(addr)
      4'b0000: data <= 4'b0010;
      4'b0001: data <= 4'b0010;
      4'b0010: data <= 4'b1110;
      4'b0011: data <= 4'b0010;
      4'b0100: data <= 4'b0100;
      4'b0101: data <= 4'b1010;
      4'b0110: data <= 4'b1100;
      4'b0111: data <= 4'b0000;
      4'b1000: data <= 4'b1010;
      4'b1001: data <= 4'b0010;
      4'b1010: data <= 4'b1110;
      4'b1011: data <= 4'b0010;
      4'b1100: data <= 4'b0100;
      4'b1101: data <= 4'b1010;
      4'b1110: data <= 4'b1100;
      4'b1111: data <= 4'b0000;
      default: data <= 4'bXXXX;
    endcase
end
endmodule
```

2.3.60 Following is Verilog code for a ROM with registered address.

```
module rominfr (clk, en, addr, data);
input       clk;
input       en;
input [4:0] addr;
output reg [3:0] data;
reg   [4:0] raddr;
always @(posedge clk)
begin
```

```
    if (en)
      raddr <= addr;
  end

  always @(raddr)
  begin
    if (en)
      case(raddr)
           4'b0000: data = 4'b0010;
           4'b0001: data = 4'b0010;
           4'b0010: data = 4'b1110;
           4'b0011: data = 4'b0010;
           4'b0100: data = 4'b0100;
           4'b0101: data = 4'b1010;
           4'b0110: data = 4'b1100;
           4'b0111: data = 4'b0000;
           4'b1000: data = 4'b1010;
           4'b1001: data = 4'b0010;
           4'b1010: data = 4'b1110;
           4'b1011: data = 4'b0010;
           4'b1100: data = 4'b0100;
           4'b1101: data = 4'b1010;
           4'b1110: data = 4'b1100;
           4'b1111: data = 4'b0000;
           default: data = 4'bXXXX;
      endcase
  end
endmodule
```

2.3.61 Following is the Verilog code for an FSM with a single process.

```
module fsm (clk, reset, x1, outp);
```

```
input        clk, reset, x1;
output       outp;
reg          outp;
reg    [1:0] state;
parameter s1 = 2'b00; parameter s2 = 2'b01;
parameter s3 = 2'b10; parameter s4 = 2'b11;
always @(posedge clk or posedge reset)
begin
  if (reset) begin
    state <= s1; outp <= 1'b1;
  end
  else begin
    case (state)
         s1: begin
                if (x1 == 1'b1) begin
                  state <= s2;
           outp  <= 1'b1;
                end
                else begin
                  state <= s3;
           outp  <= 1'b1;
                end
           end
         s2: begin
                state <= s4;
        outp  <= 1'b0;
           end
         s3: begin
                state <= s4;
        outp  <= 1'b0;
           end
         s4: begin
                state <= s1;
        outp  <= 1'b1;
```

```
                end
            endcase
          end
        end
      endmodule
```

2.3.62 Following is the Verilog code for an FSM with two processes.

```
module fsm (clk, reset, x1, outp);
input       clk, reset, x1;
output      outp;
reg         outp;
reg    [1:0] state;
parameter s1 = 2'b00; parameter s2 = 2'b01;
parameter s3 = 2'b10; parameter s4 = 2'b11;
always @(posedge clk or posedge reset)
begin
  if (reset)
    state <= s1;
  else begin
    case (state)
         s1: if (x1 == 1'b1)
                 state <= s2;
             else
                 state <= s3;
         s2: state <= s4;
         s3: state <= s4;
         s4: state <= s1;
    endcase
  end
end
always @(state) begin
```

```
    case (state)
      s1: outp = 1'b1;
      s2: outp = 1'b1;
      s3: outp = 1'b0;
      s4: outp = 1'b0;
    endcase
  end
endmodule
```

2.3.63 Following is the Verilog code for an FSM with three processes.

```
module fsm (clk, reset, x1, outp);
input       clk, reset, x1;
output      outp;
reg         outp;
reg    [1:0] state;
reg    [1:0] next_state;
parameter s1 = 2'b00; parameter s2 = 2'b01;
parameter s3 = 2'b10; parameter s4 = 2'b11;
always @(posedge clk or posedge reset)
begin
  if (reset)
    state <= s1;
  else
    state <= next_state;
end

always @(state or x1)
begin
  case (state)
    s1: if (x1 == 1'b1)
            next_state = s2;
```

```
                else
                   next_state = s3;
         s2: next_state = s4;
         s3: next_state = s4;
         s4: next_state = s1;
      endcase
   end
```

3 FIFO

3.1 Questions with answers

3.1.1 Given the following FIFO and rules, how deep does the FIFO need to be to prevent underflow or overflow?

RULES:

1) frequency(clk_A) = frequency(clk_B) / 4

2) period(en_B) = period(clk_A) * 100

3) duty_cycle(en_B) = 25%

Assume clk_B = 100MHz (10ns)

From (1), clk_A = 25MHz (40ns)

From (2), period(en_B) = 40ns * 400 = 4000ns, but we only output for 1000ns,due to (3), so 3000ns of the enable we are doing no output work. Therefore, FIFO size = 3000ns/40ns = 75 entries.

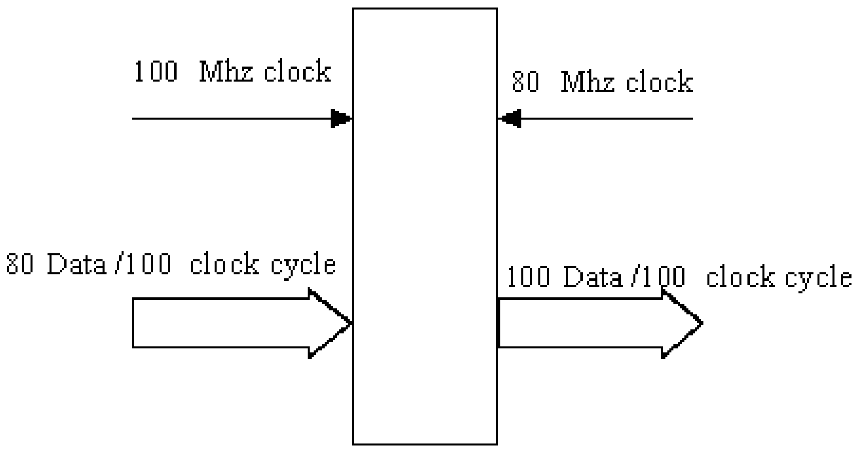
3.1.2 What is difference between RAM and FIFO?

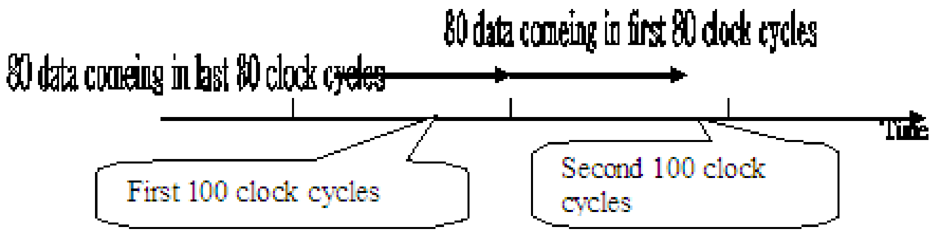
FIFO does not have address lines

RAM is used for storage purpose where as fifo is used for synchronization purpose i.e. when two peripherals are working in different clock domains then we will go for fifo.

3.1.3 An FIFO which clocks data in at 100 mhz and clocks data out at 100mhz. On the input there is only 80 data in any order during each 100 clocks. In other words, a 100 input clock will carry only 80 data and the other twenty clocks carry no data (data is scattered in any order). How big the FIFO needs to be to avoid data over/under-run.

The worst case is when 160 data come continuously with 160 clocks as shown in the bottom figure.




80 data comeing in first 80 clock cycles
80 data comeing in last 80 clock cycles
Time
First 100 clock cycles
Second 100 clock cycles

www.asichowto.com

From above figure we know that in the first 100 clock cycles only 0.8x80=64 data can get out of the FIFO. There are 16 data need to stay at the FIFO. During the next 100 clock cycles in the first 80 clock cycles the input side will get 80 data in. And at the output still only 64 data can be read out the FIFO. So again another 16 data need to stay at the FIFO. SO the FIFO depth is 32.

3.1.4 Consider the case of a FIFO where the write clock frequency is 100 MHz and 80 words are written into the FIFO in 100 clocks while the read clock frequency is 80 MHz and 8 words are read out every 10 clocks. There is no feedback mechanism to throttle the writes to the FIFO.

In the worst case, the FIFO will write 80 words in a burst into the FIFO in 800 ns. In the same time, the read side can read only ~51 words ((800/125) * 8) in that same time period. In the remaining 200 ns, only ~13 words ((200/125) * 8) can be read out of the FIFO leaving 16 words on the floor.

So the FIFO will need to be of infinite depth to make this design work!

3.1.5 Consider the case of a FIFO where the write clock frequency is 100 MHz and 50 words are written into the FIFO in 100 clocks while the read clock frequency is 50 MHz and one word is read out every clock.

In the worst case scenario, the 100 words are written into the FIFO as a burst in 1000 ns. In the same time duration, the read side can read only 50 words out of the FIFO.

The remaining 50 words are read out of the FIFO in the 50 idle write clocks. So the depth of the FIFO should be atleast 25 (+ synchronizer latency) = ~28.

The FIFO depth is calculated as

Depth = Burst_size * { 1 - (Frd/(Fwr * Idle_cycles)) }

3.1.6 We need to sample an input or output something at different rates, but I need to vary the rate? What's a clean way to do this?

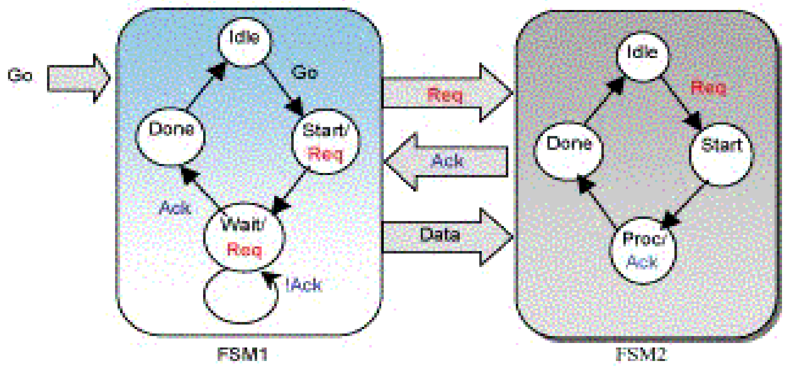
Many, many problems have this sort of variable rate requirement, yet we are usually constrained with a constant clock frequency. One trick is to implement a digital NCO (Numerically Controlled Oscillator). An NCO is actually very simple and, while it is most naturally understood as hardware, it also can be constructed in software. The NCO, quite simply, is an accumulator where you keep adding a fixed value on every clock (e.g. at a constant clock frequency). When the NCO "wraps", you sample your input or do your action. By adjusting the value added to the accumulator each clock; you finely tune the AVERAGE frequency of that wrap event. Now - you may have realized that the wrapping event may have lots of jitter on it. True, but you may use the wrap to increment yet another counter where each additional Divide-by-2 bit reduces this jitter. The DDS is a related technique. I have two examples showing both an NCOs and a DDS in my File Archive. This is tricky to grasp at first, but tremendously powerful once you have it in your bag of tricks. NCOs also relate to digital PLLs, Timing Recovery, TDMA and other "variable rate" phenomena

3.1.7 Clock domain crossing

The following section explains clock domain interfacing

One of the biggest challenges of system-on-chip (SOC) designs is that different blocks operate on independent clocks. Integrating these blocks via the processor bus, memory ports, peripheral busses, and other interfaces can be troublesome because unpredictable behavior can result when the asynchronous interfaces are not properly synchronized

A very common and robust method for synchronizing multiple data signals is a handshake technique as shown in diagram below This is popular because the handshake technique can easily manage changes in clock frequencies, while minimizing latency at the crossing. However, handshake logic is significantly more complex than standard synchronization structures.



FSM1(Transmitter) asserts the req (request) signal, asking the receiver to accept the data on the data bus. FSM2(Receiver) generally a slow module asserts the ack (acknowledge) signal, signifying that it has accepted the data.

It has loop holes: when system Receiver samples the systems Transmitter req line and Transmitter samples system Receiver ack line, they have done it with respect to their internal clock, so there will be setup and hold time violation. To avoid this we go for double or triple stage synchronizers, which increase the MTBF and thus are immune to metastability to a good extent. The figure below shows how this is done.

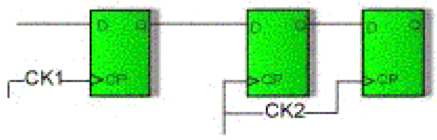


Figure 1a — Single-bit metastability sync

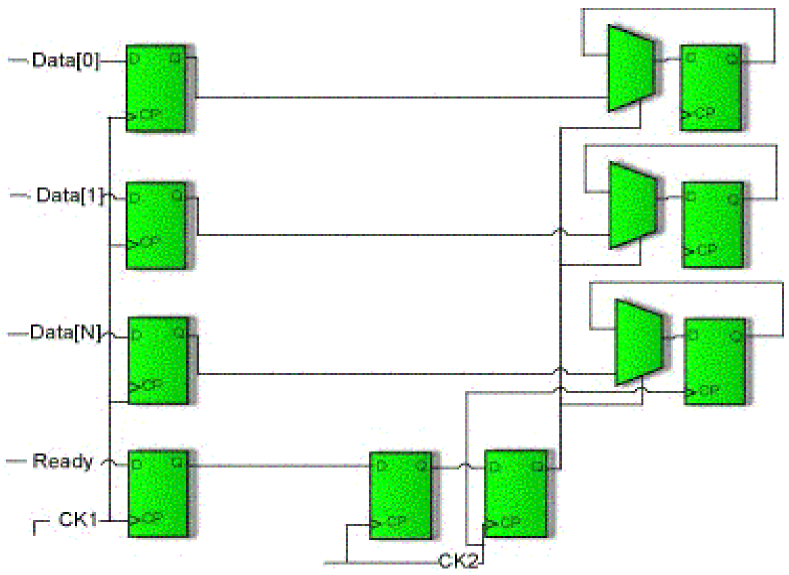


Figure 1b — Multi-bit sync

3.2 Questions without answers

3.2.1 Asynchronous FIFO design

Design a FIFO 1 byte wide and 13 words deep. The FIFO is interfacing 2 blocks with different clocks. On the rising edge of clk the FIFO stores data and increments wptr. On the rising edge of clkb the data is put on the b-output,the rptr points to the next data to be read.

If the FIFO is empty, the b-output data is not valid. When the FIFO is full the existing data should not be overriden.

When rst_N is asserted, the FIFO pointers are asynchronously reset

```
module fifo1 (full,empty,clk,clkb,ain,bout,rst_N)

output [7:0] bout;

input [7:0] ain;

input clk,clkb,rst_N;

output empty, full;

reg [3:0] wptr, rptr;

...

endmodule
```

0
1
2
3
4
5
6
7
8
9
10
11
12

Multiple clocks add complexity to this design. We need to define conditions for Empty and Full signals, take care of WR and RD pointers. Here is one of the solutions.

Empty and Full flags:

```
assign empty=((wptr == rptr) && (w_flag == r_flag);
assign full=((wptr == rptr) && (w_flag == ~r_flag);
```

where w_flag is set when wptr =12 (end of FIFO). After that wptr is reset to 0. The same for r_flag and rptr.

Pointer handling:

```
if (wptr == 12) {w_flag,wptr} <= {~w_flag,4'b0000};
else wptr <= wptr+1;
if (rptr == 12) {r_flag,rptr} <= {~r_flag,4'b0000};
else rptr <= rptr+1;
```

3.2.2 How to synchronize control signals and data between two different clock domains?

3.2.3 In a system there are two modules A and B. A is operating at 25 MHz and B at 25 KHzFrom module A if a pulse of width equal to width of clock (1/25 Micro seconds) is sent, How ensure that the pulse will be correctly received at module B without using handshaking or Buffers like FIFO?

4 FPGA

4.1 Question with Answers

4.1.1 What is Synthesis?

Synthesis is the stage in the design flow which is concerned with translating your Verilog code into gates - and that's putting it very simply! First of all, the Verilog must be written in a particular way for the synthesis tool that you are using. Of course, a synthesis tool doesn't actually produce gates - it will output a netlist of the design that you have synthesised that represents the chip which can be fabricated through an ASIC or FPGA vendor.

4.1.2 What is Clock distribution network?

In a synchronous digital system, the clock signal is used to define a time reference for the movement of data within that system. The clock distribution network distributes the clock signal(s) from a common point to all the elements that need it. Since this function is vital to the operation of a synchronous system, much attention has been given to the characteristics of these clock signals and the electrical networks used in their distribution. Clock signals are often regarded as simple control signals; however, these signals have some very special characteristics and attributes.

Clock signals are typically loaded with the greatest fanout, travel over the greatest distances, and operate at the highest speeds of any signal, either control or data, within the entire synchronous system. Since the data signals are provided with a temporal reference by the clock signals, the clock waveforms must be particularly clean and sharp. Furthermore, these clock signals are particularly affected by technology scaling (see Moore's law), in that long global interconnect lines become significantly more resistive as line dimensions are decreased. This increased line resistance is one of the primary reasons for the increasing significance of clock distribution on synchronous performance. Finally, the control of any differences and uncertainty in the arrival times of the clock signals can severely limit the maximum performance of the entire system and create catastrophic race conditions in which an incorrect data signal may latch within a register. The clock distribution network often takes a significant fraction of the power consumed by a chip. Furthermore, significant power can be wasted in transitions within blocks, even when their output is not needed. These observations have lead to a power saving technique called clock gating, which involves adding logic gates to the clock distribution tree, so portions of the tree can be turned off when not needed.

4.1.3 What is Clock Gating ?

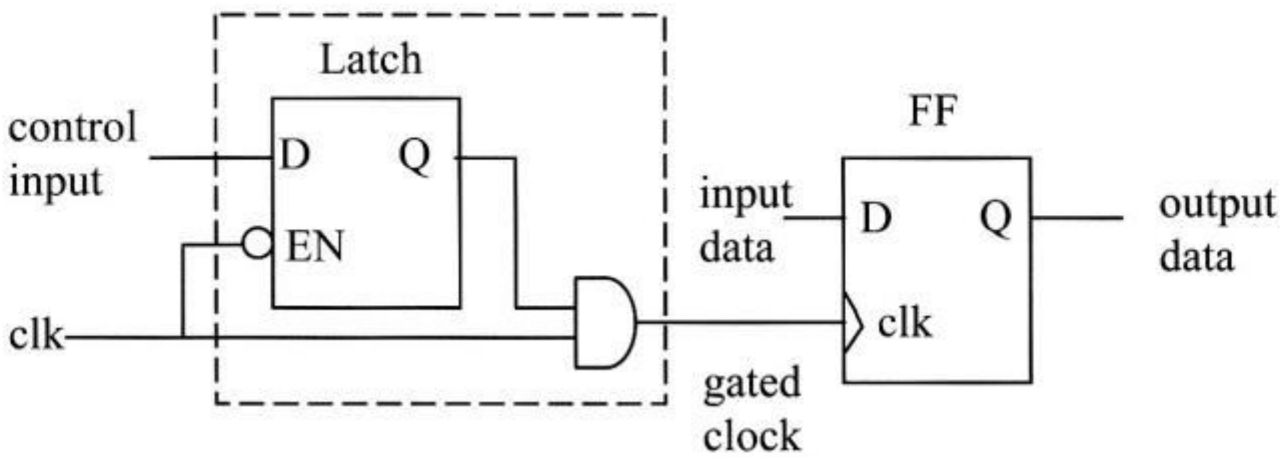
Clock gating is one of the power-saving techniques used on many synchronous circuits including the Pentium 4 processor. To save power, clock gating refers to adding additional logic to a circuit to prune the clock tree, thus disabling portions of the circuitry where flip flops do not change state. Although asynchronous circuits by definition do not have a "clock", the term "perfect clock gating" is used to illustrate how various clock gating techniques are simply approximations of the data-dependent behavior exhibited by asynchronous circuitry, and that as the granularity on which you gate the clock of a synchronous circuit approaches zero, the power consumption of that circuit approaches that of an asynchronous circuit.

4.1.4 Illustrate an example of clock gating to help in reduction of power.

Clock gating is a common mechanism to save power. This technique reduces the switching activity of the output of the FF by:

- ❖ Eliminating the need for reloading the same value in the register during multiple clock cycle.
- ❖ Reducing the clock network power dissipation.

The most common method of clock gating is through the use of a latch and a gate. The following figure illustrates the implementation of this mechanism:



When the clk is in its low phase, the latch is enabled. The control input, which actually decides whether to gate the clock or not, is now propagated through the clock to its Q output. Here, if the control input is high, the Q of the latch is high during the low phase, and remains so until the next low phase of the clk. This keeps the AND gate enabled. In the mean time, when the clk arrives, it gets propagated to the gated clock net. This happens cleanly, without any glitches, because the latch output is stable for sufficient time to meet the Flip-Flops setup requirements. When the control input goes low, it negates the AND gate and, hence, prevents the clk from being propagated to the gated clock net. This makes the gated clock net to be at 0 without any switching activity.

A simple Verilog code that illustrates the above logic is illustrated as follows. Note that the implementation of this strategy in large designs is best done through the synthesis tools without having to manually implement this strategy in the designs containing a large number of FFs.

```
module gated_ff (in1, cntrl_in, clk, reset_n, out1);

input  cntrl_in, in1, clk, reset_n;
output out1;

wire gated_clk;
reg  d_latch, out1;

always @(cntrl_in, clk) begin
  if (clk)
    d_latch <= cntrl_in;
end

assign gated_clk = d_latch & clk;

always @(posedge gated_clk or negedge reset_n) begin
  if (! reset_n)
    out1 <= 1'b0;
  else
    out1 <= in1;
end

endmodule
```

4.1.5 How to achieve 180-degree exact phase shift?

Never tell using inverter

a) dcm's an inbuilt resource in most of fpga can be configured to get 180 degree phase shift.

b) Bufgds that is differential signaling buffers which are also inbuilt resource of most of FPGA can be used.

4.1.6 What is significance of ras and cas in SDRAM?

SDRAM receives its address command in two address words.

It uses a multiplex scheme to save input pins. The first address word is latched into the DRAM chip with the row address strobe (RAS).

Following the RAS command is the column address strobe (CAS) for latching the second address word.Shortly after the RAS and CAS strobes, the stored data is valid for reading.

4.1.7 Difference between FPGA and CPLD?

FPGA:

- SRAM based technology.
- Segmented connection between elements.
- Usually used for complex logic circuits.
- Must be reprogrammed once the power is off.
- Costly

CPLD:

- Flash or EPROM based technology.
- Continuous connection between elements.
- Usually used for simpler or moderately complex logic circuits.
- Need not be reprogrammed once the power is off.
- Cheaper.

4.1.8 What is slice,clb,lut?

I am taking example of xc3s500 to answer this question

The Configurable Logic Blocks (CLBs) constitute the main logic resource for implementing synchronous as well as combinatorial circuits.

CLB are configurable logic blocks and can be configured to combo,ram or rom depending on coding style CLB consist of 4 slices and each slice consist of two 4-input LUT (look up table) F-LUT and G-LUT.

4.1.9 Can a clb configured as ram?

YES. The memory assignment is a clocked behavioral assignment, Reads from the memory are asynchronous, And all the address lines are shared by the read and write statements.

4.1.10 Suggest some ways to increase clock frequency?

- Check critical path and optimize it.
- Add more timing constraints (over constrain).
- Pipeline the architecture to the max possible extent keeping in mind latency req's.

4.1.11 FPGA design cycle

Synthesis.

The ISE™ software includes Xilinx® Synthesis Technology (XST), which synthesizes VHDL, Verilog, or mixed language designs to create Xilinx-specific netlist files known as NGC files.

Translate

The Translate process merges all of the input netlists and design constraints and outputs a Xilinx native generic database (NGD) file, which describes the logical design reduced to Xilinx primitives.

Map

The Map process maps the logic defined by an NGD file into FPGA elements, such as CLBs and IOBs. The output design is a native circuit description (NCD) file that physically represents the design mapped to the components in the Xilinx FPGA

Place and Route

The Place and Route process takes a mapped NCD file, places and routes the design, and produces an NCD file that is used as input for bit stream generation.

4.1.12 What is minimum and maximum frequency of dcm in spartan-3 series fpga?

Spartan series dcm's have a minimum frequency of 24 MHZ and a maximum of 248.

4.1.13 Tell me some of constraints you used and their purpose during your design?

There are lot of constraints and will vary for tool to tool ,I am listing some of Xilinx constraints

a) Translate on and Translate off: The Verilog code between Translate on and Translate off is ignored for synthesis.

b) CLOCK_SIGNAL: Is a synthesis constraint. In the case where a clock signal goes through combinatorial logic before being connected to the clock input of a flip-flop, XST cannot identify what input pin or internal net is the real clock signal. This constraint allows you to define the clock net.

c) XOR_COLLAPSE: Is synthesis constraint. It controls whether cascaded XORs should be collapsed into a single XOR.

For more constraints detailed description refer to constraint guide.

4.1.14 Suppose for a piece of code equivalent gate count is 600 and for another code equivalent gate count is 50,000 will the size of bitmap change?in other words will size of bitmap change it gate count change?

The size of bitmap is irrespective of resource utilization, it is always the same,for Spartan xc3s5000 it is 1.56MB and will never change.

4.1.15 What are different types of FPGA programming modes?what are you currently using ?how to change from one to another?

Before powering on the FPGA, configuration data is stored externally in a PROM or some other nonvolatile medium either on or off the board. After applying power, the configuration data is written to the FPGA using any of five different modes:

- Master Parallel
- Slave Parallel
- Master Serial
- Slave Serial
- Boundary Scan (JTAG).

The Master and Slave Parallel modes Mode selecting pins can be set to select the mode, refer data sheet for further details.

4.1.16 Tell me some of features of FPGA you are currently using?

I am taking example of xc3s5000 to answering the question .

Very low cost, high-performance logic solution for high-volume, consumer-oriented applications

- Densities as high as 74,880 logic cells

- Up to 784 I/O pins

- 622 Mb/s data transfer rate per I/O

- 18 single-ended signal standards

- 6 differential I/O standards including LVDS, RSDS

- Termination by Digitally Controlled Impedance

- Signal swing ranging from 1.14V to 3.45V

- Double Data Rate (DDR) support

• Logic resources

- Abundant logic cells with shift register capability

- Wide multiplexers

- Fast look-ahead carry logic

- Dedicated 18 x 18 multipliers

- Up to 1,872 Kbits of total block RAM

- Up to 520 Kbits of total distributed RAM

• Digital Clock Manager (up to four DCMs)

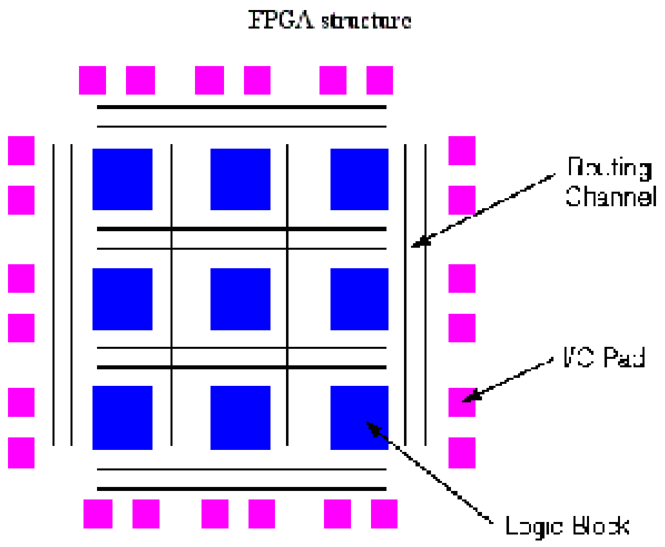
- Clock skew elimination

• Eight global clock lines and abundant routing

4.1.17 Can you explain what struck at zero means?

These stuck-at problems will appear in ASIC. Some times, the nodes will permanently tie to 1 or 0 because of some fault. To avoid that, we need to provide testability in RTL. If it is permanently 1 it is called stuck-at-1 If it is permanently 0 it is called stuck-at-0.

4.1.18 Can you draw general structure of fpga?



4.1.19 What is purpose of a constraint file what is its extension?

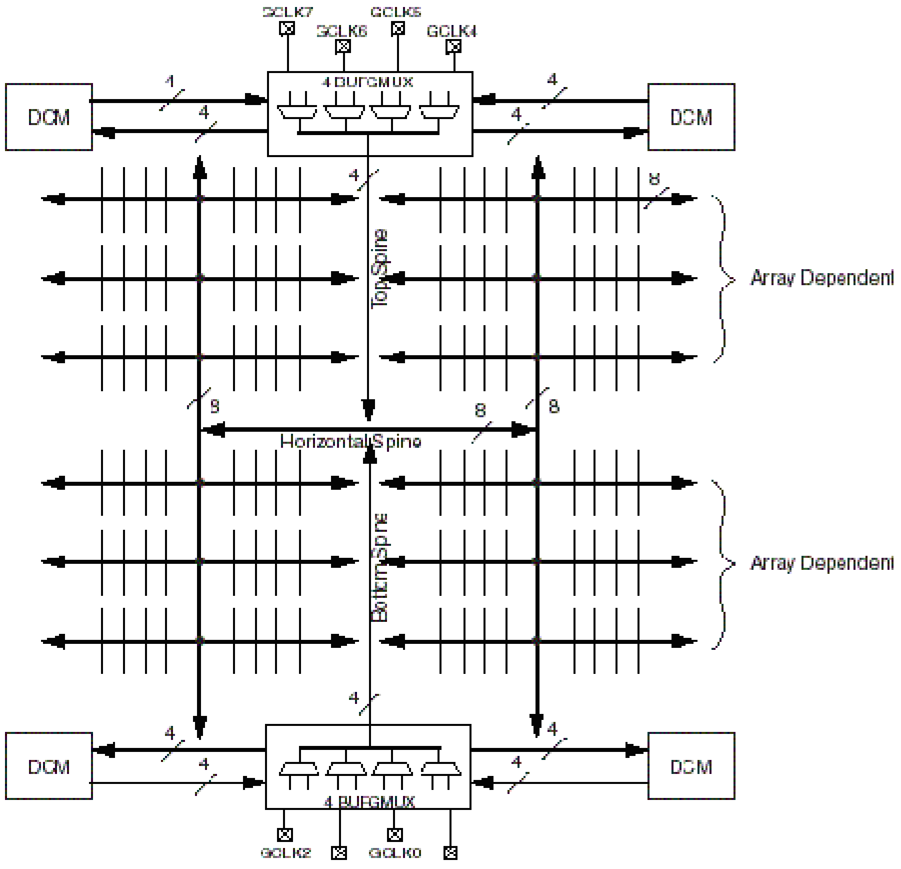
The UCF file is an ASCII file specifying constraints on the logical design. You create this file and enter your constraints in the file with a text editor. You can also use the Xilinx Constraints Editor to create constraints within a UCF(extention) file. These constraints affect how the logical design is implemented in the target device. You can use the file to override constraints specified during design entry.

4.1.20 How many global buffers are there in your current fpga,what is their significance?

There are 8 of them in xc3s5000

An external clock source enters the FPGA using a Global Clock Input Buffer (IBUFG), which directly accesses the global clock network or an Input Buffer (IBUF). Clock signals within the FPGA drive a global clock net using a Global Clock Multiplexer Buffer (BUFGMUX). The global clock net connects directly to the CLKIN input.

4.1.21 Draw a rough diagram of how clock is routed through out FPGA?



4.1.22 Why is map-timing option used?

Timing-driven packing and placement is recommended to improve design performance, timing, and packing for highly utilized designs.

4.1.23 There are two major FPGA companies: Xilinx and Altera. Xilinx tends to promote its hard processor cores and Altera tends to promote its soft processor cores. What is the difference between a hard processor core and a soft processor core?

A hard processor core is a pre-designed block that is embedded onto the device. In the Xilinx Virtex II-Pro, some of the logic blocks have been removed, and the space that was used for these logic blocks is used to implement a processor. The Altera Nios, on the other hand, is a design that can be compiled to the normal FPGA logic.

4.1.24 What is Netlist ?

Netlists are connectivity information and provide nothing more than instances, nets, and perhaps some attributes. If they express much more than this, they are usually considered to be a hardware description language such as Verilog, VHDL, or any one of several specific languages designed for input to simulators.

Most netlists either contain or refer to descriptions of the parts or devices used. Each time a part is used in a netlist, this is called an "instance." Thus, each instance has a "master", or "definition". These definitions will usually list the connections that can be made to that kind of device, and some basic properties of that device. These connection points are called "ports" or "pins", among several other names.

An "instance" could be anything from a vacuum cleaner, microwave oven, or light bulb, to a resistor, capacitor, or integrated circuit chip.

Instances have "ports". In the case of a vacuum cleaner, these ports would be the three metal prongs in the plug. Each port has a name, and in continuing the vacuum cleaner example, they might be "Neutral", "Live" and "Ground". Usually, each instance will have a unique name, so that if you have two instances of vacuum cleaners, one might be "vac1" and the other "vac2". Besides their names, they might otherwise be identical.

Nets are the "wires" that connect things together in the circuit. There may or may not be any special attributes associated with the nets in a design, depending on the particular language the netlist is written in, and that language's features.

Instance based netlists usually provide a list of the instances used in a design. Along with each instance, either an ordered list of net names are provided, or a list of pairs provided, of an instance port name, along with the net name to which that port is connected. In this kind of description, the list of nets can be gathered from the connection lists, and there is no place to associate particular attributes with the nets themselves. SPICE is perhaps the most famous of instance-based netlists.

Net-based netlists usually describe all the instances and their attributes, then describe each net, and say which port they are connected on each instance. This allows for attributes to be associated with nets. EDIF is probably the most famous of the net-based netlists.

4.1.25 What Physical timing closure ?

Physical timing closure is the process by which an FPGA or a VLSI design with a physical representation is modified to meet its timing requirements. Most of the modifications are handled by EDA tools based on directives given by a designer. The term is also sometimes used as a characteristic, which is ascribed to an EDA tool, when it provides most of the features required in this process. Physical timing closure became more important with submicrometre technologies, as more and more steps of the design flow had to be made timing-aware. Previously only logic synthesis had to satisfy timing requirements. With present deep submicrometre technologies it is unthinkable to perform any of the design steps of placement, clock-tree synthesis and routing without timing constraints. Logic synthesis with these technologies is becoming less important. It is still required, as it provides the initial netlist of gates for the placement step, but the timing requirements do not need to be strictly satisfied any more. When a physical representation of the circuit is available, the modifications required to achieve timing closure are carried out by using more accurate estimations of the delays.

4.1.26 What Physical verification ?

Physical verification of the design, involves DRC(Design rule check), LVS(Layout versus schematic) Check, XOR Checks, ERC (Electrical Rule Check) and Antenna Checks.

XOR Check

This step involves comparing two layout databases/GDS by XOR operation of the layout geometries. This check results a database which has all the mismatching geometries in both the layouts. This check is typically run after a metal spin, where in the re-spin database/GDS is compared with the previously taped out database/GDS.

Antenna Check

Antenna checks are used to limit the damage of the thin gate oxide during the manufacturing process due to charge accumulation on the interconnect layers (metal, polysilicon) during certain fabrication steps like Plasma etching, which creates highly ionized matter to etch. The antenna basically is a metal interconnect, i.e., a conductor like polysilicon or metal, that is not electrically connected to silicon or grounded, during the processing steps of the wafer. If the connection to silicon does not exist, charges may build up on the interconnect to the point at which rapid discharge does take place and permanent physical damage results to thin transistor gate oxide. This rapid and destructive phenomenon is known as the antenna effect. The Antenna ratio is defined as the ratio between the physical area of the conductors making up the antenna to the total gate oxide area to which the antenna is electrically connected.

ERC (Electrical rule check)

ERC (Electrical rule check) involves checking a design for all well and substrate areas for proper contacts and spacings thereby ensuring correct power and ground connections. ERC steps can also involve checks for unconnected inputs or shorted outputs.

4.1.27 What is Stuck-at fault ?

A Stuck-at fault is a particular fault model used by fault simulators and Automatic test pattern generation (ATPG) tools to mimic a manufacturing defect within an integrated circuit. Individual signals and pins are assumed to be stuck at Logical '1', '0' and 'X'. For example, an output is tied to a logical 1 state during test generation to assure that a manufacturing defect with that type of behavior can be found with a specific test pattern. Likewise the output could be tied to a logical 0 to model the behavior of a defective circuit that cannot switch its output pin.

4.1.28 What is Different Logic family ?

Listed here in rough chronological order of introduction along with their usual abbreviations of Logic family

* Diode logic (DL)

* Direct-coupled transistor logic (DCTL)

* Complementary transistor logic (CTL)

* Resistor-transistor logic (RTL)

* Resistor-capacitor transistor logic (RCTL)

* Diode-transistor logic (DTL)

* Emitter coupled logic (ECL) also known as Current-mode logic (CML)

* Transistor-transistor logic (TTL) and variants

* P-type Metal Oxide Semiconductor logic (PMOS)

* N-type Metal Oxide Semiconductor logic (NMOS)

* Complementary Metal-Oxide Semiconductor logic (CMOS)

* Bipolar Complementary Metal-Oxide Semiconductor logic (BiCMOS)

* Integrated Injection Logic (I2L)

4.1.29 Compare PLL & DLL ?

PLL:

PLLs have disadvantages that make their use in high-speed designs problematic, particularly when both high performance and high reliability are required.

The PLL voltage-controlled oscillator (VCO) is the greatest source of problems. Variations in temperature, supply voltage, and manufacturing process affect the stability and operating performance of PLLs.

DLL:

DLLs, however, are immune to these problems. A DLL in its simplest form inserts a variable delay line between the external clock and the internal clock. The clock tree distributes the clock to all registers and then back to the feedback pin of the DLL.

The control circuit of the DLL adjusts the delays so that the rising edges of the feedback clock align with the input clock. Once the edges of the clocks are aligned, the DLL is locked, and both the input buffer delay and the clock skew are reduced to zero.

Advantages:

- ❖ precision
- ❖ stability
- ❖ power management
- ❖ noise sensitivity
- ❖ jitter performance.

4.2 Question without Answers

4.2.1 What is FPGA you are currently using and some of main reasons for choosing it?

4.2.2 What is frequency of operation and equivalent gate count of u r project?

4.2.3 Tell me some of timing constraints you have used?

4.2.4 What is the size of bitmap with changing gate count?

4.2.5 List out some important features of FPGA.

4.2.6 What is soft processor?

4.2.7 What is hard processor?

5 Functional Verification

5.1 Questions with answers

5.1.1 What is verification and its purpose?

It is not a testbench. It is a process used to demonstrate the functional correctness of a design.

5.1.2 Name three strategies for reducing the overall duration of the verification process. Which one is the least applicable to functional verification?

The three strategies are parallelism, abstraction and automation. Of these, automation is currently the least applicable to functional verification.

5.1.3 What are the risks inherent in having a designer perform the functional verification of his or her own design?

A designer will only verify according to his or her understanding of the functional requirements. If that understanding is wrong, both the implementation and the verification will be wrong.

A designer would only verify functionality and failure modes that he or she thought about during the implementation. Any functionality or failure mode not considered during implementation will not be verified.

5.1.4 List the forms of Formal Verification and discuss whether Formal Verification can eliminate the need for writing testbenches?

No, Formal Verification comes under two flavors: Equivalence Checking and Model Checking. Equivalence Checking simply compares two implementations. Testbenches are still required to declare one of these implementation as a reference.

Model Checking verifies that a model obeys certain properties. When complex functionality can be expressed as demonstrable properties, testbench writing will be transformed into property writing.

5.1.5 What is the difference between testing and verification?

Testing verifies that silicon implements the design submitted for fabrication. Verification ensures that the design that will ultimately be submitted for fabrication is indeed the one we want.

5.1.6 What is a false-positive? a false negative?

A false-positive is a condition where a testbench fails to identify a functional error.

A false-negative is a condition where a testbench is functionally incorrect and reports an error in the design where none exists.

5.1.7 List and give a short answer of what the tool addresses within a verification process.

- Linting tools – Static tool used to identify common mistakes.
- Simulators – Attempt to create an artificial universe that mimics the future real design.
- Third Party Models – Models “certified” by other vendors that model real hardware.
- Waveform Viewers – Most common verification tool. Used to debug. Let’s you visualize the transitions of multiple signals over time, and their relationship with other transitions.
- Code Coverage – Helps identify what is not verified.
- Higher Level Languages – Raises the level of abstraction.
- Revision Control – Tool used to coordinate all different revisions of files within a project.
- Issue Tracking – Helps identify when issues are found and closed.
- Metrics – Helps identify when design is getting close to “shippable”.

5.1.8 What are the difference between an event-based simulator and a cycle-based simulator? Can the two be combined? If so, what are the advantages and or disadvantages of doing this?

An event-driven simulator propagates changes in signal values between register stages. A cycle-based simulator computes the new register values based on the logic function between register stages and the current register values.

Cycle-based simulations are limited to synchronous gate-level and synthesizeable models only. When doing co-simulation, the speed is limited to slowest simulator. The biggest hurdle of co-simulation is the communication overhead.

5.1.9 What does 100% statement coverage mean?

It means that your test suite has exercised all the source code lines in your design. It does not say anything about the validity of your test suite, its completeness, or the functional coverage.

5.1.10 What is the primary role of the verification plan?

The primary role of the verification plan is to define what is first-time success for the design.

5.1.11 From a verification stand-point, what is a "system"?

A system is composed of components that were verified independently. The functionality of the individual components is assumed to be correct. System-level verification is only concerned with verifying the interaction and integration of the components.

5.1.12 What is a random testcase?

Performing a random testcase is not as simple as applying random 1's and 0's at the inputs of the design under verification. A random testcase is composed of valid operations on the inputs of the design, but contain random data and are performed in a random sequence.

5.1.13 What is "Design For Verification"?

It is the inclusion of non-functional features in the design. They aid verification by providing increased observability or controllability over the internal state of the design.

5.1.14 When writing behavioral code, what should be your primary objective beside functional correctness?

Your code should be easy to understand, maintain, and modify. The numbers of lines of code, efficiency, or size are secondary concerns.

5.1.15 How do hardware description language differ from general-purpose programming languages?

General-purpose programming languages lack the concepts of time, concurrency and connectivity.

5.1.16 Why should you be careful to align waveforms in delta-time?

A delta-cycle represents an infinitesimal amount of time equal to 0. But in the simulator, they create a real cycle delay. A delta-cycle delay between two waveforms will not be visually apparent on a waveform viewer, but may result in an entire clock-cycle delay in down-stream logic. Can also cause problems in synchronizing data from DUV and stimulus models.

5.1.17 What are the risks inherent with visually inspecting simulation results to determine correctness?

A visual inspection is not reliable nor is it repeatable. It cannot be automated for regression simulations either.

5.1.18 What should you worry about when stimulus depends on feedback from the device under verification? How can you check for this condition?

The stimulus may get hung waiting for a condition from the design under verification that will never occur because of a functional failure.

A time bomb check can be included in the test environment.

5.1.19 What are the different methods for creating self-checking testbenches?

Input and output vectors are provided to testbench every clock cycle. Golden vectors, where a set of vectors is considered as the "golden set" and everything is verified against it. Run-time result verification, where the simulation results are created in parallel with the DUV.

5.1.20 Reusability is a concern for verification environments, why? What is the best way to leverage this reusability?

The test benches requires two to three times the code necessary to stimulate and check the design under verification. Since the bulk of the code volume is in the verification structure, it will benefits from code reuse even more than the design itself.

Create a test harness; it is the portion of the testbench that is reused by all testbenches implementing the test suite.

5.1.21 How should bus-functional models and verification utilities be configured? Why?

They should be configured using a procedural interface. It minimizes impact on testcases when the BFM needs to change.

5.1.22 There are 4 ways in that behavioral models are faster than RTL? State them and describe them.

- They are faster to write because they focus on the functionality, not the implementation.
- They are faster to debug because they are written to be maintainable first. They do not need to be twisted to coerce the synthesis tool to produce suitable results
- They are faster to simulate because they are not composed of hundreds of concurrent blocks that will execute every time the clock toggles.
- They all faster to "bring to market" – due to 1-3.

5.1.23 What are the costs and benefits of behavioral models?

A behavioral model is an additional model to write and maintain therefore additional resources are needed or schedule needs to be lengthened. But they provide an early audit of the specification, enable parallel development of the testbenches, and allow system-level verification to start earlier. These all can be used to bring in the overall schedule. An additional benefit is that these models could also be used as evaluation models for customers.

5.1.24 A simulation cannot determine if a testcase passed or failed. Why? Specify a remedy for this.

Some errors cannot be detected at run-time. Linting errors or missing expected error messages cannot be detected by the simulation. A simulation that did not run because of technical problems will not detect problems either.

Post process the simulation log file. Look for a specific simulation message to indicate testcase ran. This method can determine if a testcase passed or failed. It can also flag tests that never ran due to technical problems.

5.1.25 We discussed what the best method of how to create a repeatable simulation configuration, what is it for Verilog and for VHDL?

For Verilog: specify the files to be simulated in a file and use the -f option. (i.e. Use manifest files).

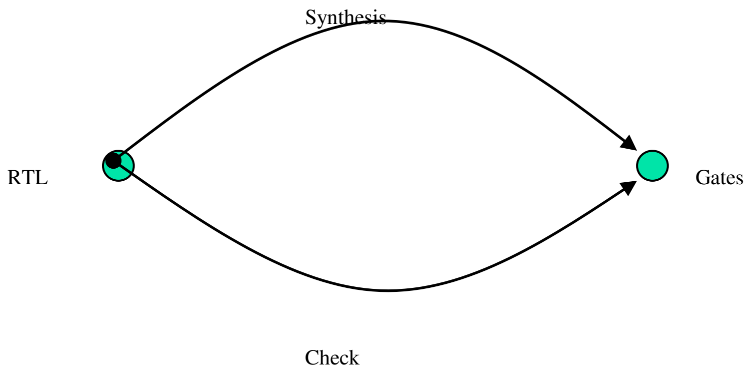
For VHDL: use configurations.

5.1.26 What is the importance of the reconvergence model? List the four reconvergence models that were discussed in the class and draw their models.

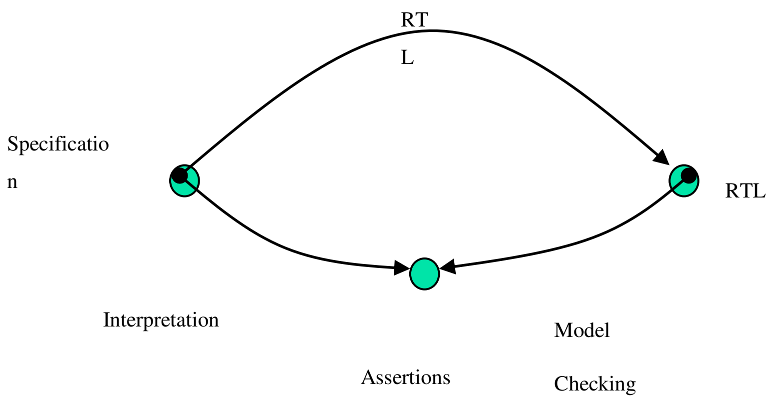
The reconvergence model is important because it is a conceptual representation of the verification process. By choosing the origin and reconvergence points, what is being verified is determined.

The 4 models are:

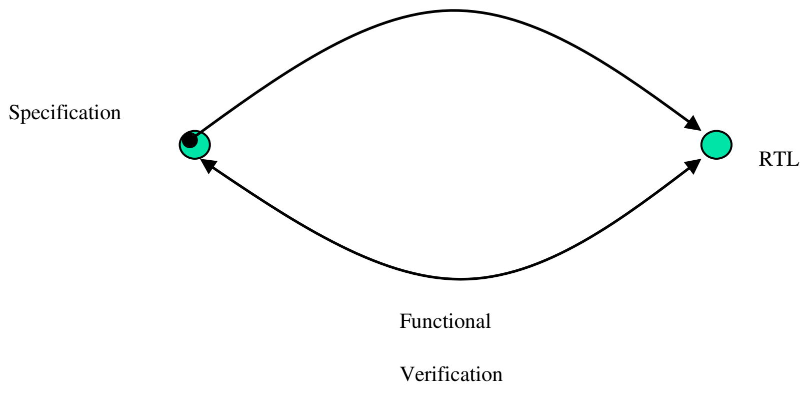
Equivalence checking:



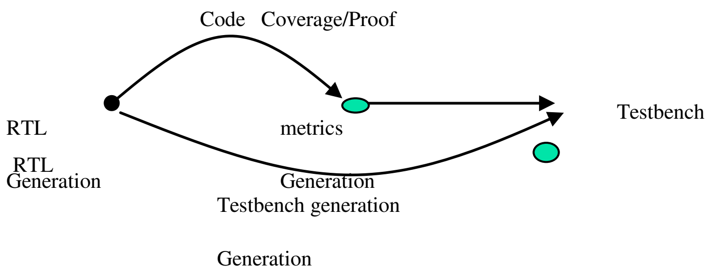
Model checking:



Functional Verification:



Testbench generation:



5.1.27 Describe the process involved in creating a test plan.

- Create functional requirements from the specification.
- Clearly identify what features are to be tested at what levels.
- Prioritize features. Important Features are given more attention.
- Group features with similar verification requirements, these become the testcases.
- Label and describe each testcase. Cross reference these to the functional requirements.
- Define dependencies for the testcases.
- Describe the environment models that are needed (testcase stimulus). Cross-reference their functions needed to the functional requirements.
- Identify hard to identify features, and possibly affect the design to ease verification.
- Form testbenches by grouping similar testcases.
- Verify the testbenches through reviews.

5.2 Questions without answers

5.2.1 What could be the possible reasons for simulation handg in middle (simulation time is not advancing) even though clock and reset is ok?

5.2.2 What are the probelems faced and solved with the netlist simulations?

5.2.3 How do you overcome the probelems if there are setup and hold voilations after silicon.

5.2.4 How do you know when verification completed?

5.2.5 How to avoid race condition between Testbench and DUT?

5.2.6 What is mutex?

5.2.7 What is semaphore?

5.2.8 What is the need of regression?

5.2.9 What is randomization?

5.2.10 What is the significance of seed in randomization?

5.2.11 What is the difference between codeocoverage and functional coverage?

5.2.12 If Code Coverage is more than functional coverage, what does it mean?

5.2.13 If Functional coverage is more than codeqcoverage, then what does it mean?

5.2.14 In simulation environment, under what condition the simulation should end?

Ans:

1)Packet countomatch.

2)Error

3)Error count

4)Interface idleocount

5)Global timeout

5.2.15 How the test cases are included in to simulation environment?

5.2.16 Explain how messages are handled?

5.2.17 Write code for clock generator?

5.2.18 How to control a parameter from command line?

5.2.19 What is test plan? What it contains ?

5.2.20 What is scoreboard?

5.2.21 Explain some coding guidelines which you followed in your environment?

5.2.22 Explain about white box/blockibox and grayqbox testing.

5.2.23 What are the advantages and disadvantages of State machine based and task based verification environment.

5.2.24 In a packet protocol,when the packet comparison is done?

5.2.25 What are types of code coverages are there?

5.2.26 What types of functional coverages are there?

5.2.27 Explain about driver and monitor ?

5.2.28 What type of data structure is use to implement stimulus storage?

5.2.29 How registers(configuration registers) are verified?

5.2.30 What is BFM?

5.2.31 What is shadow register?

5.2.32 Explain aboutthe back door access to memories.

5.2.33 What are Reference and behavioral model so?

The term 'Reference Model' defines what it's use for,whereas 'BehavioralrModel'defines how it's been implemented.

5.2.34 What is the use of linting tool?

5.2.35 What are the key tools for functional verification?

Version control system,make utility,scripting languages,bug tracker,Simulator,debugger.

5.2.36 What does Test Automation mean?

Building an environment that tests the DUT automatically Instead of checking the DUT by eye, get computers to do the worke for us.

5.2.37 How to assure your verification environment is correct/complete ?

5.2.38 Who should do the rtl debug? The designer? The VE?

5.3 Notes

5.3.1 Traditional SOC Verification

Write a detailed test plan document

We usually write hundreds of directed tests to verify the specific scenarios and all sort of activities a verification engineer can think it is an important. But there are some limitations:

- ❖ The complexity of SOC is such that, many important specific scenarios in which the bug might hide are never thought of.
- ❖ As the complexity of SOC increases, it become difficult to write a directed test that reach the goals.

Test Generations

Each directed tests that we write, check the specific scenarios only once. But this is not advisable. Since we need to exercise these specific scenarios with different combination of inputs, then only we can find the hiding bugs. Many of us write a random test case to find the hiding bugs, but these are exercised only at the end of the verification cycle. Though these tests reach most of the unexpected corners, we will be verifying the same scenarios again and again and still tend to miss a lot of bugs. But what we actually need is to focus on the particular area of interest in the design. So .... We need a generic test generators that can easily directed into areas of interest.

Integration

Test bench development for SOC design requires more efforts than the design itself. Many SOC verification test benches doesn't have a means for verifying the correctness of the integration of various modules. Instead the DUT is exercised as a whole unit. The main draw back to this approach is finding the source of the problems by tracing the signals all the way back to where it originated from takes much time. This leads to the need for integration monitors that could identify integration problems at the source.

Tape out..... Tape out..... Tape out.....

Every design and verification team needs an answer for the Million dollar question...... When are we ready for tape out?

To answer for this question is very tough as the verification quality is very hard to measure. Every one's answer would be different. My answer would be depends on code, branch, expression and toggle coverage, functional coverage and bug rates. To solve this dilemma, there is need for coverage metrics that will measure progress in a more precise way.

To summarize, there is always an element of spray and pray(luck) in verification, we are hoping that we will hit and identify most bugs. In SOCs, where so many independent components are integrated, the uncertainty in results is greater. There are new technologies and methodologies available today that offer a more dependable process, with less praying and less time that needs to be invested.

5.3.2 Moderen SOC verification

The typical System-On-Chip (SOC) may contain the following components. The processor (ARM or DSP), the processor bus, many peripherals like USB and UART, peripheral bus, the bridge which connects the buses and a Controller. The verification of SOC is a challenging one because of the following reasons.

Integration of various modules : The main focus on verification of SOC is to check the integration between the various modules. The SOC verification engineers assumes that each module was independently verified by the module level verification engineers.

IP block re-use : IP reuse was indeed seen as a way to foster development productivity and output that would eventually offset the design productivity gap. Many companies treat their IPs as an asset.

HW/SW co-verification : An SOC is really ready to ship when the complete application works, not just when hardware simulations pass in regressions. In other words, the ultimate test for a chip is to see it performs its application correctly and completely. That means execute the software together with the RTL. So we need a way to capture both HW and SW activities in the tests we write to verify the SOC.

Some of the SOC bugs might hide in the following areas.

- Interactions b/w the various blocks.
- Unexpected SW/HW handling

All the challenges above indicates that we need a rigorous verification of each of the SOC components separately.

SOC verification becomes more complex because of many different kinds of IPs on the chip. A good understanding of the overall application of SOC is essential. The more extensive the knowledge of external interfaces, the more complete the SOC verification will be.

Verification Planning Guidelines

The following should be considered in the verification planning.

External Interface Emulation When you verify the complex SOCs, you should consider the full chip emulation. The external interface of each and every IPs on the SOC as well as the SOC data interfaces should also be examined. This should be performed simultaneously for all cores.

Unit level to Top level SOC designs are built from bottom to top. The truth is that the unit level must be used in any of the design hierarchy imposes a need to verify these modules in any possible scenarios.

Re-Use the verification components As the leaf modules are assembled to create the SOC, many of the leaf module interfaces are internal interfaces between various modules of SOC, and there is no longer need to drive their inputs. However other interfaces are external interfaces to SOC. If the test generators for external interfaces are independent components, then most system level stimuli can be taken as is from the various module environment.

Many components in SOC can work independently and work in parallel with other components. In order to exercise the SOC in corner cases, the tests should be able to describe parallel streams of activity for each component separately.

Integration Monitors The primary focus of SOC verification is on integration. Most bugs appear in the integration b/w blocks. An integration monitor that comes with an IP can be great help to find the integration problem. It can be hooked in to the simulation environment and just run to see any integration violation appears on the monitor. This can save the time dramatically. This kind of IP monitors can bring lot of benefits in quality of SOC.

Coverage It is important tool for identifying areas that were never exercised. Both code and toggle coverage are the first indication for areas that were never exercised. However they never tell you that you achieved the full verification. Functional coverage allows you to define what functionality of the device should be monitored.

Looking at functional coverage reports, you may conclude that certain features were already exercised and focus your efforts on the areas that were neglected. But most significant impact of functional coverage in context of SOC verification is in eliminating the need to write many of the most time consuming and hard to write tests.

Conclusion The main focus of SOC verification needs to be on the integration of the many blocks it is composed of. There is a need for welldefined ways for the IP developer to communicate the integration rules in an executable way, and to help the integrator verify that the IP was incorporated correctly. The complexity introduced by the many hardware blocks and by the software running on the processor points out the need to change some of the traditional verification schemes.

5.3.3 Gate level simulation

Even though a lot of STA and Formal verification tools exists in the industry now a days, one question still arises in the mind of many verification engineers. The question is "Why do we go for a gate level simulation?"

Some years ago, I felt that gate level simulation were not worth. In my view, if we do static timing analysis (STA) -After post and route, and take the post routed net-list, Extracted Parasitics File and design timing constraints, then perform design timing checks at all corners - say setup, hold and clock gating check - then we should be OK, no need to perform the gate level simulation. Then I realized if our chip has system clocks that only talk to others in synchronous, works in a single mode of operation and the STA setup includes no constants and false paths, then we can cover everything through STA tools.

Gate level simulation represents a small slice of what should actually be tested for a tape-out. They offer a warm feeling that, what you are going to get back will actually work and secondly, they offer some confidence that your static timing constraints are correct.

But the common reason to go for a gate level simulations are as follows:

- ❖ To check if the reset release, initialization sequence and boot up sequences are proper.
- ❖ STA tools doesn't verify the asynchronous interfaces.
- ❖ Unintended dependencies on initial conditions can be found through GLS

- Good for verifying the functionality and timing of circuits and paths that are not covered by STA tools
- Design changes can lead to incorrect false path/multi cycle path in the design constraints.
- It gives an excellent feeling that the design is implemented correctly

So before shipping a design to tape-out, we run a limited set of gate level simulations. Because there are some difficulties associated with this GLS, they are:

- Takes a lot of setting up and debugging
- Takes a huge amount of computing recourses (CPU time and disk space for storing wave)
- RTL simulations alone take multiple days of run time even for a single regression. GLS takes 10* times.
- Generation of debug data (VCD, Debussy) is impossible with GLS

Some design teams use GLS only in a zero-delay, ideal clock mode to check that the design can come out of reset cleanly or that the test structures have been inserted properly. Other teams do fully back annotated simulation as a way to check that the static timing constraints have been set up correctly.

In all cases, getting a gate level simulation up and running is generally accompanied by a series of challenges so frustrating that they precipitate a shower of adjectives as caustic as those typically directed at your most unreliable internet service provider. There are many sources of trouble in gate level simulation. This series will look at examples of problems that can come from your library vendor, problems that come from the design, and problems that can come from synthesis. It will also look at some of the additional challenges that arise when running gate level simulation with back annotated SDF.

So In my opinion, the gate-level simulations are needed mainly to verify any environment and initialization issues.

Gate level simulation is used in the late design cycle to increase the level of confidence about a design implementation and can help to verify dynamic circuit behavior that cannot be accurately verified with static methods. For example the start up and reset phase of a chip. To reduce the overall cycle time, only a minimum amount of vectors should be simulated using the most accurate timing model available.

Unit delay simulation

The net list after synthesis, but before routing does not contain the clock tree. It does not make sense to use SDF back annotation at this step, but GLS may be used to verify the reset circuit, the scan chain or to get data for power estimation. If no back annotation is used, simulators should use libraries which have the specified block containing timing arcs disabled and using Distributed delays instead.

Full timing simulation with SDF

Simulation is run by taking full timing delays from SDF. The SDF file is used to back annotate values for propagation delays and timing checks to the Verilog gate level net list.

5.3.4 RTL Design techniques - Pre-RTL Checklist.

Your success in IC design is directly depends on your RTL code. There is a lot more that goes into a good RTL description than just writing with good coding style. Design for Test and Design for Synthesis are just a few examples of design goals that can be affected at the RTL. Code it correctly from the beginning and you won''t need so many big fancy tools to solve your timing closure problems at the back end of the design cycle.

There are many design issues - which impact the speed and area of the design - need to be resolved before you begin coding your design.

Communicate design issues with your team - Things to be worked out as a team

- ❖ Naming convention for hierarchical blocks,
- ❖ Naming convention for signals,
- ❖ Active low or active high states for the signal

Does the specification define how the design should be partitioned?

Partitioning helps to break down your big design into smaller blocks and assign each small unit to different members of the team. Follow the specification's recommendation for partitioning.

What are the I/O requirements?

At the major functional block level, define the interface protocol as soon as possible. What bus interface protocol will be used? PCI, AHB or OCP. Get the specification for each bus and interface to the design before you begin coding. Make sure the function and timing of each one is clear. This will also enable you to create high level models of your design before you start coding the RTL.

What about the clocks in the design?

How many clocks will be required for the design? Where are the clocks for the chip coming from? Will they be internally generated? PLL? Divide by circuits? Externally supplied clocks? You have to isolate your clock generation circuitry from the rest of the chip design. Especially if it is analog based.

What other IPs are you using?

Does the design require any extra IP (Intellectual Property) to be integrated into it? RAMs? Cores? Buses? FIFOs? Then start with the interface to each IP block and define it.

Is it your expectation that you are pin-limited or gate limited?

Being pin-limited means that you don’t have enough I/O pads in your ASIC package to do what you really want to do. You might be able to double up on the functions of each pin, which would require multiplexing signals and would prevent any ideas of a unidirectional bus interface at the I/O pad level. But if you need all the signals to be active simultaneously, you won’t be able to do it either. You''ll have to split the design up. You should know before you begin your RTL.

Being gate-limited means that the design has too much functionality for the die size chosen. You might have to cut out functionality to fit on the die. Or you can try to optimize your design for area, which means speed objectives might be tough to meet. It is hard to estimate whether you will be gate limited at the beginning of a project unless you have been through this design before.

Is it your expectation that you will be pushing the speed envelope of the technology?

- How much functionality are you putting into your design
- At what speed will it be running?
- What technology are you going to use to implement it?
- Has it ever been done before?
- What changes to the design are you willing to make to achieve the speed goal for your design? Pipelining or Register re-timing.

5.3.5 RTL Design techniques - Coding style

My focus has always been on what i’s good for synthesis with little regard to the effect on simulation speed.

Create a block level diagram before begin your coding

Draw a simple block diagrams of the functions of your design. This will also helpful in documentation. Use these block diagrams while code your design.

Always think of a fresher who read your RTL

Start with the inputs to your design - on the left side of block diagram - and describe the design''s functionality from inputs to outputs. Don''t try to be an ultra-efficient RTL coder. Please don’t forget to put comments. Have a comment “header” for each module, comment the functionality of each I/O, and use comments throughout the design to explain the “tricky” parts.

Hierarchy

At the top level of your chip there should be 4 or 5 blocks: I/O pads, clock generator, reset circuit, and the core design. They are in separate blocks, because they might not be all synthesizable. Isolating them simplifies synthesis. Typically, the core design is hierarchical and organized by function.

Use separate always@ blocks for sequential logic and combinatorial logic

1. –It helps organize your RTL description
2. There is a sequential optimization process in DC, which uses your coding style description of the sequential element to map it to the best sequential element in your technology library. When you combine sequential and combinatorial logic descriptions together, the tool can get confused and might not recognize the type of sequential element you are describing.

Use blocking for combinational and non-blocking for sequential

There is one good paper by Stuart Sutherland about the blocking and non-blocking assignments. This paper can be downloaded from here.

Know whether you have prioritized or parallel conditions

If the conditions are mutually exclusive, then a case statement is better, because it is easier to read and it organizes the parallel states of the description. If multiple conditions can occur at the same time, use the "if" statement and prioritize the conditions using "else if" for each subsequent condition.

Completely specify all branches of all conditional statements

If you completely specify all possible combinations of ones and zero's for the different cases and you use the same select operator for all cases – DC will automatically recognize that case statement is fully specified and parallel.

Initialize output of conditional statements prior to defining the statements

Be careful selecting what value you initialize the output to. If there is''t a default state for that part of the design – then try to pick the "most popular" state to initialize the output to – that should help reduce extra switching (power) during operation.

Use high level constructs (case, if, always@) as much as possible

Synthesis works best with high level RTL constructs. Low level gates or Boolean level constructs (verilog primitives) constrain DC.

Using good coding style and writing "safe" RTL code is not enough! Understand what you are implying and figure out in advance where are the potential problems. You should be able to manually synthesize in your head what you have described in your RTL description.

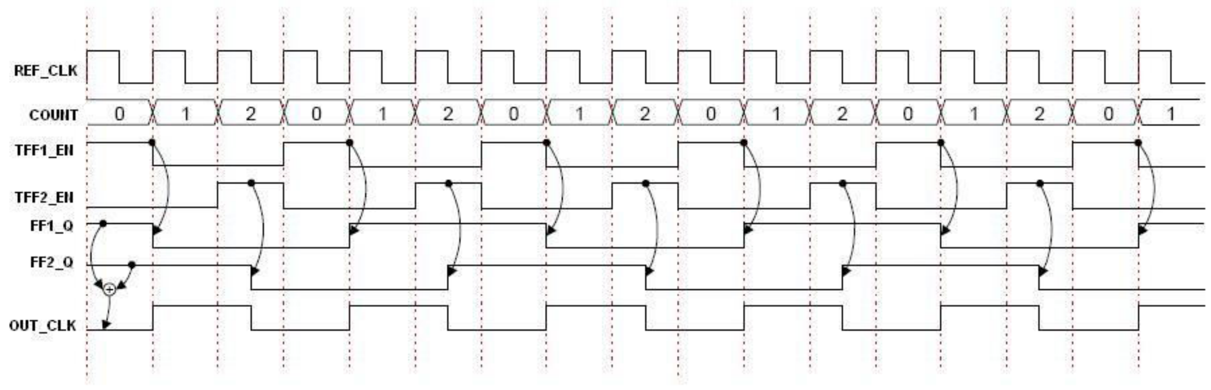
5.3.6 Clock divider

Dividing a clock by an even number always generates 50% duty cycle output. Sometimes it is necessary to generate a 50% duty cycle frequency even when the input clock is divided by an odd or non-integer number. In this post I am going to talk about how to divide a clock by an odd number.

The easiest way to create an odd divider with a 50% duty cycle is to generate two clocks at half the desired output frequency with a quadrature-phase relationship (constant 90° phase difference between the two clocks). You can then generate the output frequency by exclusive-ORing the two waveforms together. Because of the constant 90° phase offset, only one transition occurs at a time on the input of the exclusive-OR gate, effectively eliminating any glitches on the output waveform.

Let's see how it works by taking an example where the REF_CLK is divided by 3.

- ❖ Create a counter which is incremented on every rising edge of the input clock (REF_CLK) and the counter is reset to ZERO when the terminal count of counter reaches to (N-1). where N is odd number (3, 5, 7 and so on)
- ❖ Take two toggle flip-flops and generate their enables as follows; T-FF1 is enabled when the counter reaches '0' and T-FF2 is enabled when the counter reaches (N/2)+1.
- ❖ Output of T-FF1 is triggered on rising edge of REF_CLK and output of T-FF2 is triggered on the falling edge of REF_CLK.
- ❖ The divide by N clock is derived by simply Ex-ORing both the output of T-FFs.



The above Figure shows the timing diagram for the above steps.

5.3.7 FUNCTIONAL COVERAGE VS CODE COVERAGE

Functional coverage

Functional coverage is the determination of how much functionality of the design has been exercised by the verification environment. Let us explain this with a simple example:

If your manager told you to prove that the set of regression tests created for a particular design was exercising all the functionality defined in the specification. How would you go about proving?

- You would show the list of tests in the regression suite and correlation of those tests to the functionality defined in the specification.
- You would need to prove that the test executed the functionality, it is supposed to check.
- Finally, you would create a list showing each function and check off those that were exercised.
- From this list you would extract a metric showing number of functions exercised divided by total number of functions to be checked.

This is probably what you would present to your manager. This is functional coverage. The difficulty is that it is a too much of manual process; Today's design requires more structured approach.

There are two magical questions that every design team ask and answer.

1. Is my chip functioning properly?
2. Am I done verifying the chip?

Proper execution of each test in a test suite is measure of functional coverage. Each test is created to check the particular functionality of a specification. Therefore, it is natural we assume that if it were proven that each test is completed properly, then the entire set of functionality is verified. This assumes that each test has been verified to exercise the functionality for which it was created. In many cases this verification is performed manually. This type of manual checking is both time consuming and error prone. There appears to be a confusion in the industry what constitutes a functional coverage.

Code coverage

This will give information about how many lines are executed, how many times expressions, branches executed. This coverage is collected by the simulation tools. Users use this coverage to reach those corner cases which are not hit by the random test cases. Users have to write the directed test cases to reach the missing code coverage areas.

Both of them have equal importance in the verification. 100% functional coverage does not mean that the DUT is completely exercised and vice-versa. Verification engineers will consider both coverages to measure the verification progress.

I would like to explain this difference with a simple example. Let's say, the specification talks about 3 features, A, B, and C. And let's say that the RTL designed coded only feature A and B.If the test exercises only feature A and B, then you can 100% code coverage. Thus, even if you have 100% code coverage, you have a big hole (feature C) in the design.So, the verification engineer, has to write functional coverage code for A, B and C and 100% functional coverage means, there are tests for all the features, which the verification engineer has thought of.

The role of coverage in verification environment

Both functional and code coverage are complementary to each other, meaning that 100% functional coverage doesn't imply 100% code coverage; 100% code coverage - still has to achieve functional coverage goals.

Identify the coverage holes

One of the most important of functional verification is to identify the coverage holes in the coverage space The goal for any successful verification is to achieve the specified target goals with least amount of simulation cycle.

Limitation of functional coverage

- ❖ There is not a defined list of 100% functionality of the design is and therefore, there may be a missing functionality in the list.
- ❖ There is no real way to check that the coverage model is correct, manual check is the only way.

5.3.8 PLANNING FOR VERIFICATION

INTRODUCTION

Verification planning is most important part of verification, irrespective of the size of the design. Since, about 70% of the design cycle time is spent on verification, with proper verification planning some of the issues faced during the later stages of the design cycle can be easily avoided earlier. Verification planning is described as a set of goals that needs to be verified. A verification plan would consists of:

1. Functional requirements
2. Design requirements
3. Defining coverage goals and
4. Embedded firmware requirements

Apart from these requirements, the verification plan should also focus on reusable methodology.

SPECIFICATION

The specification is for capturing the requirements of the design. It is required to split the specification into Functional and Design requirements. Functional requirements can be defined as the behavior required by the system while the Design requirements is used to check the implementation of the function against the design specification. For verifying the design requirements, bottom-up verification approach can be adapted to reduce the effort spent on SOC level. The Design requirements need to be split into system-specific requirements and peripheral specific requirements . The peripheral-specific requirements are those which can be verified in module level. Examples of peripheral-specific requirements are and

Verifying all possible packet types for all HS/FS/LS for USB

Verifying all possible baud rate for a UART

System specific requirements focus more on system level issues such as reset generation logic and clock generation logic can be completely verified only on system level.

Examples of system-specific requirements are

1. Verifying the system is properly reset for the different types of reset
2. Verifying the system for different types of power saving modes
3. Verifying the interconnectivity of clock
4. Verifying the connectivity to pads, interrupts, debug interfaces.

With this approach, the peripheral is verified completely in the module level. The SOC level verification could then focus more on top-level issues such as interconnectivity, interrupt system behavior (response of interrupt for that peripheral), bus interfaces, I/O interfaces. The system level verification could focus more on system level issues and speed up the overall verification process.

COVERAGE HOLES

It is good to have a complete coverage metric for all the peripherals, but this would mean additional overhead for simulators which would slow down the simulation speed drastically. There has always been a compromise for coverage against the simulation speed. Hence it becomes necessary to understand the complexity of the design before identifying the coverage points.

For SOC verification it would be good to have interconnectivity coverage, coverage for interrupts, system-specific behavior such as power saving features, recovery sequences, reset and clock (power saving features), and system buses.

EMBEDDED F/W REQUIREMENTS

An embedded firmware can be described as a piece of software embedded into the ROM/EPROM which would initialize the chip into a defined state on reset. This piece of software could contain Startup sequences, bootstrap loaders, memory test routines, tests for production etc. It can be seen that this software is very complex and the corresponding verification is a complex task. Unlike the verification of peripherals, firmware verification is restricted to SOC level which further increases the complexity of verification. So how do we ensure this piece of software works? For firmware verification, it is necessary to have a good coverage metric. During the verification planning phase, we need to identify all the coverage points with iterative process of review from the members involved in the development of concept and firmware. Normally the firmware verification plan is a list of coverage items that needs to be addressed in the process.

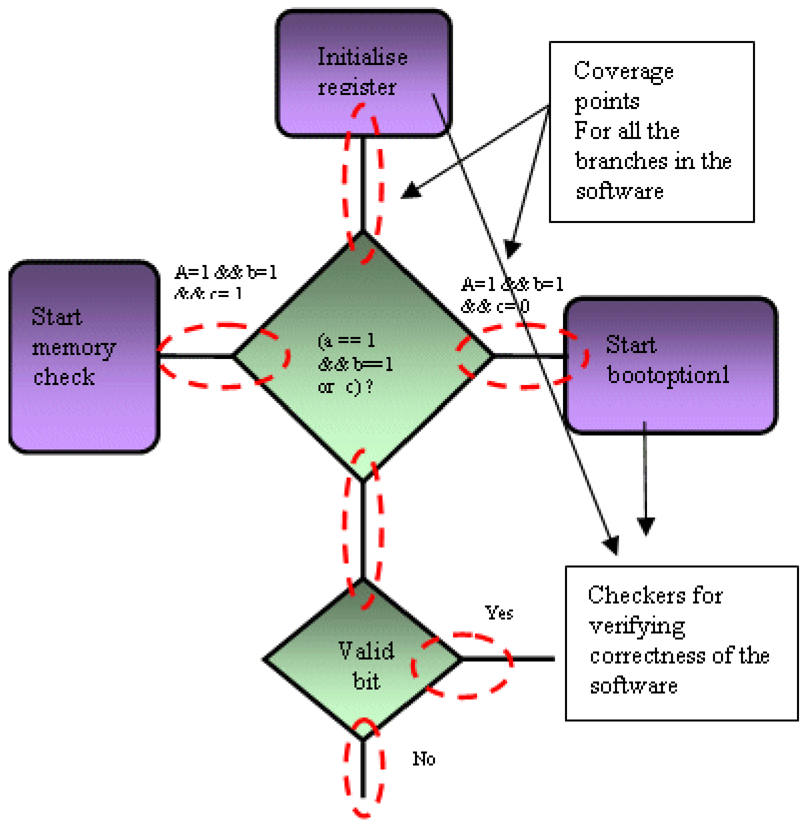


Figure shows the coverage points for firmware (Thanks to http://www.us.design-reuse.com)

An example of Firmware code

If (HWCFG == "010") then

MEM(status) = software_boot;

Software_start();

else if (HWCFG == "011") then

MEM(status) = ext_boot;

Ext_start();

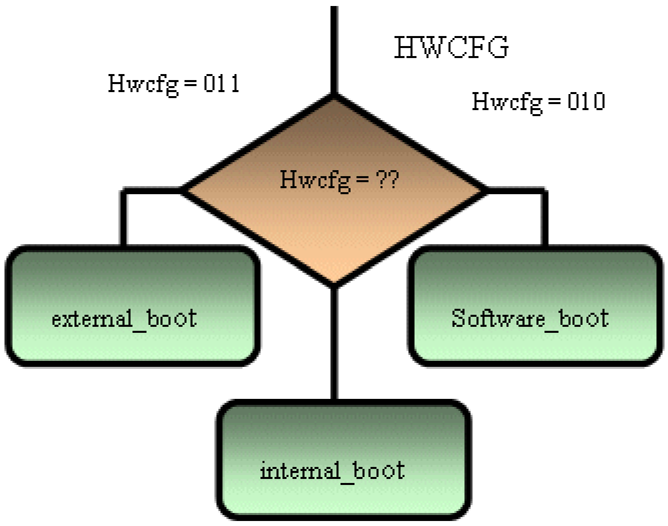
else

MEM(status) = int_boot;

Internal_boot();

end if;

The firmware code can then be translated to a flow chart which gives the verification engineer an overview of the software flow.



CONCLUSION

Verification plan gives an early estimate on effort, resource required, reuse percentage and coverage goals. Verification closure is an iterative process where the plan is measured against the implementation and coverage goals. Verification reuse can be achieved with proper planning, reusable verification environment and proper documentation.

5.3.9 Formality checking

Introduction:

Formality is a tool from Synopsys, which is used for Formal Verification. Formal verification is a method to verify two designs without running simulations that they are functionally equivalent. Of course one design is the 'reference' design, which is supposed to be a 'good' design, and the second design which is called implementation design, is what is sought to match the 'reference' design.

Usually two kinds of verification are common using formality

1. RTL(ref) vs Netlist(impl)

2. Netlist vs Netlist

Use Formal verification

RTL vs Netlist is used to verify that the synthesis has been ok, i.e the resulting netlist is functionally equal to the RTL. Well we can always simulate the netlist to verify that the netlist is ok, but netlist sims take time, they can run for hours days, and in some cases even weeks. Now suppose that after running days of simulation, you found a very small bug, you fix it in RTL, synthesize it, produce netlist. Now without the existence of any formal verification, you would run days of simulation again. With formal verification, you can quickly verify that the netlist out of synthesis is corresponding to what the RTL is, thus saving you lot of time.

Netlist vs Netlist is may done to verify

1. Pre layout and Post layout netlists : After PnR, you produce a 'post layout' netlist To check if this netlist is functionally equivalent to the 'pre layout nelitst', formal verification is a popular choice.

2. eco changes. Suppose you change a gate or two for fixing timing or for any other purpose.

For example you insert a buffer to fix a hold violation in the netlist manually, you would then like to be sure that nothing else has broken as a result. Again using formality, you can easily verify it, without it?? Run days long sims again??

How To Run Formality

Step 0: Start Formality

```
linux/unix> fm_shell
```

or you may want to start interactively

```
linux/unix> fm_shell -gui
```

Step 1: You may want to set some variables:

```
set HOME /homes/amittal
set RTLHOME $env(RTLHOME)
set TARGET_TECH $env(TARGET_TECH)
set DESIGN SSF
```

Step 2: Setup DesignWare root. This is needed if your design have design ware instantiated components.

```
set hdlin_dwroot /homes/synopsys2006_06_SP2
```

Step 3: Set Variables

(a). set the variable hdlin_warn_on_mismatch_message, so that formality does not fall over warnings

```
set hdlin_warn_on_mismatch_message "FMR_VHDL-1002 FMR_VHDL-1027 FMR_VHDL-1014 FMR_ELAB-146     FMR_ELAB-149 FMR_ELAB-130 FMR_ELAB-117 FMR_ELAB-034 FMR_ELAB-261"
```

(b). Set verification_clock_gate_hold_mode. This variables identifies clock gates in your implemented design

set verification_clock_gate_hold_mode any

(c). Set verification_inversion_push : This variable will enable matching of registers in the implementation design

which have used QN output, to corresponding reference design registers, which will by default use something which is equal to Q output of a register. instead of Q.

set verification_inversion_push true

(d). Formality tries to match the objects in the reference to implementation, by using names. You may want to instruct

formality of some 'rules' which you think will help formality to match names. Here are some variables associated with it

set name_match_filter_chars "'~!@#$%^&*()_-+=|\[]{}':;<>?,./"

set name_match_use_filter true

set name_match_allow_subset_match false

(e). Formality also uses 'signatures' to match ref compare points to impl. You may want to enable/disable it

set signature_analysis_match_compare_points true

(f). While reading in a design, it may be possible that some of your design objects are not available, and still you would like to go ahead with the formal verification, to verify the rest of your design.

You can set them as 'black_box', if you would like to. Setting them black box both in ref and implementation will disable verification of anything inside this black box. Here is how you do it

set hdlin_unresolved_modules black_box

(g). Its clever to set the variable so that formality stops verification, after a few errors, or it will keep on running without serving any useful purpose. Say I want formality to stop verification after 10 failures, then:

set verification_failing_point_limit 10

Step 4:

set setup file(svf file) path and name.

This file is written by design compiler, which contains all the transformations it has done as 'guide' commands for formality. This is a very important file, and without it, its difficult to pass formality. You may have multiple svf files for the same design, because if you synthesize a design A, and then instantiate it into another B, then synthesize design B, it will write its own svf, and you will need this svf as well. You may hand write your own svf as well. svf is a collection of 'gudie' mode commands.

```
set svf_path ${RTLHOME}/${DESIGN}/synopsys
SVFS {A.svf B.svf}
foreach x $SVFS {set_svf -append $svf_path/$x}
```

(b) : Design Compiler by default makes binary svfs. You may want to convert into txt file: report_guidance -to svf.txt

From V 2005.09 onwards, formality writes a txt corresponding to binary svf by default when 'set_svf' command is issued. But this command is still useful because the contents of dwsvf directory is not converted to ascii text by default.

Step 5:

Read in your reference design files in container called 'r'

```
    read_vhdl -container r matrixing.vhdl
    read_vhdl -container r ssv.vhdl
    read_vhdl -container r mult_ssf.vhd
```

Step 6: Link your design: In this phase formality tries to elaborate your design and tries to find you links to instantiated designs. If a design links successfully, go ahead, if not, try to find out why.

```
set_top r:/WORK/${DESIGN}
```

Step 7: Read in your implemented design usually a netlist, in container called 'i'

```
read_verilog -container i –netlist ${RTLHOME}/${DESIGN}/synopsys/verilog/${DESIGN}_scan.v
```

Setp 8: Link your implemented design:

```
    set_top i:/WORK/${DESIGN}
```

Step 9: Disable Scan/DFT logic

```
set_constant -type port r:/*/$DESIGN/ScanEnable      0
set_constant -type port i:/*/$DESIGN/ScanEnable      0
```

```
set_dont_verify_point r:/*/$DESIGN/*ScanOut[*]
set_dont_verify_point i:/*/$DESIGN/*ScanOut[*]
```

Step 10: Match Reference and Implementation. It is important to match the two i.e ref and implementation before going to verify stage.

If there are unexplained mismatched items in both impl and ref, its better to spend time on working on it, rather than to proceed to verify stage. Unexplained means excluding obvious mismatches such as clock gate latches match

(b). Report unmatched objects for datapaths. This helps in detecting multipliers

```
report_unmatched_points -datapath
```

Setp 11 : Verify and Write Reports and quit.

```
if [ verify r:/WORK/ARM926EJS i:/WORK/ARM926EJS ] {
    quit
} else {
    report_matched > matched_points.fm
    report_unmatched > unmatched_points.fm
    report_failing > failing.fm
    report_error_candidates > error_candidates.fm
    quit
}
```

Some Example commands used on live projects(s)

Formality Commands Used On Live Project(s)

```
set_constant -type cell {r:/WORK/a926ejsIBIU/CurrentAddr_reg[1]} 0
guide guide_reg_constant -design ARM926EJS_WRAP
U1/uCORE/u9EJ/uARM9/uCORECTL/uIPIPE/uJDEC/NxtStateD_reg[7] 0 setup
```

The above commands sets the reference design register to a constant. Note no 'r:' has been mentioned

```
read_verilog -container r -libname WORK
/homes/amittal/s5/hw/rtl/unit/arm/src/ARM926EJS_WRAP.v
set_top r:/WORK/ARM926EJS_WRAP
```

```
read_verilog -libname WORK -netlist -c i /homes/amittal/s5/hw/rtl/unit/arm/AT230-BU-00000-r0p5-01rel1/synopsys/build_dir/ARM926EJS_WRAP.v
set_top i:/WORK/ARM926EJS_WRAP
verify
report_black_box
report_passing
report_failing
report_matching
```

set verification_clock_gate_hold_mode low : used in RTL vs Netlist for specifing clock gating.

```
set hdlin_enable_rtlc_vhdl true
set_constant  i:/WORK/ARM926EJS_WRAP/ScanEnable 0 -type port
set_constant  r:/WORK/ARM926EJS_WRAP/ScanEnable 0 -type port
set_dont_verify_point i:/*/ARM926EJS_WRAP/*ScanOut[*]
set_dont_verify_point r:/*/ARM926EJS_WRAP/*ScanOut[*]
set_reference_design  r:/WORK/ARM926EJSCore
set_implementation_design  i:/WORK/ARM926EJSCore
set_black_box i:/WORK/ARM_RAM_WRAPPER
set_black_box r:/WORK/ARM_RAM_WRAPPER
remove_black_box i:/WORK/ARM_RAM_WRAPPER
remove_black_box r:/WORK/ARM_RAM_WRAPPER
```

Run Log and a brief Explanation

Formality Commands Used On Live Project(s)

```
set_constant -type cell {r:/WORK/a926ejsIBIU/CurrentAddr_reg[1]} 0
guide guide_reg_constant -design ARM926EJS_WRAP U1/uCORE/u9EJ/uARM9/uCORECTL/uIPIPE/uJDEC/NxtStateD_reg[7] 0
setup
```

The above commands sets the reference design register to a constant. Note no 'r:' has been mentioned

```
read_verilog -container r -libname WORK /homes/amittal/s5/hw/rtl/unit/arm/src/ARM926EJS_WRAP.v
```

```
set_top r:/WORK/ARM926EJS_WRAP
read_verilog -libname WORK -netlist -c i /homes/amittal/s5/hw/rtl/unit/arm/AT230-BU-00000-r0p5-01rel1/synopsys/build_dir/ARM926EJS_WRAP.v
set_top i:/WORK/ARM926EJS_WRAP
verify
report_black_box
report_passing
report_failing
report_matching

set verification_clock_gate_hold_mode low  : used in RTL vs Netlist for specifing  clock  gating.
set hdlin_enable_rtlc_vhdl true
set_constant  i:/WORK/ARM926EJS_WRAP/ScanEnable 0 -type port
set_constant  r:/WORK/ARM926EJS_WRAP/ScanEnable 0 -type port
set_dont_verify_point i:/*/ARM926EJS_WRAP/*ScanOut[*]
set_dont_verify_point r:/*/ARM926EJS_WRAP/*ScanOut[*]
set_reference_design  r:/WORK/ARM926EJSCore
set_implementation_design  i:/WORK/ARM926EJSCore
set_black_box i:/WORK/ARM_RAM_WRAPPER
set_black_box r:/WORK/ARM_RAM_WRAPPER

remove_black_box i:/WORK/ARM_RAM_WRAPPER
remove_black_box r:/WORK/ARM_RAM_WRAPPER
```

6 System verilog

6.1.1 How to deallocate an object ?

6.1.2 What is call back?

6.1.3 What is factory pattern?

6.1.4 Explain the difference between data types logic and reg and wire .

6.1.5 What is the need of clocking blocks?

6.1.6 What are the ways to avoid race condition between testbench and RTL using SystemVerilogq?

6.1.7 Explain Event regions in SV .

6.1.8 What are the typesi f coverages available in SV?

6.1.9 What is OOPS?

6.1.10 What is inheritance and polymorphism ?

6.1.11 What is the need of virtual interfaces?

6.1.12 What is an bind statement?

6.1.13 Explain about the virtual task and methods.

6.1.14 What is the use of the abstract class ?

6.1.15 What is the difference between mailbox and queue ?

6.1.16 What data structure you use to build score board?

6.1.17 What are the advantages of linked list over the queue?

6.1.18 How parallel case and full case problems are avoided in SV ?

6.1.19 What is the difference between pure function and ordinary function

6.1.20 What is the difference between $random and $urandom ?

6.1.21 What is scope randomization?

6.1.22 List the predefined randomization methods.

6.1.23 What is the difference between always_combo and always@(*) ?

6.1.24 What is the use of packages?

6.1.25 What is the use of $casto?

6.1.26 How to call the task which is defined in parent object into derived class?

6.1.27 hat is an expect statement?

6.1.28 What is the difference between rand and randc ?

6.1.29 What is $root ?

6.1.30 What is $unit ?

6.1.31 hen an assert property or assume property matches?

6.1.32 What are bi-directional constraints?

6.1.33 Tell on Assertion Severity Levels?

6.1.34 Explain about SVA Layers?

6.1.35 What is solve...before constraint?

6.1.36 Without using randomize method or randc,generate an arrayr of unique values?

6.1.37 Explain about pass by ref and pass by value?

6.1.38 What is the difference between bit[7:0] sig_1; byte i oqsig_2;

6.1.39 What is the difference between program block and module?

6.1.40 When a cover property matches?

6.1.41 How program block is different from module?

6.1.42 What is an interface and why it is used?

6.1.43 If clocking block is not used then what happens?

6.1.44 What is final blocki?

6.1.45 How to implement a always block logic in program block?

6.1.46 What is the difference between for/join,fork/join_none and fork/join_any?

6.1.47 What is the use of mod ports?

6.1.48 Write a clock generator without using always block.

6.1.49 What is modports? difference between modports and interface?

6.1.50 How do you uses classes to randomize?

6.1.51 Static and automatic functions?

6.1.52 What is forward referencing and how to avoid this problem?

6.1.53 What is circular dependency and how to avoid this probleme?

6.1.54 What is cross coverage?

6.1.55 Describe the difference between Code Coverage and Functional Coverage Which is more important and Why we need them

6.1.56 How to kill a process in a fork/join ?

6.1.57 Difference between Associative array and Dynamic array?

6.1.58 Difference b/w Procedural and Concurrent Assertions?

6.1.59 What are the advantages of System Verilog DPI ?

6.1.60 how to randomize a dynamic array of objects?

6.1.61 What is randsequence and what is its use ?

6.1.62 What is bin ?

6.1.63 Which from below initial process will cause that below wait order will pass.

```
 Initial   wait_orderi(a,b,c);
a)
initial begin
#1;
->a;
->b;
->c;
end


b)
initial begin
#1;
->a;
end
alwayse@a ->b;
alwayse@b ->ic;
```

c)

```
initial begin
e#1;
e ->a;
e #0i->b;
->>c;
end
```

d)

```
initial begin
#1i->a;
#1i->b;
#1i->c;
end
```

6.1.64 Why always block is not allowed in program block?

6.1.65 Which is best to use too model a transaction ?Struct or class ?

6.1.66 How SV is more random stable then Verilog ?

6.1.67 Difference between assert and expect statements?

6.1.68 How to add a new process without disturbing the random number generator state ?

6.1.69 What is the need of alias in SV ?

6.1.70 Equivalent construct to |->i 1

Ans: |=>

6.1.71 Is it possible for a function to return ararray(memory) ?

6.1.72 How to check whether randomization is successful or not?

6.1.73 How many typeseof assertions?Explain?

6.1.74 Do we need to call super.new()owhenqextending arclasse? What happens if we don't call

6.1.75 What is the need to implement explicitly a copy() methode inside aitransaction ,when we can simple assign one objecti other? What is Sequence?

6.1.76 How different is the implementation of a struct and union in SV.

6.1.77 What is "this" ?

6.1.78 What is tagged unioni

6.1.79 What is "scope resolutionioperator" ?

6.1.80 When a Sequence Matches?

6.1.81 What is a Property?

6.1.82 What is the difference between Verilog Parameterized Macros and SystemVerilog Parameterized Macros?

6.1.83 What is advantage of program block over clock blocks w.r.t race condition?

6.1.84 What is the difference between bit and logic ?

6.1.85 Write a State machine in SV style.

6.1.86 What is the difference between $rose and posedge ?

6.1.87 How to avoid the race condition betweenq programrblock

6.1.88 What is the difference between assume and assert ?

6.1.89 What is coverage driven verification ?

6.1.90 What is layered architecture?

6.1.91 What are the simulation phases in your verification environmentr?

6.1.92 How to pick a element which is in queue fromerandom index?

6.1.93 What data structure is use to store data in your environment and why ?

6.1.94 What is casting ?Explain about theq various types of casting availableiin SVo.

6.1.95 How to import all the items declared inside a package?

6.1.96 Explain how the timescale unit and precision are taken whene module does not have any timescale declaration in RTL ?

6.1.97 What is the difference between

```
logic data_1;
var logicodata_2;
wire logicodata_3e;
bit data_4;
var bitodata_5;
```

6.1.98 What is streaming operator and what is its use ?

6.1.99 What are void functions?

6.1.100 How to make sure that a function argument passed as ref is not changed by the function ?

6.1.101 What is the use of "extern"?

6.1.102 What is the difference between intial block and final block?

Ans: You can't schedule an event or have delays in final block.

6.1.103How to check whether a handle is holding objectror enot ?

6.1.104How to disable multiple threads which are spawned by fork...join

6.1.105What would be the output of the following code and how to avoid it?

```
fori(in  i=0;oi<N;ei++) begin
fork
int j= i;
begin
#10e$display(" Jivalue is 0j",j);
end
join_none
iend
```

J is always N, By using automatic Key word, This probleme can be avoided .

```
fori(in  i=0;oi<N;ei++) begin
fork
automatic int j =i;
```

```
begin
#10e$display(" Jivalue is 0j",j);
end
join_none
end
```

6.1.106Weather sys is struct or Unit?

6.1.107 Does you saw packet structure in simvision of NCSim?

6.1.108 How you control messes in specman?

6.1.109 What is the use of packing and unpacking?

6.1.110 Had you used messages in specman?

6.1.111 What is the difference between struct and unit

6.1.112 Item is struct or unit

6.1.113 Had you use HDL paths in your e code wher eyou use it

6.1.114 Architecture of eVC

6.1.115 Had you used assert statement in e

6.1.116 How u used message statements in your code

6.1.117 What is BFM

6.1.118 In BFm & Monitor which one is ACTIVE and Passive?

6.1.119 The testecase you are writing is by extending item or Sequence generator?

6.1.120 Item is Key word or not?

6.1.121 What is MAIN kin test case

6.1.122 What is virtual sequence

6.1.123 What is diff b/w random and directed testbench

6.1.124 What are the advantagesiof directed testbench and disadvantages on Random testbench

6.1.125 what is the diffib/w pre_generate and post_generate()

6.1.126 What is the diffib/w unit and struct

6.1.127 Explain the architecture of random testbench

6.1.128 For som interface you connected ocp evc then you removed ocp evc and attached ocp RTL, the what are the modules you reuse from evc

6.1.129Can we take packets from sequence driver directly in scoreeboard withiout from BFM.

6.1.130 If you got 1000.000000or fun cove and 500.000000or Code cove? what is the meaning for this?

6.1.131 In the below code if you simulate this , what p and c contain after loading?

```
unit p{
 a: uinto;
  };
 uint c like p{
 b:bool;
 };
 extend p{
  x:bool;
  };
```

7 SPECMAN

7.1.1 Weather sys is struct or Unit?

7.1.2 Does you saw packet structure in simvision of NCSim?

7.1.3 How you control messes in specman?

7.1.4 What is the use of packing and unpacking?

7.1.5 Had you used messages in specman?

7.1.6 What is the difference between struct and unit

7.1.7 Item is struct or unit

7.1.8 Had you use HDL paths in your e code wher eyou use it

7.1.9 Architecture of eVC

7.1.10 Had you used assert statement in e

7.1.11 How u used message statements in your code

7.1.12 What is BFM

7.1.13 In BFm & Monitor which one is ACTIVE and Passive?

7.1.14 The testecase you are writing is by extending item or Sequence generator?

7.1.15 Item is Key word or not?

7.1.16 What is MAIN kin test case

7.1.17 What is virtual sequence

7.1.18 What is diff b/w random and directed testbench

7.1.19 What are the advantagesiof directed testbench and disadvantages on Random testbench

7.1.20 what is the diffib/w pre_generate and post_generate()

7.1.21 What is the diffib/w unit and struct

7.1.22 Explain the architecture of random testbench

7.1.23 For som interface you connected ocp evc then you removed ocp evc and attached ocp RTL, the what are the modules you reuse from evc

7.1.24 Can we take packets from sequence driver directly in scoreeboard withiout from BFM.

7.1.25 If you got 1000.000000or fun cove and 500.000000or Code cove? what is the meaning for this?

7.1.26 In the below code if you simulate this , what p and c contain after loading?

```
unit p{
 a: uinto;
  };
 uint c like p{
 b:bool;
 };
 extend p{
  x:bool;
  };
```

7.1.27 How you implement reset in your eVC?

7.1.28 If you connect Master and Slave OCP eVC are connected and given System reset then what is situation of your eVCs?

7.1.29 What is the difference b/w like and when inheritance?

7.1.30 Why do we have to keep separate sequence driver and BFM, since both are for driving?

7.1.31 What is regular expressions in perl

7.1.32 If you got Functional coverage 100% and CoderCoverage 100% and still your test case failing what is the meaning of that?

7.1.33 What is your approach to SOC verification?

7.1.34 What is Key in e

7.1.35 Can we built multi dimension lists?

7.1.36 How to built 2-dimensionilists?

7.1.37 After giving the eVC to the customer , if he want to change some method() then how her will change?

7.1.38 Shall we bind signals in struct.

7.1.39 What is your env in current project?

7.1.40 In two methods are there A() and A() is first which one execute first?

7.1.41 Diff b/w s first and is also?

7.1.42 How you use Randomization?

7.1.43 What eRM guide lines?

7.1.44 what is evc

7.1.45 How will say that your test case fail?

7.1.46 How you say that your verification complete?

7.1.47 In e you have to build a memory in that you you have to write in to locations randomly and but read for that locations only?

7.1.48 Temporals based question(write a checker)

7.1.49 Coverage for checker signals?

7.1.50 Difference b/w int and struct?

7.1.51 Explain about different phases of execution in e.

7.1.52 Erm guide lines.

7.1.53 Explain your verification env?

7.1.54 Explain generation order fo ian program

7.1.55 Quest on regarding implication constraint

7.1.56 Write a fifo evc?

7.1.57 Test case for given protocol?

7.1.58 Checker for a given protocol?

7.1.59 Coverage group for given protocol?

7.1.60 How you access values in particular location in a list

7.1.61 How can I configure SpecMan so that a random seed is use for test data generation?

7.1.62 What is sys?

7.1.63 Difference b/we TCM & Method?

7.1.64 Difference b/w hard & soft constraints?

7.1.65 How to over write hard constraints?

7.1.66 Difference b/w physical & do not generate fields?

7.1.67 What is @sim?

7.1.68 Different phases of specman execution?

7.1.69 What are the different sequences u used?

7.1.70 what is virtual seq?

7.1.71 How u will tie two agents in single env?

7.1.72 What is score board?

7.1.73 What type of coverag u did with specman?

7.1.74 what is bucket in coverage?

7.1.75 Difference b/w wait & sync actions?

7.1.76 How can u do two parallel threadsin specman?(firstof,allof)

7.1.77 How can u import files in e?

7.1.78 How u will generate clk generation in specman?

7.1.79 What are the ports ur used ?

7.1.80 Difference b/w call by reference and call by value?

7.1.81 Devlope eVC for an and gate?

7.1.82 What is the difference b/w sync and wait?

7.1.83 Does a tcm execute on its own?

7.1.84 What is the difference b/w on the fly generation and static generation?

7.1.85 What is the significance or purpose of sys.any event?

7.1.86 What is the usage of "fail" temporal operator? / How can "fail" temporal operator succeed?

7.1.87 Consider the following : event t1 is(@a or @b) @clk_rise;

7.1.88 How are soft constraints evaluated? What happens when a softeconstraint applied conflicts with the already applied constraint?

7.1.89 How to apply a constraint conditionally?

7.1.90 Can two hard constraints be applied on a field?

7.1.91 How do I constrain the distribution of values to a single field?

7.1.92 Where is "me" and "it" use while specifying constraints?

7.1.93 What is the advantage or use of coverage based methodology?

7.1.94 What is event based coverage?

7.1.95 How can you cover rtl state machine transitions using specman?

7.1.96 What is the usage of events?

7.1.97 Can "wait" and "sync" be use outside a TCM?

7.1.98 How do I check for a number of occurrences of a event and if it exceeds a pre-defined number then flag an error?

7.1.99 What is the difference b/w Regular GC and OTF GC?

7.1.100Is any garbage collection done during pre-run generation phase?

7.1.101How can one control the memory setting options?

7.1.102When should "is " extension be use for extending the methods?

7.1.103What is the max.number of parameters that can be defined inside a method?

7.1.104Can I use wait and sync constructs inside a method?

7.1.105Once a method is extended using is also when does the extended part of the method execute?

7.1.106 What are the differences between structs and units?

A warm up question. Units are static objects that exist from the start of the simulation right up to its end, while structs are created on the fly and automatically destroyed by the garbage collection mechanism when they can not be accesses anymore. Units are used to store data that is required by many entities, because it is easy to know where they are and reach them from anywhere in an environment (for example using such unit specific methods as get_enclosing_unit(). They are also used to connect to DUT interfaces (BFMs) because these interfaces are active from reset to reset (which is normally equivalent to from start to end of simulation), and for other fixed environment elements such as scoreboards.

7.1.107 What are the special unit related fields and methods?

The most important method (in fact pseudo method) related to units is get_enclosing_unit().

The most widely used field in a unit is its HDL path. It makes it possible to use relative paths in order to access DUT signals, which is crucial if a move from block level to chip level is planned.

7.1.108 How can you pass a struct by reference in e?

The question is phrased in a tricky way because passing by reference is the default and only possible way to pass structs in e. In other words, every time you call a method with a struct parameter, the method gets the original copy, and any modifications it does will persist even after it returns. To pass a struct by value you can use deep_copy() to clone it. In C/C++ you could prevent a calling function from modifying a parameter by defining that function parameter as const. It is too bad that e doesn't support const as well…

7.1.109 How do you pass basic types by reference?

In the case of basic types such as bool, int, uint, etc. you can in fact choose between the two forms. If you prefix the method parameter in the method definition with a star, then it will be passed by reference (modifications persist). Otherwise, it will be passed by value.

My interviewer then asked me what happens when you prefix a struct parameter with a star…turns out that in that case the called method can completely delete the struct and replace it with another or with nothing at all (for example by calling “gen” on the parameter, or by assigning NULL to it)

7.1.110 What is the use of coverage per instance? How can you use it to prevent the creation of fake coverage holes?

Specman 5.1 supports two main types of per instance coverage, referred to, somewhat confusingly, as – “cover group per instance coverage” and “item per instance coverage” (previous versions support only the “item per instance coverage”). Here are two examples of situations where the two types perfectly fit.

Using a “cover group per instance coverage” would be helpful if you have several interfaces that implement the same protocol, but were not written by the same designer, or better put, are not replicated instances of the same code. In this case you will have to check that each of the interfaces obeys to all of the protocol rules. So, if you have interfaces A and B that have to comply with rules 1 and 2, then you would have to check that 1 and 2 work for A and that 1 and 2 work for B. On the other hand, if your interfaces are just replicated instances of the exact same code, then if you check that 1 works for A and 2 works for B, you can say you’re done. This last option corresponds to the way in which e coverage works by default.

Using “item per instance coverage” is beneficial if you have several subtypes of the same interface. For example, consider a situation where someone wrote a bus interface that can be synthesizes either with a Simple option, in which case it will support only atomic reads and writes and will require fewer gates, or with an Extended option in which case it will also support bursts. It goes without saying that you wouldn’t want to collect burst coverage on the Simple interfaces, for the obvious reason that no bursts should happen there (and most probably a burst indication signal doesn’t even exist). The “item per instance coverage” option is used to prevent Specman from collecting burst coverage on the simple interfaces, and thus creating false “coverage holes”. This is shown in the code below:

```
<'
type InterfaceKind : [Simple, Burst];

unit BusInterface {
  kind : InterfaceKind;

  rd : bit;
  wr : bit;

  event transactionDone;
  cover transactionDone is {
    item kind;
    item rd;
    item wr;
  };
};

// written this way you will have
// a fake coverage hole for Simple
// BusInterfaces because the signal
// burst does not exist for these
// and will not be sampled...

extend Burst BusInterface {
  burst : bit;

  cover transactionDone is also {
    item burst;
  };
};

// written this way you will have no
// fake coverage hole

extend Burst BusInterface {
  burst : bit;

  cover transactionDone(kind==Burst) is also {
    item burst;
  };
};

// code to check that I'm not bullshiting
// you
extend sys {
  busInterfaces : list of BusInterface is instance;
  keep for each in busInterfaces {
    index == 0 => it.kind == Simple;
    index == 1 => it.kind == Burst;
  }; // keep for each i...

  post_generate() is also {
    for each in busInterfaces {
      for i from 0 to 50 {
        gen it.rd; gen it.wr;
        if it is a Burst BusInterface (burstIf) {
          gen burstIf.burst;
        };
        emit it.transactionDone;
      };
    };
  };
};
'>
```

Two comments as an end note to this lengthy explanation:

I. If you find the terms “cover group per instance” and “item per instance” confusing, you are not alone. The confusion arises because it is not really clear what instance the words “per instance” refer to: while it seems that in “cover group per instance” they refer to an instance of an HDL module or an e unit, it is obvious that this is not the case for “item per instance”, where coverage from several instances of the same subtype will be grouped together. I guess you are even more confused after reading this sentence than before….

II. Unfortunately you can not use “cover group per instance” and “item per instance” together. This would be very useful where you have a single protocol that can be extended and that was implemented by several independent designers. This comment is written with the OCP bus interface in mind, where each IP vendor implements a subset of the protocol separately.

7.1.111 What logical structure (object structure) and physical structure (file structure) are defined by the eRM? Which conventions are defined by the eRM? What type of object is used as a container for an eVC?

7.1.112What would you change in Specman?

A nice one ha? I decided to dedicate it a separate entry.

7.1.113 What will be the output if following code is loaded and test command is issued into Specman ?

```
<'
struct x
{
y : uint;
keep soft y == 8;
};
extend sys
{
run() is also
{
var x : x;
gen x;
var a : uint = 2;
outf ("a = %d x.y = %d\n", a, x.y);
update_field(a);
update_struct(x);
outf ("a = %d x.y = %d\n", a, x.y);
};
update_field(a: uint) is
{
a = 4;
};
update_struct(x: x) is
{
x.y = 4;
};
};
'>
```

Answer:

```
a = 2 x.y = 8
a = 2 x.y = 4
```

Here, variable uint a is not updated and struct x is updated. Why is so? Because in specman, unlike user defined data types (e.g. structs, units), all variables with standard data types are passed by value in the methods. All user defined data types (like structs) are passed by reference. So, whenever, called methods update the passed structs, it is actually updating the original struct and not its copy.

If one wants user defined data types to be passed by value then he/she can use deep_copy to copy the existing variable and pass it to the method as shown below.

```
<'
struct x
{
y : uint;
keep soft y == 8;
};
extend sys
{
run() is also
{
var x : x;
gen x;
var a : uint = 2;
outf ("a = %d x.y = %d\n", a, x.y);
update_field(a);
update_struct(deep_copy(x));
outf ("a = %d x.y = %d\n", a, x.y);
};
update_field(a: uint) is
{
```

```
a = 4;
};
update_struct(x: x) is
{
x.y = 4;
};
};
'>
```

7.1.114 What is the difference between inheritance implemented by "when" construct and implemented by "like" construct?

Answer:

like inheritance is the concept of the object oriented programming (OOP) where as when inheritance is the concept of aspect oriented programming (AOP). For more detail on AOP, please refer Aspect-Oriented Programming with the e Verification Language book written by David Robinson (Verilab).

like construct is used when someone wants to derive a child object from the already defined struct/unit. This derived child will have new struct/unit name. When someone derive the child object using when construct, the base name of the child will remain the same.

Another difference between like and when construct is that, once the child is derived using like inheritance, one can not add extra fields in the parent struct/unit, wherease, if child is derived using when construct, parent struct/unit can have extra fields.

Take a look at the following example. packet_valid field is added in the parent struct packet_s after the child is derived using like inheritance. This is not allowed and Specman will issue an error during loading phase.

```
<'
type packet_type_t : [GOOD, BAD, UGLY];
struct packet_s
{
kind : packet_type_t;
};
struct good_packet_s like packet_s
{
packet_size : uint (bits: 5);
};
extend packet_s
{
packet_valid : bool;
ack() is
{
outf ("NOTE :: This is packet %s\n", me);
};
};
extend sys
{
packet : packet_s;
good_packet : good_packet_s;
};
'>
```

Specman will issue following error.

*** Error: Cannot add new field 'packet_valid' to struct 'packet_s': it has like children (e.g. 'good_packet_s') with fields.

Note that ack() method will be added in the parent struct packet_s even though child is derived using like inheritance.

Posted by Sandeep Gor at 10:06

Labels: Interview

1 comments:

Rashid said...

Hi Sandeep,

Your blog is very interesting.

For Question No.2

To achieve pass by reference for a standard data type we can prefix '*' with data type

Fo ex:

```
<'
struct x{
y : uint;
keep soft y == 8;
};

extend sys{
```

```
run() is also{
var x : x;
gen x;
var a : uint = 2;
outf ("a = %d x.y = %d\n", a, x.y);
update_field(a);
update_struct(deep_copy(x));
outf ("a = %d x.y = %d\n", a, x.y);
};

update_field(a: *uint) is{
a = 4;
};

update_struct(x: x) is{
x.y = 4;
};
};

'>
```

Answer

```
Starting the test ...
Running the test ...
a = 2 x.y = 8
a = 4 x.y = 8
```

8 MISC

8.1.1 What is a SoC (System On Chip), ASIC, “full custom chip”, and an FPGA?

There are no precise definitions

First, 15 years ago, people were unclear on exactly what VLSI meant. Was it 50000 gates? 100000 gates? was is just anything bigger than LSI? My professor simply told me that; VLSI is a level of complexity and integration in a chip that demands Electronic Design Automation tools in order to succeed. In other words, big enough that manually drawing lots of little blue, red and green lines is too much for a human to reasonably do. I think that, likewise, SoC is that level of integration onto a chip that demands more expertise beyond traditional skills of electronics. In other words, pulling off a SoC demands Hardware, Software, and Systems Engineering talent. So, trivially, SoCs aggressively combine HW/SW on a single chip. Maybe more pragmatically, SoC just means that ASIC and Software folks are learning a little bit more about each other’s techniques and tools than they did before. Two other interpretations of SoC are 1) a chip that integrates various IP (Intellectual Property) blocks on it and is thus highly centered with issues like Reuse, and 2) a chip integrating multiple classes of electronic circuitry such as Digital CMOS, mixed-signal digital and analog (e.g. sensors, modulators, A/Ds), DRAM memory, high voltage power, etc.

ASIC stands for “Application Specific Integrated Circuit”. A chip designed for a specific application. Usually, I think people associate ASICs with the Standard Cell design methodology. Standard Cell design and the typical “ASIC flow” usually means that designers are using Hardware Description Languages, Synthesis and a library of primitive cells (e.g. libraries containing AND, NAND, OR, NOR, NOT, FLIP-FLOP, LATCH, ADDER, BUFFER, PAD cells that are wired together (real libraries are not this simple, but you get the idea..). Design usually is NOT done at a transistor level. There is a high reliance on automated tools because the assumption is that the chip is being made for a SPECIFIC APPLICATION where time is of the essence. But, the chip is manufactured from scratch in that no pre-made circuitry is being programmed or reused. ASIC designer may, or may not, even be aware of the locations of various pieces of circuitry on the chip since the tools do much of the construction, placement and wiring of all the little pieces.

Full Custom, in contrast to ASIC (or Standard Cell), means that every geometric feature going onto the chip being designed (think of those pretty chip pictures we have all seen) is controlled, more or less, by the human design. Automated tools are certainly used to wire up different parts of the circuit and maybe even manipulate (repeat, rotate, etc.) sections of the chip. But, the human designer is actively engaged with the physical features of the circuitry. Higher human crafting and less reliance on standard cells takes more time and implies higher NRE costs, but lowers RE costs for standard parts like memories, processors, uarts, etc.

FPGAs, or Field Programmable Gate Arrays are completely designed chips that designers load a programming pattern into to achieve a specific digital function. A bit pattern (almost like a software program) is loaded into the already manufactured device which essentially interconnects lots of available gates to meet the designers purposes. FPGAs are sometimes thought of as a “Sea of Gates” where the designer specifies how they are connected. FPGA designers often use many of the same tools that ASIC designers use, even though the FPGA is inherently more flexible.

All these things can be intermixed in hybrid sorts of ways. For example, FPGAs are now available that have microprocessor embedded within them which were designed in a full custom manner, all of which now demands "SoC" types of HW/SW integration skills from the designer.

8.1.2 How do I model Analog and Mixed-Signal blocks in Verilog?

First, this is a big area. Analog and Mixed-Signal designers use tools like Spice to fully characterize and model their designs. My only involvement with Mixed-Signal blocks has been to utilize behavioral models of things like PLLs, A/Ds, D/As within a larger SoC. There are some specific Verilog tricks to this which is what this FAQ is about (I do not wish to trivialize true Mixed-Signal methodology, but us chip-level folks need to know this trick).

A mixed-signal behavioral model might model the digital and analog input/output behavior of, for example, a D/A (Digital to Analog Converter). So, digital input in and analog voltage out. Things to model might be the timing (say, the D/A utilizes an internal Success Approximation algorithm), output range based on power supply voltages, voltage biases, etc. A behavioral model may not have any knowledge of the physical layout and therefore may not offer any fidelity whatsoever in terms of noise, interface, cross-talk, etc. A model might be parameterized given a specific characterization for a block. Be very careful about the assumptions and limitations of the model!

Issue #1; how do we model analog voltages in Verilog. Answer: use the Verilog real data type, declare "analog wires" as wire[63:0] in order to use a 64-bit floating-type represenation, and use the built-in PLI functions:

```
$rtoi converts reals to integers w/truncation e.g. 123.45 -> 123
$itor converts integers to reals e.g. 123 -> 123.0
$realtobits converts reals to 64-bit vector
$bitstoreal converts bit pattern to real
```

That was a lot. This is a trick to be used in vanilla Verilog. The 64-bit wire is simply a ways to actually interface to the ports of the mixed-signal block. In other words, our example D/A module may have an output called AOUT which is a voltage. Verilog does not allow us to declare an output port of type REAL. So, instead declare AOUT like this:

```
module dtoa (clk, reset..... aout.....);
....
```

```
wire [63:0]       aout;     // Analog output
....
```

We use 64 bits because we can use floating-point numbers to represent out voltage output (e.g. 1.22x10-3 for 1.22 millivolts). The floating-point value is relevant only to Verilog and your workstation and processor, and the IEEE floating-point format has NOTHING to do with the D/A implementation. Note the disconnect in terms of the netlist itself. The physical "netlist" that you might see in GDS may have a single metal interconnect that is AOUT, and obviously NOT 64 metal wires. Again, this is a trick. The 64-bit bus is only for wiring. You may have to do some quick netlist substitutions when you hand off a netlist.

In Verilog, the real data type is basically a floating-point number (e.g. like double in C). If you want to model an analog value either within the mixed-signal behavorial model, or externally in the system testbench (e.g. the sensor or actuator), use the real data type. You can convert back and forth between real and your wire [63:0] using the PLI functions listed above. A trivial D/A model could simply take the digital input value, convert it to real, scale it according to some #defines, and output the value on AOUT as the 64-bit "psuedo-analog" value. Your testbench can then do the reverse and print out the value, or whatever. More sophisticated models can model the Successive Approximation algorithm, employ look-ups, equations, etc. etc.

That's it. If you are getting a mixed-signal block from a vendor, then you may also receive (or you should ask for) the behavioral Verilog models for the IP.

8.1.3 How do I interface one clock domain to another?

Depends... Here are 3 methods that I've used lately. First, "synchronizers" are very common. A synchronizer is just a flip-flop that accepts signals from one clock domain but is clocked by the local clock domain. People argue about how many levels of these synchronizers to use (2 are common). The idea is that the incoming signal could hit the flop and violate a setup/hold time. When this happens, the flop could go "metastable" and basically "ring" for an indeterminate time. A series of synchronizers helps isolate the main circuitry from potential ringing and resynchronizes the inputs to the new clock domain. There are many, many papers on "Metastability" where you will find equations involving the two frequencies, setup/hold times of the flops, rise times, and MTBF. Some ASIC and FPGA libraries actually have flip-flops intended for use as synchronizers (they have tighter setup/hold times and higher internal gains). Another approach is to use a Dual-Port RAM at the interface (e.g. usually a FIFO). DPRAMs will usually have separate clocks for each domain. DPRAMs can also help with data rate conversions (e.g. smooth out burstiness in an input for example). Another scenario that arises is handshaking across an asynchronous interface. For example, one domain provides an AVAILABLE signal indicating data and the other domain acknowledges this data with some sort of ACKNOWLEDGE signal. Often in these types of interfaces, there are NO explicit clocks. Very fast handshaking interfaces can be designed by designing to the edges in the protocol and using standard flip-flops using the clock inputs to detect edges and then asynchronous sets/clears to reset them. This is how, for example, a narrow pulse can be detected by a clock domain with a much slower period. Well, that wasn't very clear and requires a schematic. ISD Magazine has had several good articles on this sort of thing in the last couple of years. Also, look for several papers that have been presented at ESNUG (try www.deepchip.com). Be careful out there, but there are techniques work.

Some good insight into FIFOs can be found at the XILINX site (XAPP 051) or just do some searches (also note some free FIFO IP at www.free-ip.com).

8.1.4 How do I synthesize Verilog into gates with Synopsys?

The answer can, of course, occupy several lifetimes to completely answer.. BUT.. a straight-forward Verilog module can be very easily synthesized using Design Compiler (e.g. dc_shell). Most ASIC projects will create very elaborate synthesis scripts, CSH scripts, Makefiles, etc. This is all important in order automate the process and generalize the synthesis methodology for an ASIC project or an organization. BUT don't let this stop you from creating your own simple dc_shell experiments!

Let's say you create a Verilog module named foo.v that has a single clock input named 'clk'. You want to synthesize it so that you know it is synthesizable, know how big it is, how fast it is, etc. etc. Try this:

```
target_library = { CORELIB.db } <--- This part you need to get from your vendor...
read -format verilog foo.v
create_clock -name clk -period 37.0
set_clock_skew -uncertainty 0.4 clk
set_input_delay 1.0 -clock clk all_inputs() - clk - reset
set_output_delay 1.0 -clock clk all_outputs()
compile
report_area
report_timing
write -format db -hierarchy -output foo.db
write -format verilog -hierarchy -output foo.vg
quit
```

You can enter all this in interactively, or put it into a file called 'synth_foo.scr' and then enter:

```
dc_shell -f synth_foo.scr
```

You can spend your life learning more and more Synopsys and synthesis-related commands and techniques, but don't be afraid to begin using these simple commands.

8.1.5 What is "Scan" ?

Scan Insertion and ATPG helps test ASICs (e.g. chips) during manufacture. If you know what JTAG boundary scan is, then Scan is the same idea except that it is done inside the chip instead of on the entire board. Scan tests for defects in the chip's circuitry after it is manufactured (e.g. Scan does not help you test whether your Design functions as intended). ASIC designers usually implement the scan themselves and occurs just after synthesis. ATPG (Automated Test Pattern Generation) refers to the creation of "Test Vectors" that the Scan circuitry enables to be introduced into the chip. Here's a brief summary:

- Scan Insertion is done by a tool and results in all (or most) of your design's flip-flops to be replaced by special "Scan Flip-flops". Scan flops have additional inputs/outputs that allow them to be configured into a "chain" (e.g. a big shift register) when the chip is put into a test mode.
- The Scan flip-flops are connected up into a chain (perhaps multiple chains)
- The ATPG tool, which knows about the scan chain you've created, generates a series of test vectors.
- The ATPG test vectors include both "Stimulus" and "Expected" bit patterns. These bit vectors are shifted into the chip on the scan chains, and the chips reaction to the stimulus is shifted back out again.
- The ATE (Automated Test Equipment) at the chip factory can put the chip into the scan test mode, and apply the test vectors. If any vectors do not match, then the chip is defective and it is thrown away.
- Scan/ATPG tools will strive to maximize the "coverage" of the ATPG vectors. In other words, given some measure of the total number of nodes in the chip that could be faulty (shorted, grounded, "stuck at 1", "stuck at 0"), what percentage of them can be detected with the ATPG vectors? Scan is a good technology and can achive high coverage in the 90% range.
- Scan testing does not solve all test problems. Scan testing typically does not test memories (no flip-flops!), needs a gate-level netlist to work with, and can take a long time to run on the ATE.
- FPGA designers may be unfamiliar with scan since FPGA testing has already been done by the FPGA manufacturer. ASIC designers do not have this luxury and must handle all the manufacturing test details themselves.
-

8.1.6 How do I generate a random number either in a C program (e.g. if there is no access to standard rand() function) or in hardware?

Use an LFSR (Linear Feedback Shift Register). This is a useful building block for many things including random numbers. Check out the program, myrand.v in my File Archive.

8.1.7 I need to sample an input or output something at different rates, but I need to vary the rate? What's a clean way to do this?

Many, many problems have this sort of variable rate requirement, yet we are usually constrained with a constant clock frequency. One trick is to implement a digital NCO (Numerically Controlled Oscillator). An NCO is actually very simple and, while it is most naturally understood as hardware, it also can be constructed in software. The NCO, quite simply, is an accumulator where you keep adding a fixed value on every clock (e.g. at a constant clock frequency). When the NCO "wraps", you sample your input or do your action. By adjusting the value added to the accumulator each clock, you finely tune the AVERAGE frequency of that wrap event. Now - you may have realized that the wrapping event may have lots of jitter on it. True, but you may use the wrap to increment yet another counter where each additional Divide-by-2 bit reduces this jitter. The DDS is a related technique. I have two examples showing both an NCOs and a DDS in my File Archive. This is tricky to grasp at first, but tremendously powerful once you have it in your bag of tricks. NCOs also relate to digital PLLs, Timing Recovery, TDMA and other "variable rate" phenomena.

8.1.8 What is Boundary Scan?

Boundary scan is a board level design technique that provides test access to the input and output pads of ICs on PCBs. Boundary scan modifies the IO circuitry of individual ICs, so that the input and output pads of every boundary scan IC can be connected to form one or more serial boundary scan chains.

Boundary scan is used to check for shorts and opens in the interconnect between ICs on PCBs. It performs the checks by using the boundary scan chains to pass logic values b/w the pads of different ICs.

At the top level the boundary scan logic has 3 modules: the Test Access Port (TAP), the instruction and data registers.

The TAP, an external I/F between the chip and the board, consists of Test Mode Select (TMS), Test Clock (TCK), Test Data In (TDI) and Test Data Out (TDO) - that control the state of the boundary scan test logic and that provide the serial access to the instruction and data modules.

8.1.9 What are the various Design constraints used while performing Synthesis for a design?

1. Create the clocks (frequency, duty-cycle).

2. Define the transition-time requirements for the input-ports.

3. Specify the load values for the output ports

4. For the inputs and the output specify the delay values(input delay and ouput delay), which are already consumed by the neighbour chip.

5. Specify the case-setting (in case of a mux) to report the timing to a specific paths.

6. Specify the false-paths in the design

7. Specify the multi-cycle paths in the design.

8. Specify the clock-uncertainity values(w.r.t jitter and the margin values for setup/hold).

8.1.10 What does formal verification mean?

Formal verification uses Mathematical techniquest by prooving the design through assertions or properties. Correctness of the design can be achieved through assertions with out the necessity for simulations. The methods of formal verification are

1. Equivalence checking In this method of checking the designs are compared based on mathematical equations and compared whether they are equal or not .

 * Original RTL vs Modified RTL

 * RTL vs Netlist

 * Golden Netlist vs Modified/Edited Netlist

 * Synthesis Netlist vs Place and route Netlist

Remember : Formal verification doesnt check for functionality of the RTL code. It will be only checking the equivalence.

2. Model checking Property specification languages like PSL or SVA, are formally analyzed to see if they are always true for a design. This can exhaustively prove if a property is correct, but does tend to suffer from state-space explosion: the time to analyse a design is directly propotional to the amount of states.

8.1.11 We have multiple instances in RTL(Register Transfer Language), do you do anything special during synthesis stage?

While writing RTL(Register Transfer language),say in verilog or in VHDL language, we dont write the same module functionality again and again, we use a concept called as instantiation, where in as per the language, the instanciation of a module will behave like the parent module in terms of functionality, where during synthesis stage we need the full code so that the synthesis tool can study the logic , structure and map it to the library cells, so we use a command in synthesis , called as "UNIQUIFY" which will replace the instantiations with the real logic, because once we are in a synthesis stages we have to visualize as real cells and no more modelling just for functionality alone, we need to visualize in-terms of physical world as well.

8.1.12 What do you call an event and when do you call an assertion?

Assertion based Verification Tools, checks whether a statement holds a defined property or not, whereas, Event based Simulators, checks whether there is change in any event, say for every edge of a clock whether there is some activity in a signal or not, in case of an asynchronous designs, checks whether a signal is enabled or not.

8.1.13 What are the measures or precautions to be taken in the Design when the chip has both analog and digital portions?

As today's IC has analog components also inbuilt, some design practices are required for optimal integration. Ensure in the floor-planning stage that the analog block and the digital block are not siting close-by, to reduce the noise. Ensure that there exists separate ground for digital and analog ground to reduce the noise. Place appropriate guard-rings around the analog-macro's. Incorporating in-built DAC-ADC converters, allows us to test the analog portion using digital testers in an analog loop-back fashion. Perform techniques like clock-dithering for the digital portion.

8.1.14 How to increse simulation speed

First figure out what is eating away your cpu cycles. Is it

1. compile time - Use a Make file to compile only files with changes and not all files else just use the incremental compile option in your simulator

2. loading/ elaborate - If you are using nc-sim / vcs then you are allowed to do incremental elaborate by using -update option when elab state. If you use modelsim - you cannot gain any time here

3. SImulating/ Running - I will give sub points for this

a) Check with the `timescale directive most people use `timescale 100ps/1ps , but their code never uses any delay at all or uses delays in ns . this is a overkill which will slow down your sim speed check your `timescale and check the min. resolution needed for this just stick with this. like `timescale 1ns/1ns

b) Memory declarations in verilog, most newbies decalre huge mem. using verilog like reg [32:0] men [1023:0] . If you do have such declaration in your testbench remove them and use sparce models . DONOT touch it in design , if you do make sure you make required adjustments before synthesis (I wouldnot be too intrested in changing things in design)

c) In your test case look at how you wait for any condition to occur does your bfm / models keep scanning at every event of clock , these things can be speed by using event based waits rather than check at each clock edge.

d) Hardware - Are you running a huge design on a sissy system . Well you cannot do much except change you machine.

8.1.15 Adrress decoder is having bug. If you try to write into a location it is get witten into the other location.

- **How do you find out the flaw, which of the address getting written wrongly.**

Fill the entire memory once with either random data or sequential data then after filling completely read the data from the memory. Compare the write data and read data, you can find the location if the data mismatches.

Consider a example address decoder which is mapping the transactions to the different memories. Let us assume there is bug in the address decoder when accessing the a 256 byte memory.when ever we want to write the data to the address location AA it is getting written to the location BB.

If we write all the locations continueosly then the write data to the location AA will be written to to the location BB and data to the location BB again written to the location BB thus overwriting the previous value.

When we compare the write and read data there will be mismatch at position AA, since the value is overridden already.

- **If the address decoder, let say you want to write/read to address A, B but the address decoder is mapping to locations B, A how do you find out the problem.**

I think there is no problem with this, since those two acting as two different location. For the user it doesn't matter in which location the data is stored, unless a different data is retrived when it is read back again.

8.1.16 Question on maximum frequency of opereation of a frequency divider. The values given Ts(1,1), Tpdcomb(4,2) and Tcq(3,2). Here two values given for each parameter indicates the maximum and minimum vaules for that, since the delay may vary due to temperature. Where Ts – set up time,Tpdcomb- Propagation delay of the combo logic and Tcq - clock to q dealy of the flop.

The answer is 200 Mhz. since considering the minimum delays, the time takes is 1 + 2 + 3 = 5 ns.

8.1.17 Write the veilog code for a latch

```
Always ( en or d)
Begin
 If(en)
  Q = d;
End
```

8.1.18 How do you solve problem of hold violation after the silicon came out.

8.1.19 How the temperature effecting the delays in a chip

The delays are directly proportional to the temperature.As the temperatue increases the delays are increases and chip will mal function.

8.1.20 Tell me the difference between the Static rams and dynaminc rams.

Static RAM:

- ❖ It is costly since each bit of memory will take around 6 CMOS transistors.
- ❖ More speedy since transistors are used.

Dynamic RAM:

- ❖ It is cheap since it is made up of capcitors.

8.1.21 Why do you need netlist simulation even though Formality and STA analysis done.