VLSI Interview Questions with Answers

Sam Sony

Content

Questions and Answers are divided into following categories in the respective order.

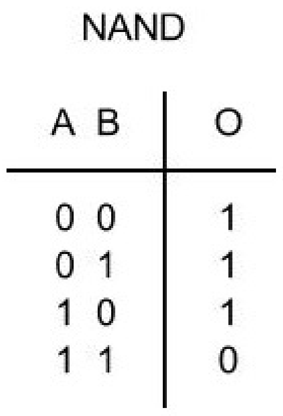
Digital Design Questions & Answers.

Question D1): You can use just a NAND gate, How many ways can you come up with an inverter ?
Answer D1):
This is a test for your very basic gate understanding. Very crucial, you need to get your fundamentals in order.

For all such questions, you should start by drawing truth tables. If needed go through all basic truth tables for inverter, and, or, nand, nor, xor, xnor gates.

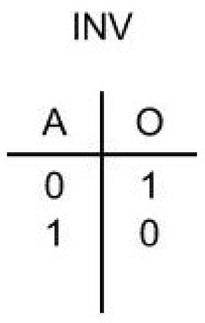
Draw the truth table of what you have and the table for what you want to achieve. Check if you could find simple optimization like following example to get what you need. If it becomes difficult, next method is to use Karnaugh Map.

We have NAND gate and we need to make INV. Truth tables for both are here.



NAND

A	B	O
0	0	1
0	1	1
1	0	1
1	1	0

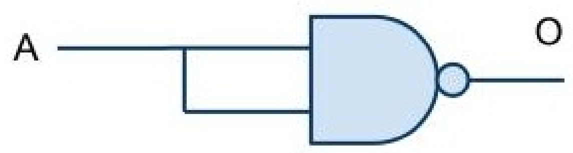


INV

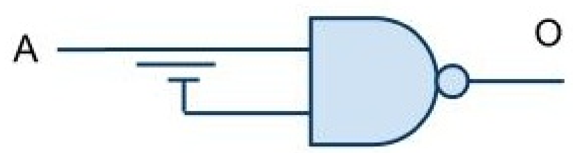
A	O
0	1
1	0

We can see that when both inputs A & B are ‘0’ the output is ‘1’ and vice-versa, i.e. when both inputs are ‘1’, the output is ‘0’, which is similar to the INV behavior, hence one possibility is to short both inputs of NAND to achieve INV. Following are the two ways to make inverter out of NAND.

If you observe NAND table closely you see that when A input is ‘1’, B input appears inverted at output. Which brings us to other possibility that if tie A input of NAND to ‘1’ we get INV.



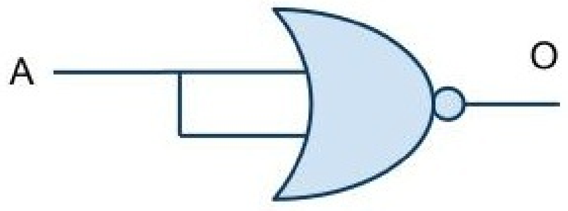
A	O
0	1
1	0



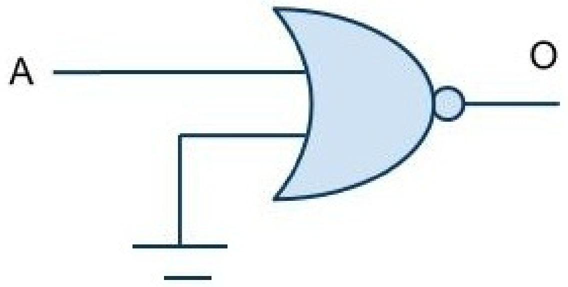
A	O
0	1
1	0

Figure D1. Two input NAND gate as inverter.

Question D2): Make an INVERTER using only 2 input NOR gates.
Answer D2):
Go through similar exercise as we went through in previous question. Following are the two ways that you get to make inverter out of NOR.



A	O
0	1
1	0



A	O
0	1
1	0

Figure D2. Two input NOR gate as inverter.

Question D3): Make an OR gate, using just NAND gates.
Answer D3):
You need to know the equivalent representations of NOR gates, which we can come up using De Morgan's law
De Morgans law : (A+B)bar = (A)bar * (B)bar
Which is as shown in the figure below.

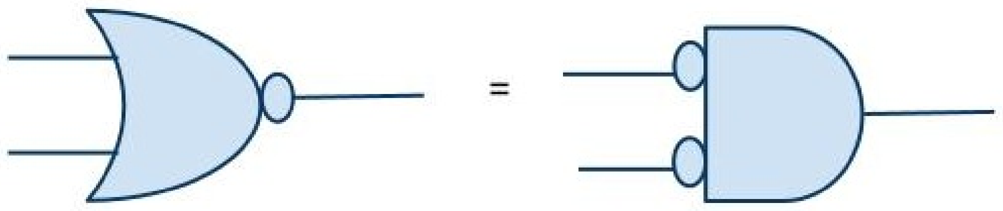


Figure D3. De Morgans law.

We need to get an OR gate. We'll work backwards, hence we'll start with OR gate and go through transformation steps using De Morgans law eventually converting to NAND gates as described below.
Following is the sequence of simplification. At the end we're left with just NAND and INVERTER, we know from previous questions that we can easily convert NAND into an INVERTER.

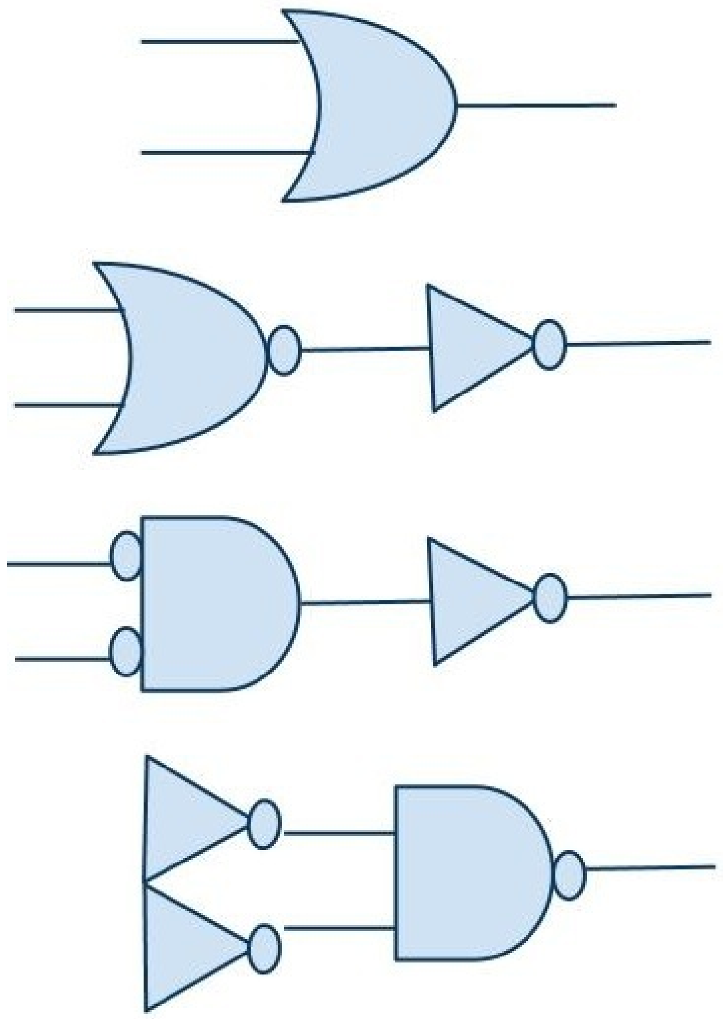


Figure D4. NAND gates as an OR gate

Question D4): Make an INVERTER using a 2 to 1 MUX

Answer D4):

When it comes to dealing with MUXes, remember the equation of a 2:1 MUX. Utilize it to try and come up with the desired behavior.

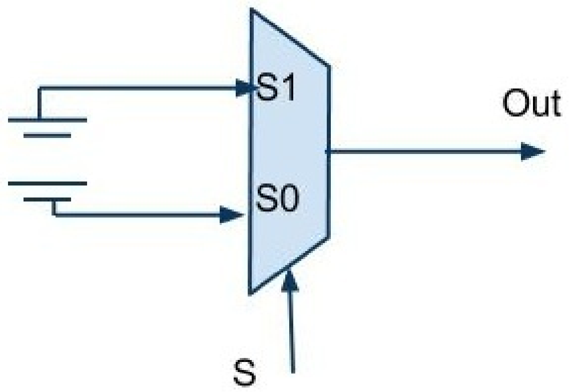
Equation of a MUX is : Out = S * A + (S)bar * B

If we replace A with ‘0’ and B with ‘1’ we get INV like functionality.

Out = S * 0 + (S)bar * 1

Out = 0 + (S)bar = (S)bar

You can verify using the truth tables. Following figure describes INVERTER using a 2 to 1 MUX

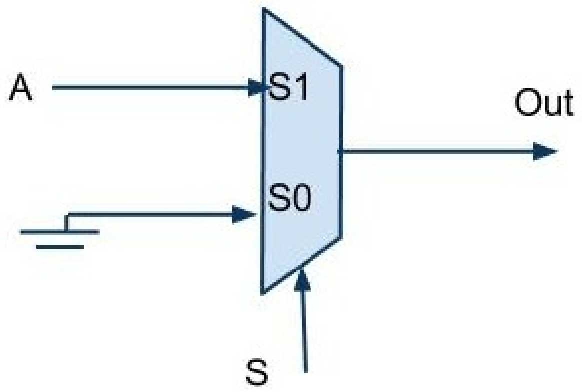


S	Out
0	1
1	0

Figure D5. INV using 2:1 MUX

Question D5): Make an AND gate using 2 to 1 MUX
Answer D5):
Equation of a MUX is : Out = S * A + (S)bar * B
If we tie B to zero, which is what we do down below. We get

Out = S * A + (S)bar * 0 = S * A + 0 = S * A



S	A	Out
0	0	0
0	1	0
1	0	0
1	1	1

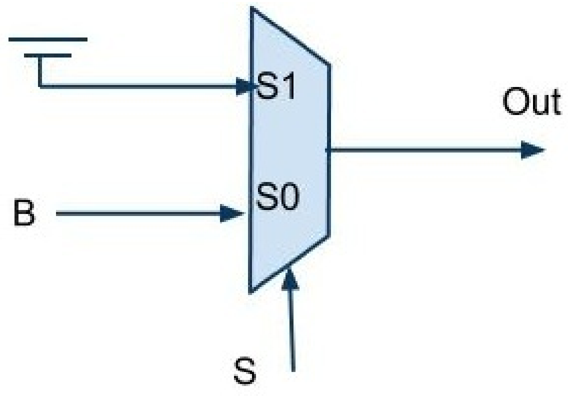
Figure D6. AND using 2:1 MUX

Question D6): Make a NAND gate using 2 to 1 MUX
Answer D6):
In previous two questions we made AND & INVERTER out of 2 to 1 MUX, we can combine both and come up with NAND gate.

Question D7): Make an OR gate using 2 to 1 MUX

Answer D7):
Equation of a MUX is : Out = S * A + (S)bar * B
If we tie A to ‘1’, which is what we do down below. We get

Out = S * 1 + (S)bar * B = S + (S)bar * B = S + B



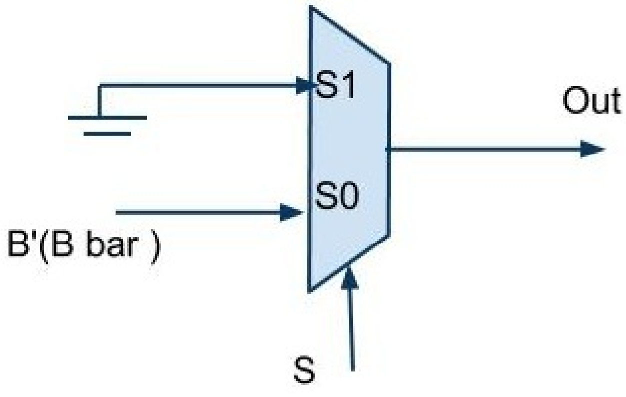
S B	Out
0 0	0
0 1	1
1 0	1
1 1	1

Figure D7. OR using 2:1 MUX

Question D8): Make an NOR gate using 2 to 1 MUX
Answer D8):
Equation of a MUX is : Out = S * A + (S)bar * B

If we tie A to ‘0’, and if we use (B)bar at input instead of B, We get

Out = S * 0 + (S)bar * (B)bar = 0 + (S)bar * (B)bar = (S)bar * (B)bar = (S + B)bar



S B	Out
0 0	1
0 1	0
1 0	0
1 1	0

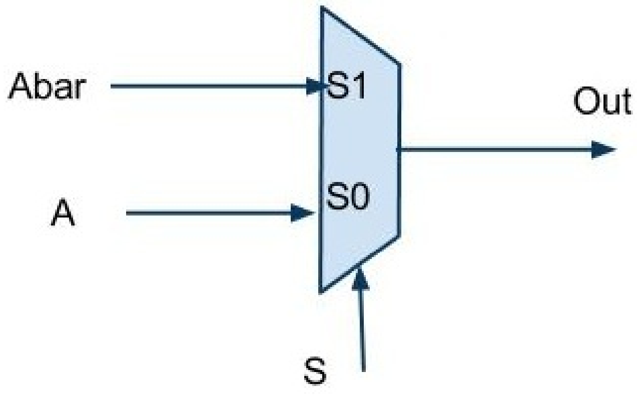
Figure D8. NOR using 2:1 MUX

Other option is to take MUX OR gate and MUX INVERTER and combine them.

Question D9): Make an XOR gate using 2 to 1 MUX
Answer D9):
Equation of a MUX is : Out = S * A + (S)bar * B

If we tie (A)bar to A, and if we use A at input instead of B, We get

Out = S * (A)bar + (S)bar * A
Which is equation for XOR gate.



S	A	Out
0	0	0
0	1	1
1	0	1
1	1	0

Figure D9. XOR using 2:1 MUX

Question D10): Design a full adder with 2-1 mux
Answer D10): Full Adder can be implemented by two half adder; a half adder can be implemented by a XOR and AND gate. XOR and AND gate can be implemented by 2:1 MUX.

Question D11): Simplify logic : MUX with D1 input tied to ground, and inverter at the select input.
Answer D11):

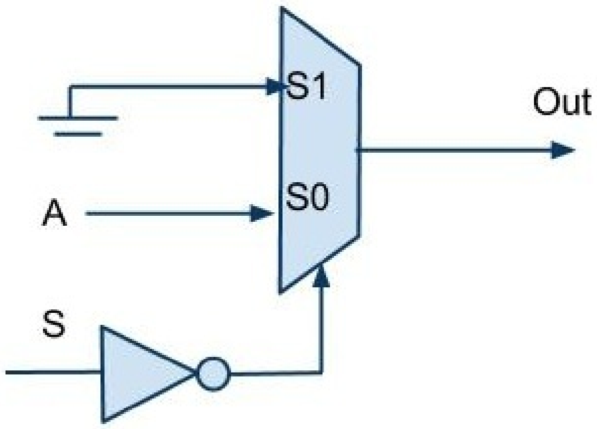


Figure D11. Logic simplification.

Here you can see that Out = (S)bar * 0 + S * A = S * A
So this is just an AND gate.

Question D12): Form a 2 to 1 MUX using only NAND & inverters.
Answer D12):
Equation of a MUX is : Out = S * A + (S)bar * B

We can use DeMorgans law and above mentioned equation to turn it into NAND and INV gates. Following is the sequence of changes to make it NAND only MUX

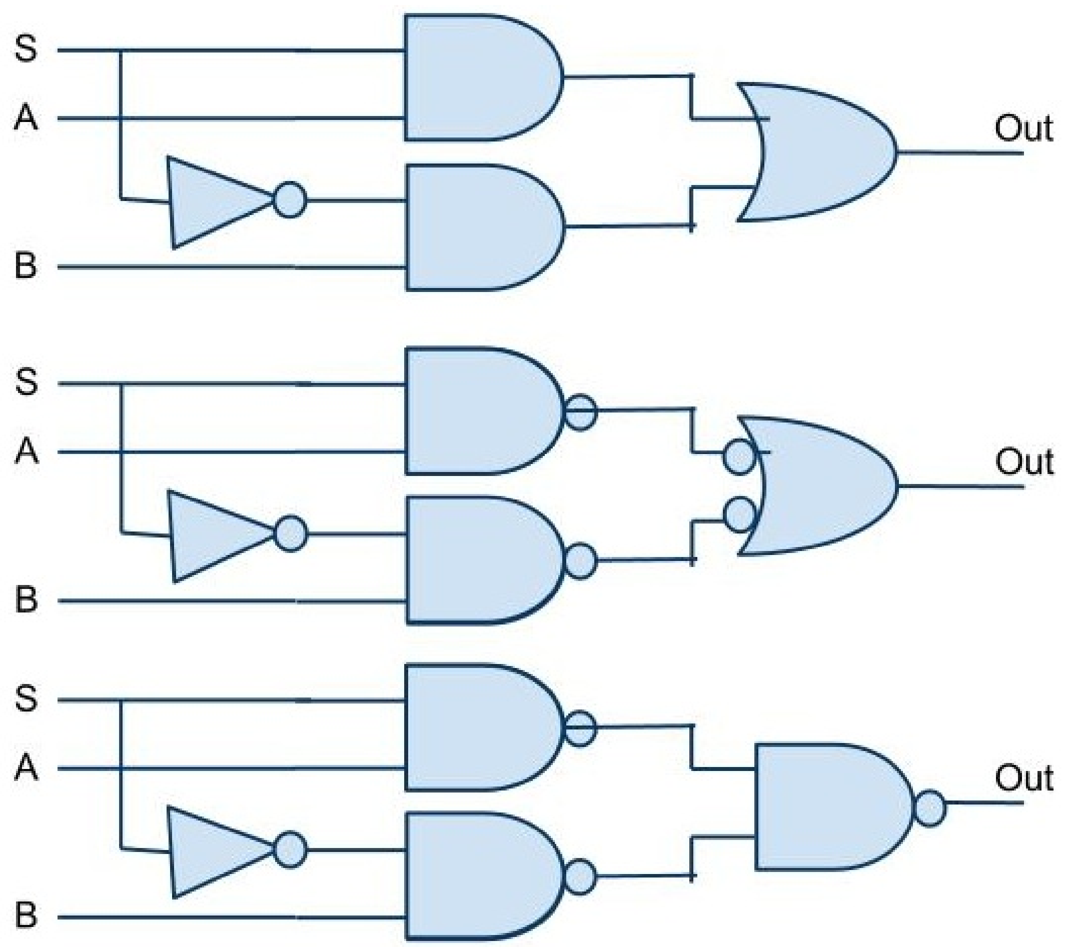


Figure D12. 2:1 MUX using NAND and INV.

Question D13): Form a 2 to 1 MUX using only NOR gates & inverters.

Answer D13):
Equation of a MUX is : Out = S * A + (S)bar * B
Using techniques similar to previous question, we can have following sequence of transformations to get 2:1 MUX using only NOR gates and INV.

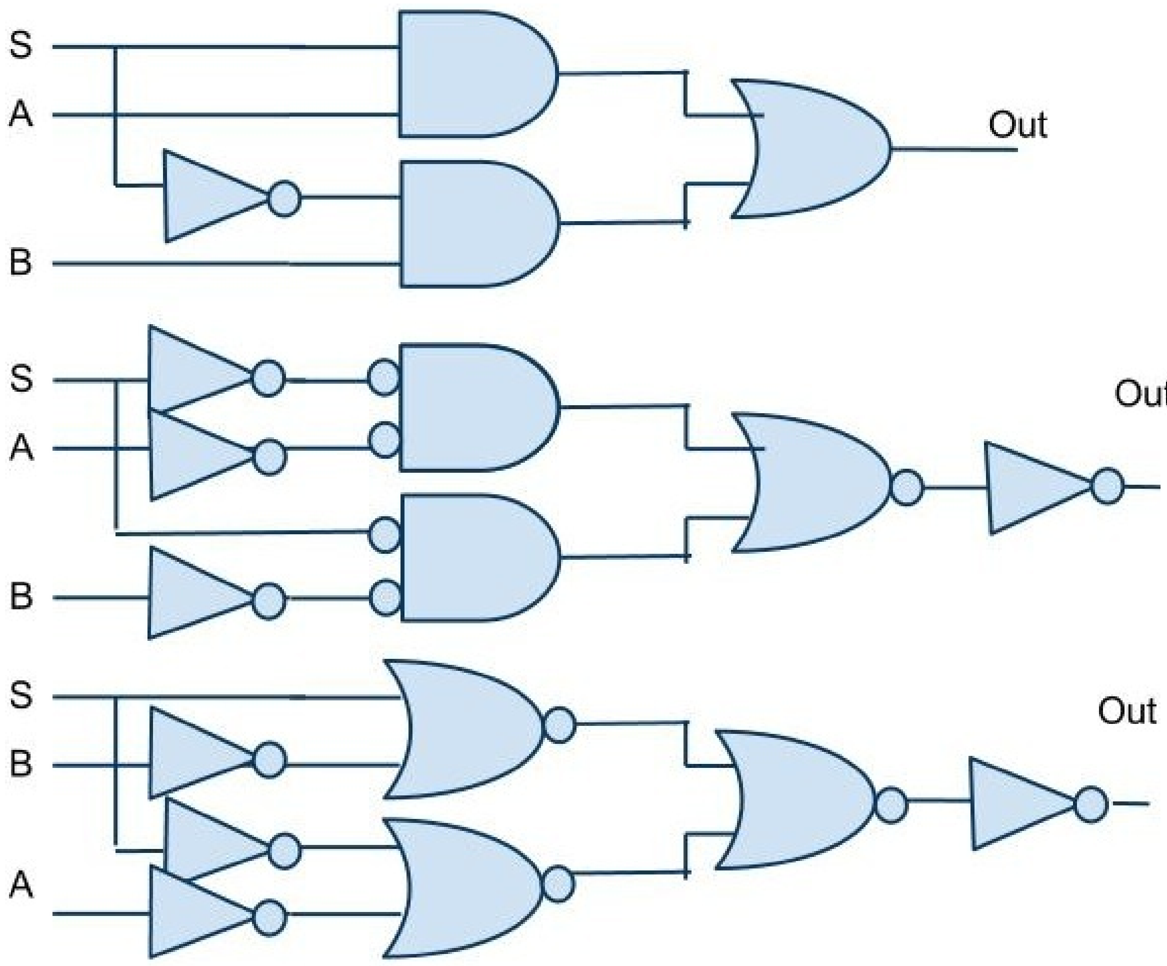
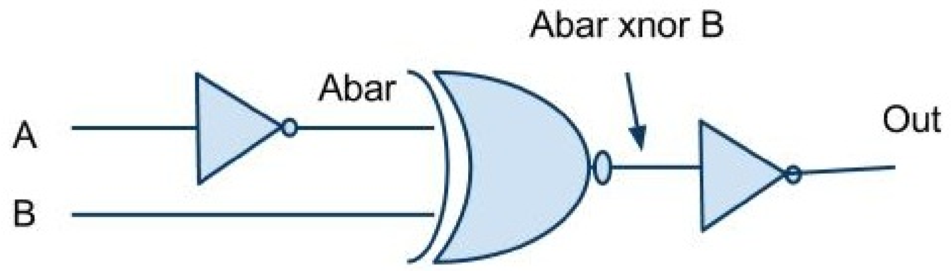


Figure D13. 2:1 MUX using NOR and INV.

Question D14): Can a 2 to 1 MUX be made using less than 3 NOR gates ?
Answer D14):
No

Question D15): Simplify logic in following figure :
Answer D15):

You can examine following circuit carefully and you can see that Out is same as A XOR B Or A XNOR B. Following is the simplification.



A	B	Abar	Abar xnor B	Out (Same as A xnor B)
0	0	1	0	1
0	1	1	1	0
1	0	0	1	0
1	1	0	0	1

Figure D15. Logic simplification for XNOR and INVs.

There is a corollary to this questions. If you inverter one of the inputs of an XOR gate resulting structure is XNOR gate and if you invert one of the gate of an XNOR gate, resulting structure is XOR gate, you can verify using truth tables.

Question D16): What could be the problems after simplifying logic in previous question ?

Answer D16):

There is a physical design and implementation aspect to the logic minimization or optimization. Optimized logic output has to be able to drive the load that unoptimized logic output was driving.

This is true in general with all simplification where we are reducing logic gate counts. If you're now driving a large load with just one XNOR gate than you could have a drive strength issue and you might have to increase the size of the XNOR gate so much that it defeats the purpose of optimization in first place.

Also if you have implemented XNOR gate using transmission gate/ pass gate devices you want to have input buffering provided by inverter at the input.

Question D17): Simplify equation Z = A + B((A * B + B) + A * (B)bar)

Answer D17):
You need to know the rules of digital design. The rules

Z = A + B(B * (A + 1) + A * (B)bar)

Z = A + B(B * 1 + A * (B)bar)

Z = A + B * B + B * A * (B)bar

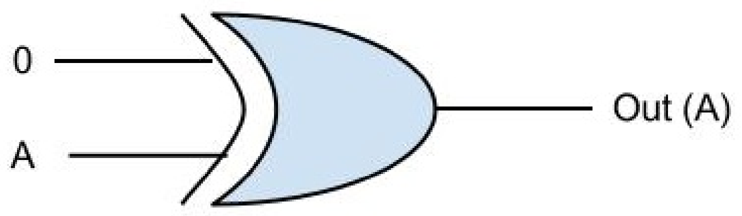
Z = A + B + A * 0

Z = A + B

Question D18) Build a buffer from a single XOR gate. Build an inverter from single XOR.
Answer D18):
For an XOR gate the boolean equation is following Assuming A,B inputs and O as output

O = A * (B)bar + (A)bar * B
If we make B = 0 in this equation we get

O = A * 1 + (A)bar * 0 = A
You can do similar exercise and find out that tying other input to '1' will give you inverter.



A	Out
0	0
1	1

Figure D18. BUF and INV using XOR.

Question D19) What is a multiplexer ?
Answer D19):
Multiplexer is a combinational circuit with multiple inputs and single output. It chooses one of the input signals based on the selector control input and directs it to the single output.

Question D20) Come up with 3 input NAND gate using minimum number of 2 input NAND gates.
Answer D20): Three input NAND gate equation can be written as following

O = (A * B * C)bar   We can rewrite this as following.

O = ((A * B) * C)bar

O = ((((A * B))bar)bar * C)bar
This means that we can use a two input NAND gate for input A & B and invert it’s output and feed output of inverter to one more 2 input NAND gate with C being another input. We also know that we can form inverter using 2 input NAND gate in two different ways. Following figure describes the circuit.

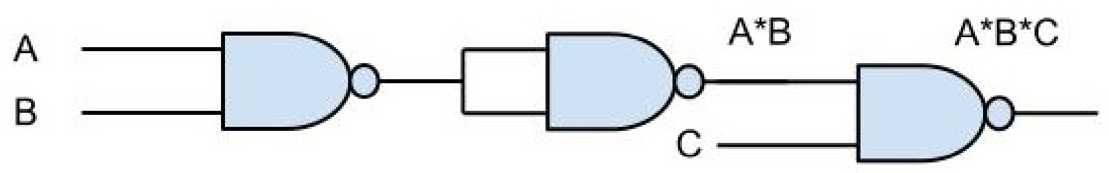


Figure D20. 3 Input NAND using 2 input NAND.

Question D21) Come up with an XOR gate using only 2 input mux and inverter.
Answer D21):
2 input XOR function is here :

O = A * (B)bar + (A)bar * B
This means when input A is true, we invert other input B and pass it on to output or when input A is not true (is ‘0’) we pass other input B as it is. We also know that MUX function is like following for input A, B and selector S.

O = A * (S)bar + B * S

If you think of A as the select for a 2 input MUX all we need to do is tie $\overline{B}$to true input of MUX and tie B to the other input of MUX. Following figure showing this clearly.

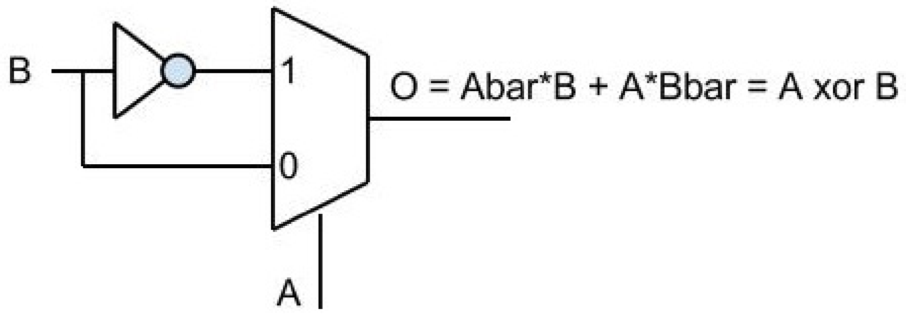


Figure D21. XOR using 2:1 MUX.

Question D22) Design a 4:1 MUX using only 2:1 MUXes ?
Answer D22)
Following is an option to do this, using 3 2:1 MUXes.

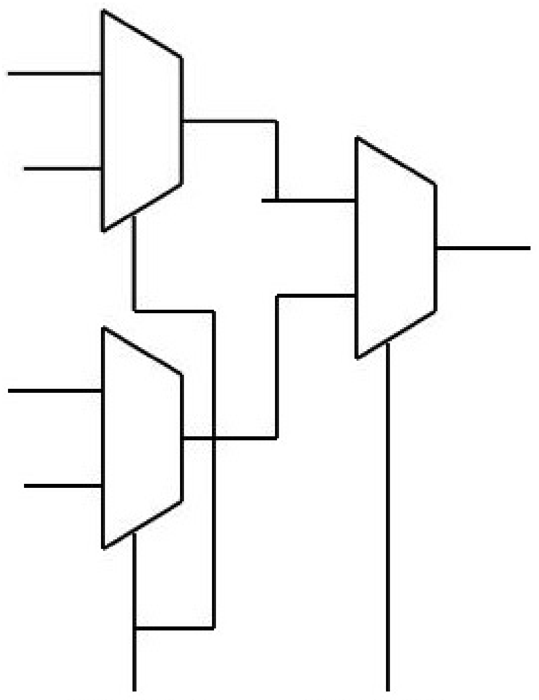


Figure D22. 4:1 MUX Using 2:1 MUXes

Question D23) : Come up with logic that counts number of ‘1’s in a 7 bit wide vector. You can only use combinational logic.

Answer D23):
Following is one of the ways to come up with such logic.
Input vector is 7 bit wide. To sum up 7 bits we need 3 bits of binary encoded output.
We’ve full adders available. A single full adder can add 3 input bits and generate 2 bits of binary encoded output.
E.g. a full adder can add 3 bit wide input vector ‘111’ and generate ‘11’ output.
We can pick two full adders and add up 6 bits of the input vector and will end up with two sets of two bit wide binary encoded data.
E.g. if input vector is ‘1100111’, we can assume two full adders adding up first 6 bits ‘110011’ where first three bits ‘110’ are input to first adder and ‘011’ are input to second adder. First adder will output ‘10’ (decimal 2) and second adder will also output ‘10’ (decimal 2), and we need to add up two two bit binary vectors. We can again

employ full adders to do this as we still have to account for the 7th input bit of the input vector. That can go into the least significant full adder carry-input.
For the above example :
Input vector ‘1100111’
input ‘110’ => full adder => ‘10’ output
input ‘011’ => full adder => ‘10’ output

```
     10
    +10
     ------
      100  => output (4)
```

Now accounting for the seventh input bit ‘1’ as carry into the least significant adder.

```
       1  <= Carry in.
    10
     +10
     -----
     101 => Binary encoded decimal 5 which is the input of 1s in
```

input vector ‘1100111’.

Full adders can be used to add-up 3 input bits at a time. Outputs of first level of full adders represent the two bit encoded version of the total ‘1’s count, which we need to add up get the final two digit encoded version of total ‘1’s. Since we need to add up 7 bit input vector, 7th input vector can be used as ‘Carry In’ in the second level of full adders.

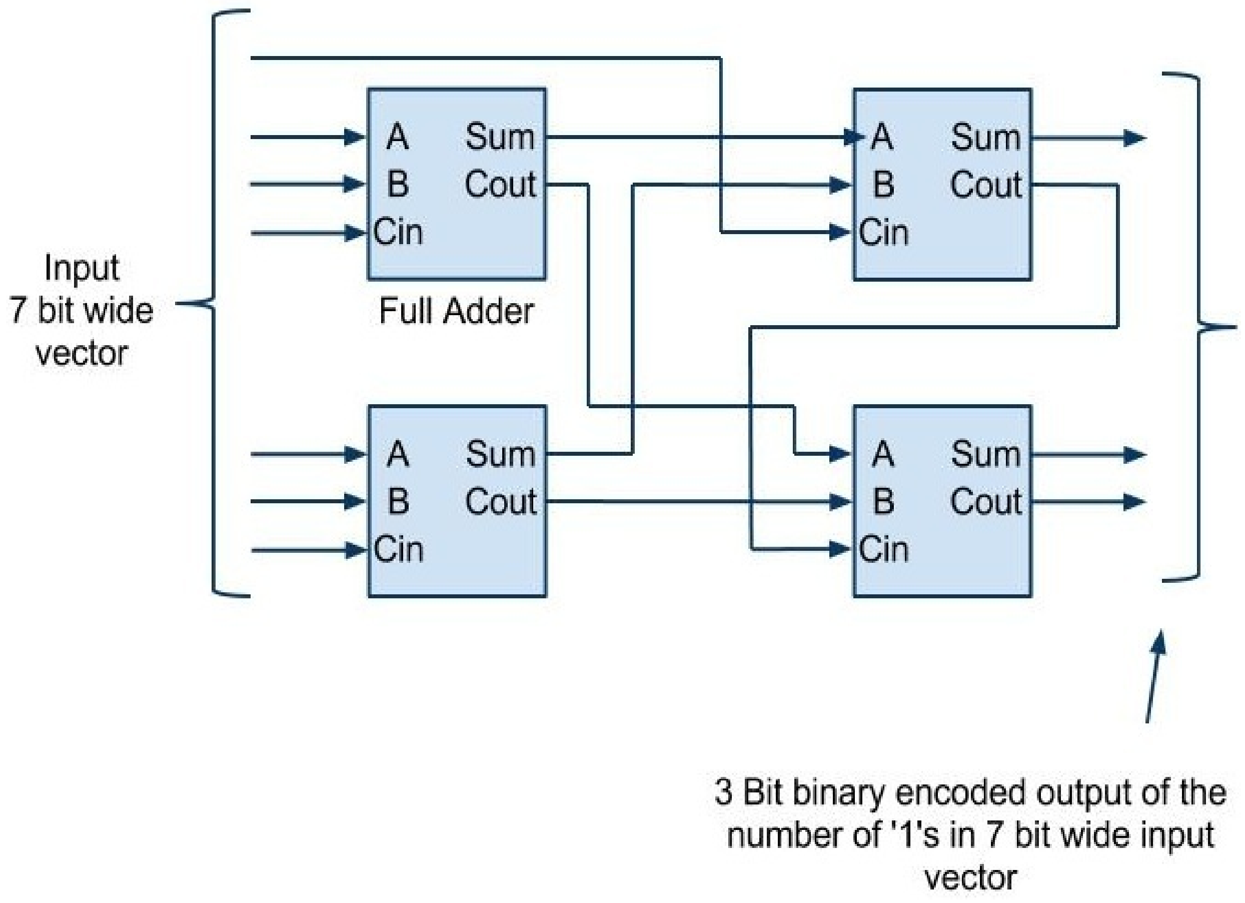

Input
7 bit wide
vector
A
B
Cin
Sum
Cout
Full Adder
3 Bit binary encoded output of the number of '1's in 7 bit wide input vector

Figure D23. Count number of ‘1’s

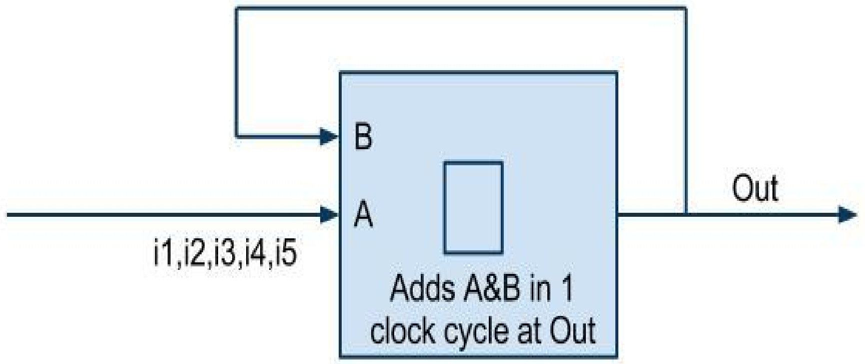
Question D24): We have an input vector where on subsequent clock cycles you get i1,i2,i3,i4 values. We want to output i1,i1+i2,i1+i2+i3 on subsequent clock edges. You’re given a black box which adds two input vectors and generates one output. Assume digital block output latency is one clock cycle.

Answer D24):

This question tests your knowledge of digital design with respect to pipelining. Best way to approach such questions is again, first draw the given input waveform and draw the required output waveform.

Then start with the given logic and come up with your first best guess of what logic should look like and start analyzing what resulting output waveform you get with your initial block of guessed logic. Keep iterating based on your comparison of what you initially got v/s what is needed.

Following logic will perform the required operation.



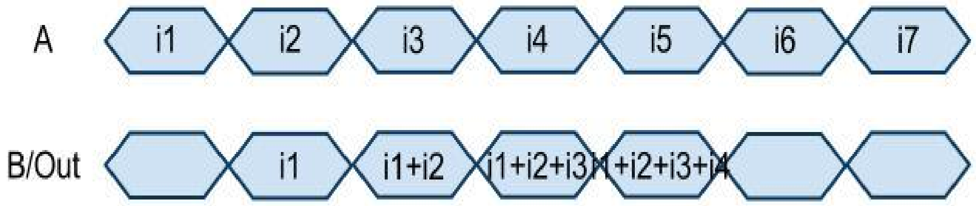


Figure D24. Adder state machine with specified delay.

Question D25) : How does this circuit change if adder latency changes to 2 cycles. ?

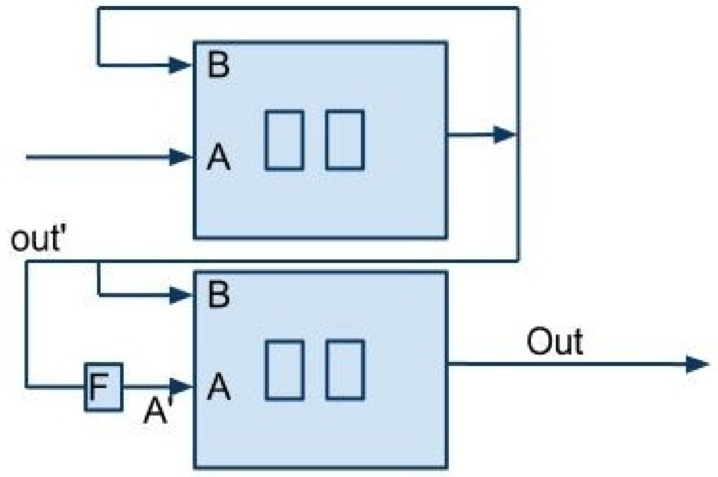
Answer D25):

Approach this question same as previous question. Remember you don’t necessarily have to get exact answer. What counts is how you approach the question.

Always keep thinking out loud. Continue spelling out the dialogue that's going on in your mind, don't keep your thoughts to yourself. Interviewer is looking at your approach, he wants to make sure you're trying your best to analytical approach such tough questions.

In this specific example, start with and initial simplistic guess of two clock cycle delay circuit (back to back flipflops) and draw the initial output waveform. Compare initial output waveform with the required output waveform and based on differences iterate more. Many times there isn't a systematic way to get your answer, you've to go trial and error.

Following logic will perform the required operation.



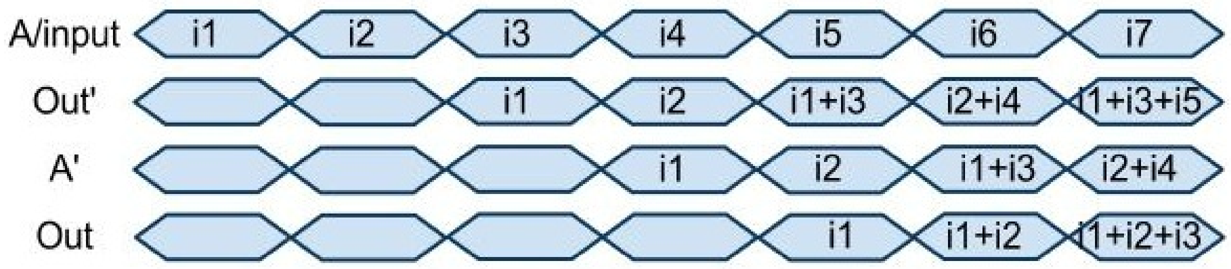


Figure D25. Adder state machine with more clock delay.

Question D26): Explain the working of a FIFO.
Answer D26):

FIFO is used for high throughput asynchronous data transfer. When you’re sending data from one domain to another domain and if high performance is required, you can not just get away with simple synchronizer(Metaflop).

As you can’t afford to loose clock cycles(In synchronizer you merely wait for additional clock cycles until you guarantee metastability free operation), you come up with storage element and reasonably complex handshaking scheme for control signals to facilitate the transfer.

An Asynchronous FIFO has two interfaces, one for writing the data into the FIFO and the other for reading the data out of FIFO. It has two clocks, one for writing and the other for reading.

Block A writes the data in the FIFO and Block B reads out the data from it. To facilitate error free operations, we have FIFO full and FIFO empty signals. These signals are generated with respect to the corresponding clock.

Keep in mind that, because control signals are generated in their corresponding domains and such domains are asynchronous to each other, these control signals have to be synchronized through the synchronizer !

FIFO full signal is used by block A (when FIFO is full, we don't want block A to write data into FIFO, this data will be lost), so it will be driven by the write clock. Similarly, FIFO empty will be driven by the read clock. Here read clock means block B clock and write clock means block A clock

Asynchronous FIFO is used at places when the performance matters more, when one does not want to waste clock cycles in handshake and more resources are available.

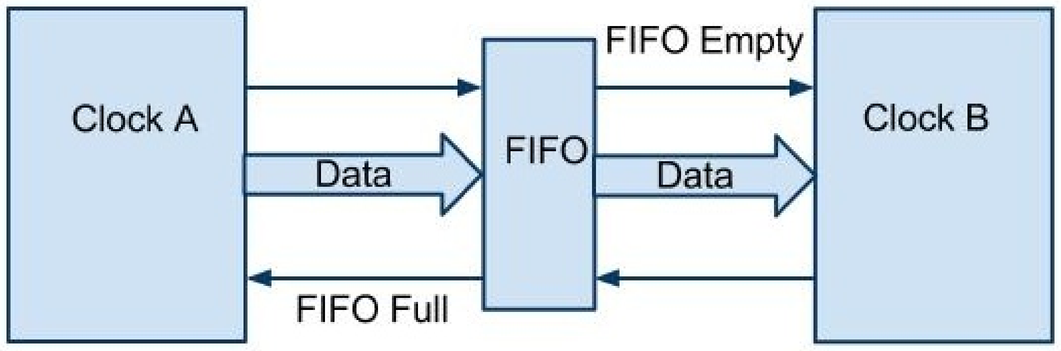


Figure D26. Asynchronous FIFO.

Question D27): How is FIFO depth/size determined ?
Answer D27):
Size of the FIFO depends on both read and write clock domain frequencies, their respective clock skew and write and read data rates. Data rate can vary depending on the two clock domain operation and requirement and frequency. FIFO has to be able to handle the case when data rate of writing operation is maximum and for read operation it's minimum.

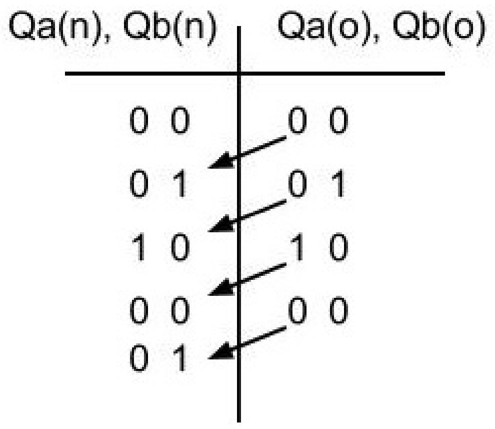
Question D28): Come up with a counter circuit which counts from 0 to 2 [0,1,2,0].
Answer D28):
This is a basic circuit which everyone should know. The purpose of this question is to build background for next question.

For counters you always begin by making truth table with two columns, current state and next state. In the figure (o) means old state and (n) stands for new state.
Now we can form boolean equations for Qa & Qb.

Qb(n) = (Qa(o))bar * (Qb(o))bar

Qa(n) = Qb(o) * (Qa(o))bar = Qb(o)
Old values go into the input of the flip flops and new values are output of flip flops. Given that we need 3 states, we need two flip flops. Figure 20 is the state transition table used to come with above mentioned equations. Based on the equation circuit shown in Figure 21 can be conceived. Waveforms are shown in Figure 22, it can be

seen that Qb or Qa waveform can be used as a divided by ‘3’ clock waveform, just that it’s not 50% duty cycle. It’s rather 33.33% duty cycle.



Qa(n), Qb(n)	Qa(o), Qb(o)
0 0	0 0
0 1	0 1
1 0	1 0
0 0	0 0
0 1	

Figure 20. State-transition truth table for 0-2 counter

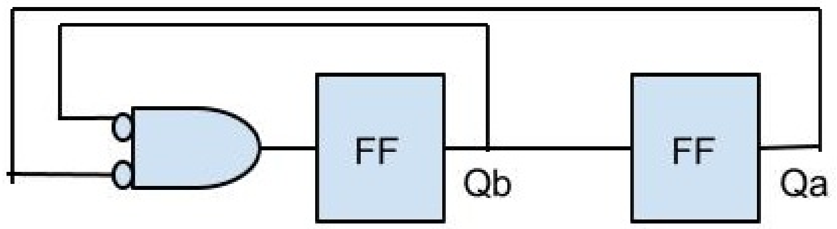


Figure D281. Circuit for 0-2 counter

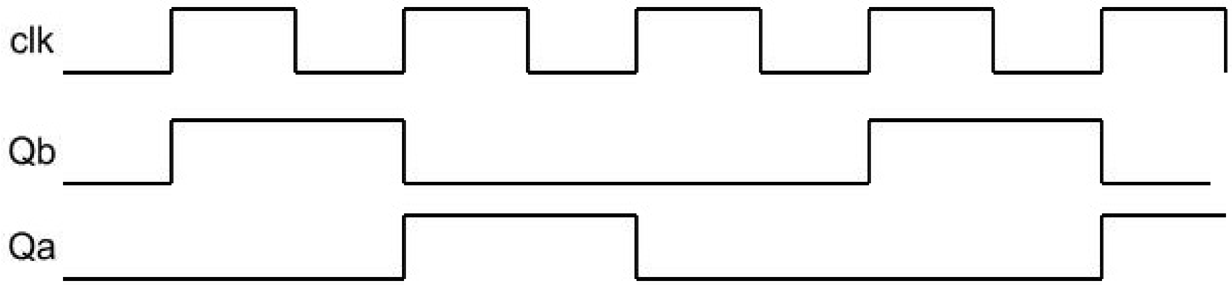


Figure D282. Clock and data waveforms for 0-2 counter

Question D29): Come up with a frequency divide by 3 clock circuit.
Answer D29):

Any good digital design interviewer, will very likely ask a clock divider circuit question. Clock divider by odd numbers, especially divide by ‘3’ are tricky circuits to come up with.

It is a question that is very likely to be asked and there in lies and opportunity for you to impress your interviewer. Spend enough time to familiarize yourself with clock divider circuits.

Remember that divide by circuits are some sort of variations of counters. Divide by ‘2’ can be thought of as variation of ‘2’ bit counter. Divide by ‘3’ as ‘3’ bit counter. This concept was dealt with in previous question. Counters are easy to understand and build. It is recommended first you read up basics of counter circuits and then try your couple of counter examples yourself to begin with.

There is a very good paper on clock divider, which describes a systematic way for coming up with divide by state machine for clocks. Paper can be found here : http://ebookbrowse.com/clock-dividers-made-easy-pdf-d75765174

If you look at the output of a ‘3’ bit counter, it’s easy to derive a divide by ‘3’ waveform which has a duty cycle of 33.33%. Again as described in previous question.

It takes a bit trick to get a 50% duty cycle divide by ‘3’ waveform. After experimenting a bit, came up with following circuit for a divide by ‘3’ clock divider.
You can see that in previous question what we need to do is delay Qa waveform by a phase and OR it with Qb and we’ll have a 50% duty cycle divide by ‘3’ clock waveform.

That’s exactly what we do in following circuit. We add a latch for delaying output by a phase, we use p-first latch to provide phase delay and we introduce explicit OR. Practically we can keep NOR in place of OR.

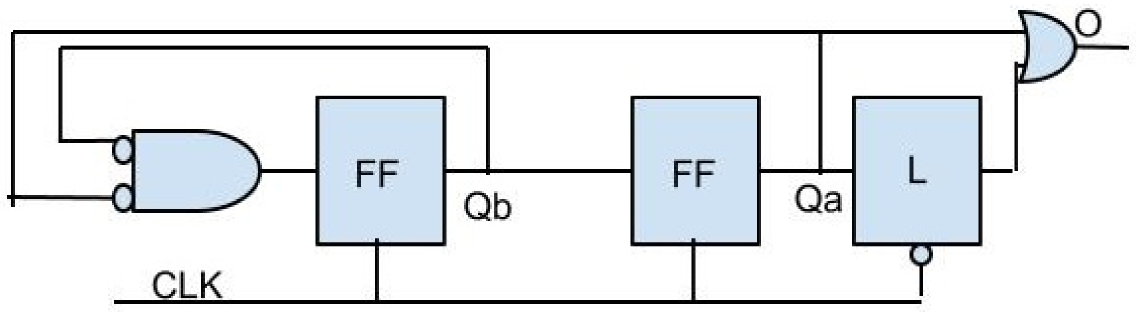


Figure D291. 50% duty cycle divide by ‘3’ clock generator

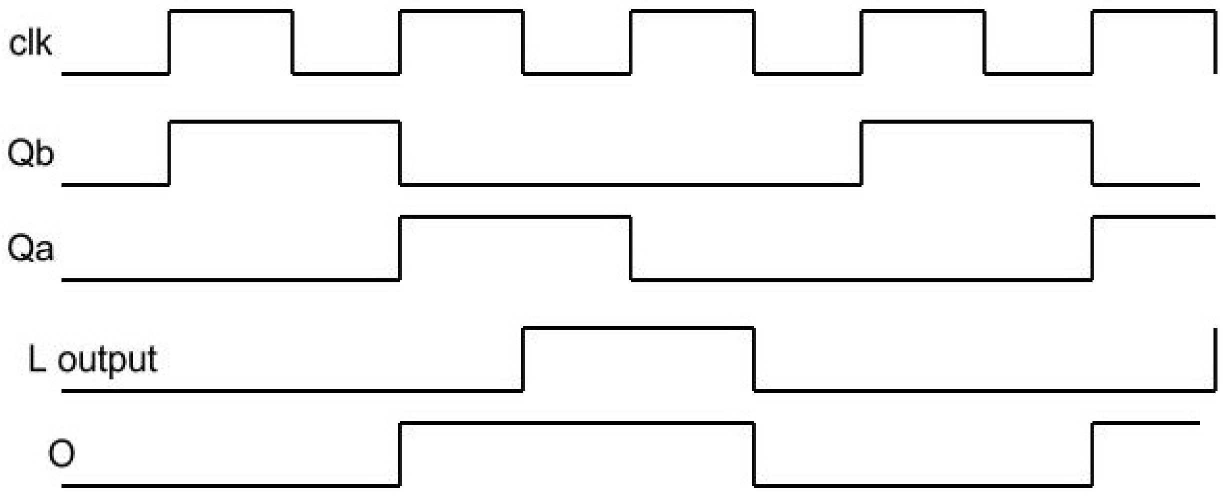


Figure D292. 50% duty cycle divide by ‘3’ clock generator waveforms

The paper mentioned earlier describes a way to get a divide by 3 circuit with duty cycle of 50%. First build a counter which counts from 0 to 2; then generate two enable signals, one active at time n=0, the other active at n=2; apply the two enable signals to two FF, the first one triggered on positive edge, the second one triggered on negative edge and then xor the FF outputs.

Question D30): Design an FSM detecting signal 1101
Answer D30):
For FSM, there no magic trick, you’ve to practice. Practice at least 10 different FSMs and you’ll get a hang of it.
Following is the FSM that detects sequence 1101. The numbers on transition lines show input/output. For example, if machines is in S0 state and 0 input arrives, it remains in S0 state and output 0, hence

0/0. S1 is the accepting state when arrived from S3, that's when output is asserted high.

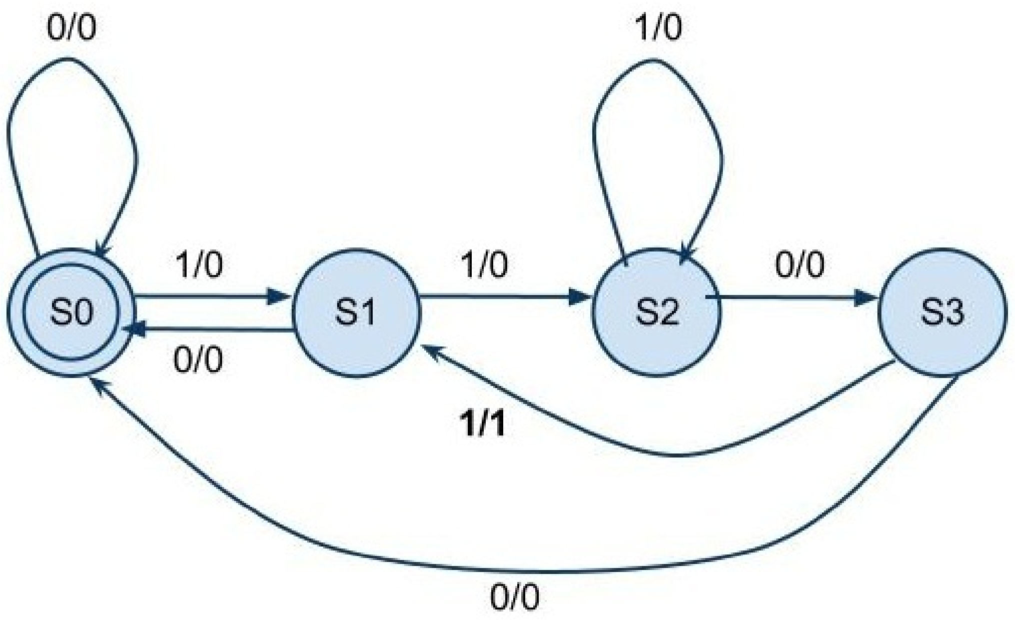


Figure D30. FSM detecting series 1101

Question D31): Given a DC signal how would you make a 50ps pulse? You have a 50ps inverter available.
Answer D31):
Use an XOR and inverter on one of the inputs.

Question D32): Why do we need TLB ?
Answer D32):
At least, brush up your basics of memory sub systems.
We need TLB to improve virtual address translation speed.

Question D33): What is a page table?
Answer D33):
A page table is the data structure used by a virtual memory system in a computer operating system to store the mapping between virtual addresses and physical addresses. Virtual addresses are those unique to the accessing process. Physical addresses are those unique to the hardware, i.e., RAM.

Question D34) What is a ring counter ?

Answer D34):
Ring counter is essentially a circular shift register. The output of the last shift register is fed to the input of the first register.

Question D35): Why are the drawbacks of associativity of Memory?
Answer D35):
With more associativity we need to do wider tag comparison, which means more logic and more power consumed. So with increasing associativity we increase hit rate, but the cost is more area and power, also there is a diminishing return after certain point

Question D36) What would you use synchronous reset or asynchronous reset ?
Answer D36):
Choice for type of reset is mostly a tradeoff. There is no clear winner. Both synchronous reset and asynchronous reset have advantages and disadvantages and based on their characteristics and the designers needs, one has to choose particular implementation.

Synchronous reset :

Advantages :

- This is the obvious advantage. synchronous reset conforms to synchronous design guidelines hence it ensures your design is 100% synchronous. This may not be a requirement for everyone, but many times it is a requirement that design be 100% synchronous. In such cases, it will be better to go with synchronous reset implementation.

- Protection against spurious glitches. Synchronous reset has to set up to the active clock edge in order to be effective. This provides for protection against accidental glitches as long these glitches don’t happen near the active clock edges. In that sense it is not 100% protection as random glitch could happen near the active clock edge and meet both setup and hold requirements and can cause flops to reset, when they are not expected to be reset.

This type of random glitches are more likely to happen if reset is generated by some internal conditions, which most of the time means reset travels through some combinational logic before it finally gets distributed throughout the system.

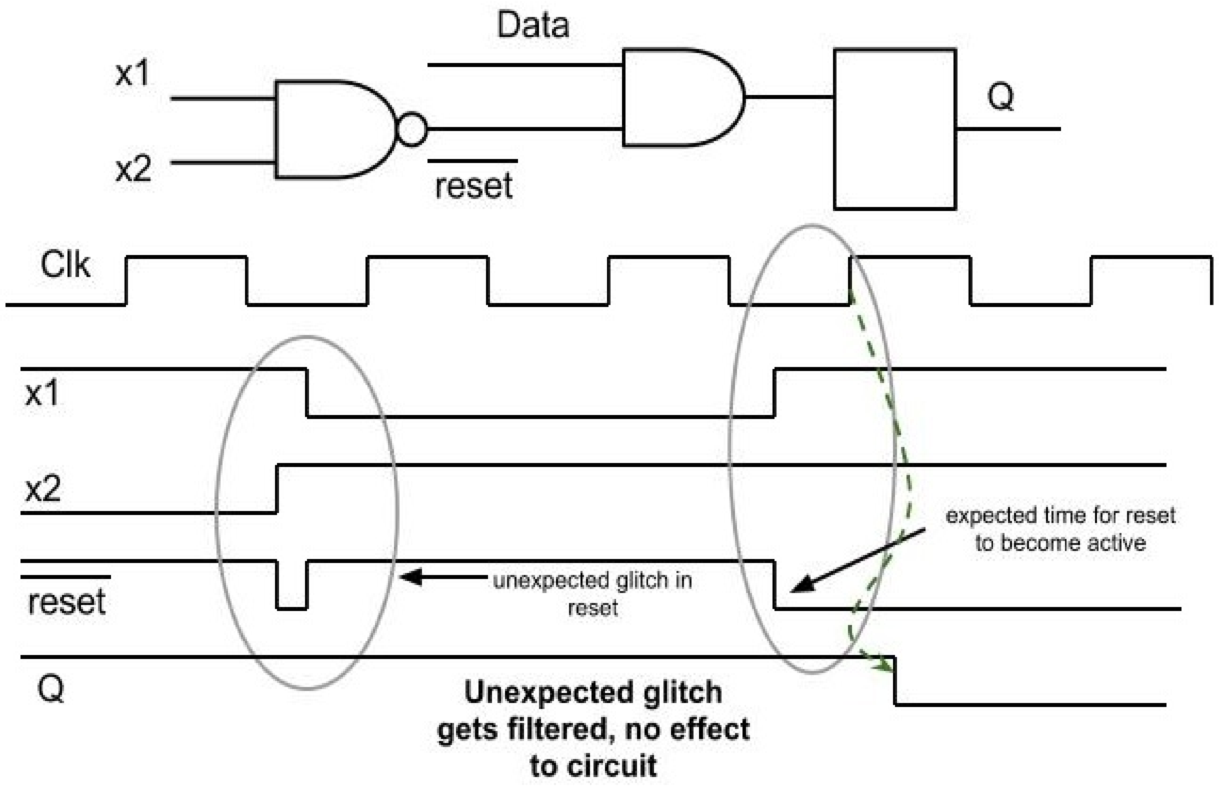

Data
x1
x2
reset
Q
Clk
x1
x2
reset
Q
unexpected glitch in reset
expected time for reset to become active
Unexpected glitch gets filtered, no effect to circuit

Figure D361: Glitch with synchronous resets

As shown in the figure, x1 and x2 generate (reset)bar. Because of the way x1 and x2 transition during the first clock cycle we get a glitch on reset signal, but because reset is synchronous and because glitch did not happen near the active clock edge, it got filtered and we only saw reset take effect later during the beginning of 4th clock cycle, where it was expected.

- One advantage that is touted for synchronous resets is smaller flops or the area savings. This is really not that much of an advantage. In terms of area savings it is really a wash between synchronous and asynchronous resets.

Synchronous reset flops are smaller as reset is just and-ed outside the flop with data, but you need that extra and gate per flop to accommodate reset. While asynchronous reset flop has to factor reset inside the flop design, where typically one of the last inverters in the feedback loop of the slave device is converted into NAND gate

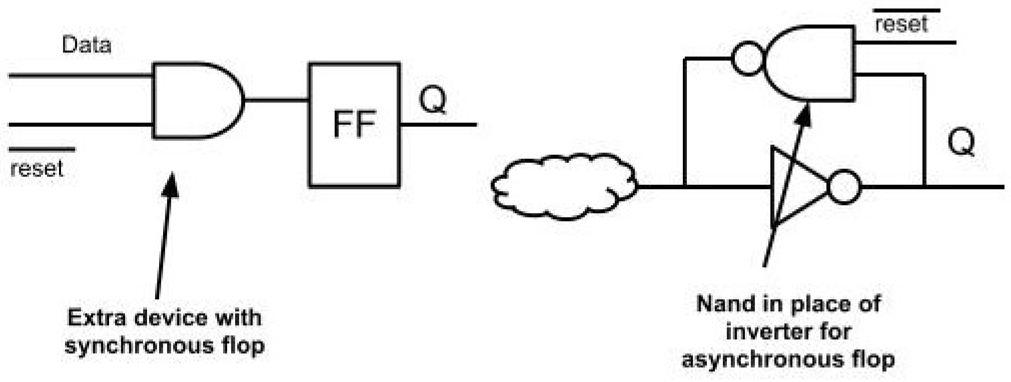

Data
reset
FF
Q
Extra device with synchronous flop
reset
Q
Nand in place of inverter for asynchronous flop

Figure D362: Synchronous vs Asynchronous reset flop comparison

Disadvantages :

- Wide enough pulse of the reset signal. We saw that being synchronous, reset has to meet the setup to the clock. We saw earlier in the figure that spurious glitches gets filtered in synchronous design, but this very behavior could be a problem. On the flip side when we do intend the reset to work, the reset pulse has to be wide enough such that it meets setup to the active edge of the clock.

- Another major issue with synchronous is clock gating. Designs are increasingly being clock gated to save power. Clock gating is the technique where clock is passed through an and gate with an enable signal, which can turn off the clock toggling when clock is not used thus saving power. This is in direct conflict with reset. When chip powers up, initially the clocks are not active and they could be gated by the clock enable, but right during the power up we need to force the chip into an known set and we need to use reset to achieve that. Synchronous reset will not take into effect unless there is active edge and if clock enable is off, there is no active edge of the clock.

Designer has to carefully account for this situation and design reset and clock enabling strategy which accounts for proper circuit operation.

- Use of tri-state structures. When tri-state devices are used, they need to be disabled at power-up. Because, when inadvertently enabled, tri-state device could crowbar and excessive current could flow through them and it could damage the chip. If tri-state enable is driven by a synchronous reset flop, the flop output could not be low, until the active edge of the clock arrives, and hence there is a potential to turn on tri-state device.

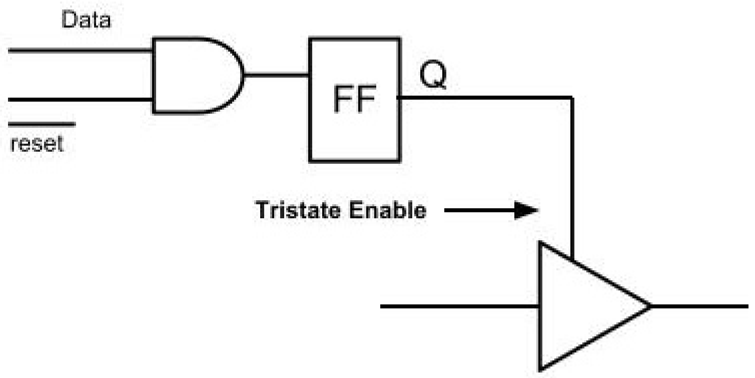


Figure D363 : Tri-state En. driven by synchronous reset flop.

Asynchronous reset :

Advantages :

- Faster data path. Asynchronous reset scheme removes that AND gate at the input of the flop, thus saving one stage delay along the data path. When you are pushing the timing limits of the chip. This is very helpful.

- It has obvious advantage of being able to reset flops without the need of a clock. Basically assertion of the reset doesn't have to setup to clock, it can come anytime and reset the flop. This could be double edged sword as we have seen earlier, but if your design permits the use of asynchronous reset, this could be an advantage.

Disadvantages :

- Biggest issue with asynchronous reset is reset de-assertion edge. Remember that when we refer to reset as 'asynchronous', we are referring to only the assertion of reset. You can see in figure about synchronous and asynchronous reset comparison, that one of the way asynchronous reset is implemented is through converting one the feedback loop inverters into NAND gate. You can see that when reset input of the NAND gate, goes low it forces the Q output to be low irrespective of the input of the feedback loop. But as soon as you deassert reset, that NAND gate immediately becomes an inverter and we are back to normal flop, which is susceptible to the setup and hold requirements. Hence deassertion of the reset could cause flop output to go metastable depending upon the relative timing between de-assertion and the clock edge. This is also called reset recovery time check. You don't have this problem in synchronous reset, as you are explicitly forced to check both setup and hold on reset as well as data, as both are AND-ed and fed to the flop.

- Spurious glitches. With asynchronous reset, unintended glitches will cause circuit to go into reset state. Usually a glitch filter has to be introduced right at the reset input port. Or one may have to switch to synchronous reset.

- If reset is internally generated and is not coming directly from the chip input port, it has to be excluded for DFT purposes. The reason is that, in order for the ATPG test vectors to work correctly, test program has to be able to control all flop inputs, including data, clock and all resets. During the test vector application, we can not have any flop get reset. If reset is coming externally, test program hold it at its inactive value. If master asynchronous reset is coming externally, test program also holds it at inactive state, but if asynchronous reset is generated internally, test program has no control on the final reset output and hence the asynchronous reset net has to be removed for DFT purpose.

One issue that is common to both type of reset is that reset release has to happen within one cycle. If reset release happen in different clock cycles, then different flops will come out of reset in different clock cycles and this will corrupt the state of your circuit. This could very well happen with large reset distribution trees, where by some of receivers are closer to the master distribution point and others could be farther away.

Thus reset tree distribution is non-trivial and almost as important as clock distribution. Although you don't have to meet skew requirements like clock, but the tree has to guarantee that all its branches are balanced such that the difference between time delay of any two branches is not more than a clock cycle, thus guaranteeing that reset removal will happen within one clock cycle and all flops in the design will come out of reset within one clock cycle, maintaining the coherent state of the design.

To address this problem with asynchronous reset, where it could be more severe, the master asynchronous reset coming off chip, is synchronized using a synchronizer, the synchronizer essentially converts asynchronous reset to be more like synchronous reset and it becomes the master distributor of the reset (head of reset tree). By clocking this synchronizer with the clock similar to the clock for the flops(last stage clock in clock distribution), we can minimize the risk of reset tree distribution not happening within one clock.

Question D37): How does asynchronous FIFO work ?
Answer D37):
When you want to transfer data from one clock domain to another clock domain, where both clocks are independent, there are two choices to accomplish this.

1) Use synchronizer
2) Use Asynchronous FIFO

The main concern for clock domain crossing is metastability. Please refer to the question about the metastability and synchronizer for getting more details. Basically lets assume we are sampling some data in xclk domain using xclk clock and if data is coming from ycllk domain where it was generated using yclk clock. Given that xclk and yclk are independent, we have no knowledge whether data coming yclk domain will setup correctly to xclk. In fact it is very likely that yclk data will violate the setup requirement for xclk flop and this will cause xclk flop to go metastable. To prevent this metastability we use metastability hardened flops or sometimes we simply refer to as synchronizer. It is nothing but a back to back series of flip flops where you essentially wait for additional clock cycles of the sampling domain to make sure metastability is resolved.

The key with synchronizer is that we wait for additional clock cycles to allow time for metastability resolution. This additional wait time might not be acceptable for some timing critical cross domain transactions. One can argue that, in that case we should not be having cross domain transactions in such cases, but today’s design reality is that, with SOCs cross clock domains are inevitable and you are very likely to have performance critical transactions that would happen across different clock domains.

There comes asynchronous FIFO in picture. It allows for a better cross clock domain transfer mechanism compared to synchronizer. Unlike synchronizer, where all data must wait for at least one additional local clock cycle, in async FIFO, we don’t wait for additional clock cycles, but a high speed memory FIFO, allows for on demand data data transfer between two clock domain. Whenever one of the domain is ready to send data to another domain it can do so without waiting as long as it knows other side has caught up. Similarly when one of the domain want to read data it can do without waiting as long as it knows other side has caught up.

Notice the qualification. FIFO doesn’t make everything completely seamless. One one domain is writing into memory how does it know whether memory is full or not ? It has to have indication of where the other domain is in terms of reading from the memory. And as we know it can not know what is going on in other domain without synchronizing the control signals coming from other domain with respect to its own clock. And synchronizing means waiting for additional clock cycle. But it is not as bad as synchronizer, infact as we will see later, when FIFO does become full or empty we end up conservatively writing into or reading from FIFO for additional clock cycles. Depending on few variables including the size of FIFO memory, this conservatism doesn’t come into play with every time a write or a read is performed.

Thus async FIFO does improve overall throughput of a clock domain crossing with the hardware overhead that comes in the form of memory and control logic for figuring out FIFO full and empty conditions. But depending upon the criticality of the signals involved in clock crossing it might be worth implementing extra hardware.

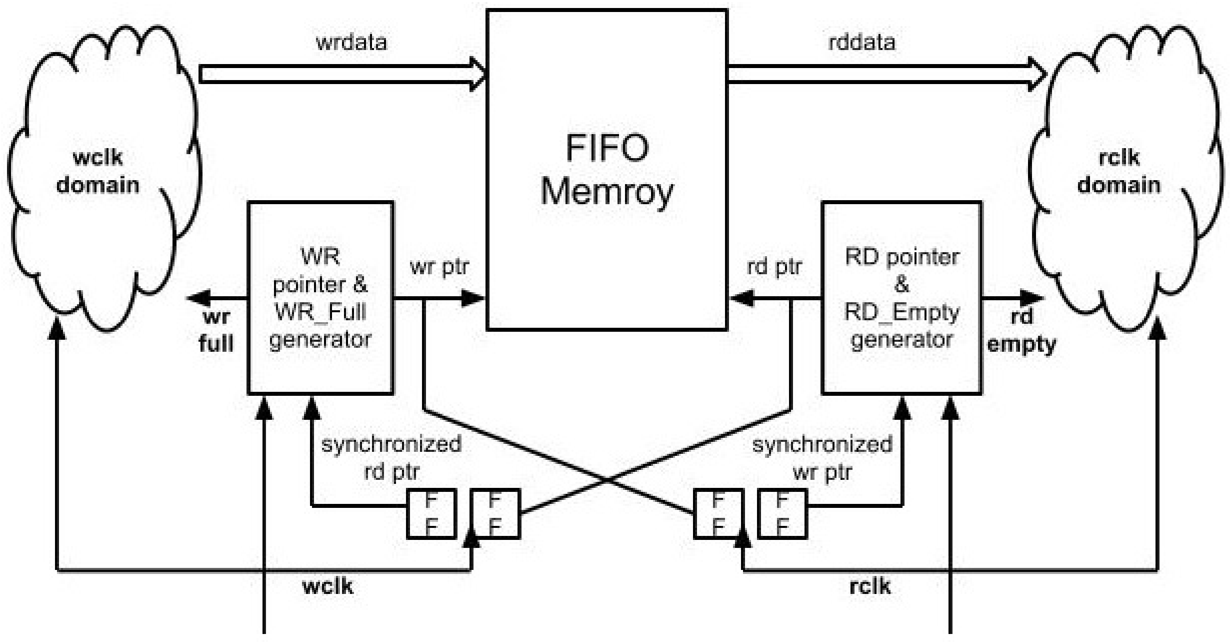


Figure D371: Asynchronous FIFO

Above shown figure shows the overall structure of async FIFO. You have a fast memory where actual data from one clock domain is written into and read out by other clock domain.

As shown here lets say our two clock domains are wclk and rclk. Lets assume our primary timing critical transaction is passing data from wclk domain to rclk domain. This FIFO is supporting the transfer of data from wclk domain to rclk. If data transfer in other direction that is, from rclk to wclk is critical we may have to implement similar FIFO that operate in other direction.

Going back to FIFO in the figure, the algorithm for writing into FIFO is very simple. When wclk domain has data ready to be send to rclk domain, it checks whether ‘wr full’ is asserted or not. The ‘wr full’ signal indicates FIFO is full, and if that is the case we have to wait until it clears up before we can write into the FIFO. If ‘wr full’ is clear, the data is written into FIFO at the location pointed by the write pointer ‘wr ptr’ and ‘wr ptr’ is incremented. As you can see ‘wr ptr’ is always pointing to the next location in FIFO which is ready to be written into.

Lets go through an example. Take a very simple case of 3 entry FIFO. Initially when design is reset, both ‘wr ptr’ and ‘rd ptr’ are pointing to zero and FIFO is empty.

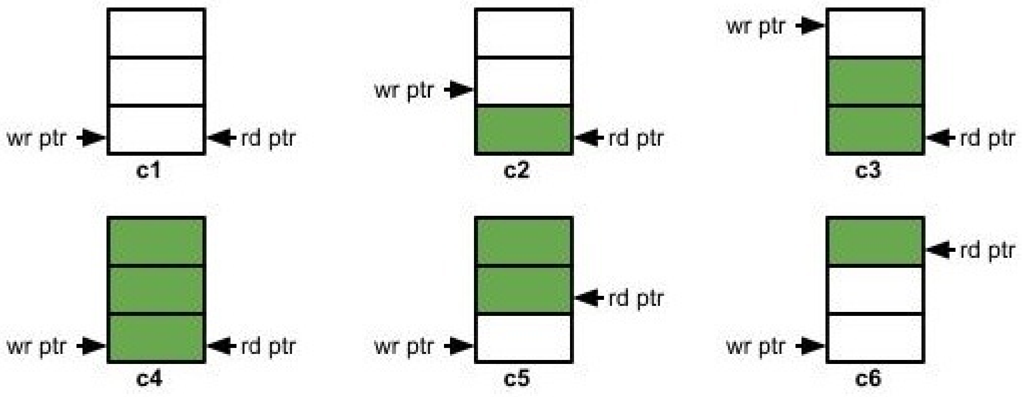


Figure D372: FIFO getting full.

We start at c1(clock cycle 1), whereby initially FIFO is empty and both read and write pointers are pointing to location zero. In subsequent clock cycles for wclk domain, for each cycle FIFO is written into and by clock cycle c4, FIFO becomes full,

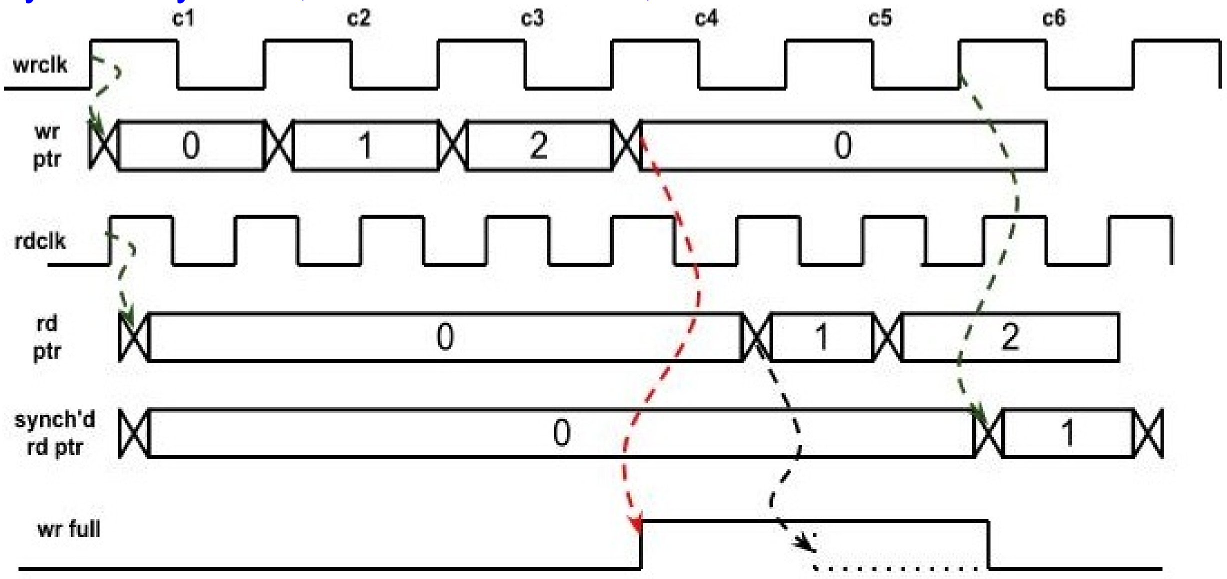


Figure D373: FIFO getting full waveforms.

As you can see in the waveforms, at the beginning of ‘wrclk’ cycle c4, ‘wr ptr’ goes to zero and now ‘wr ptr’ and ‘rd ptr’ are both equal indicating FIFO is full. You might wonder analyzing previous figure about FIFO getting full, that, initially when we start out and FIFO is empty, ‘wr ptr’ and ‘rd ptr’ are both equal and are equal to zero. So ‘wr ptr’ and ‘rd ptr’ are same when FIFO is empty as well as when FIFO is full. How do we find out whether FIFO is full or empty ?

To address this problem one of the approach is to add extra ‘MSB’ bit to both ‘wr ptr’ and ‘rd ptr’. In our case FIFO length is three entries, so normally FIFO address would be two bits wide, but with one extra MSB bit, we’ll have 3 bit wide FIFO pointers. When any of the pointer wraps around FIFO, we increment(toggle) MSB bit. So initially when FIFO is empty, ‘wr ptr’ would be ‘000’ and ‘rd ptr’ is also ‘000’, but when ‘wr ptr’ wraps around FIFO and FIFO becomes full in 4th wrclk cycle, ‘wr ptr’ address would be ‘100’ and ‘rd ptr’ would still be ‘000’. Our algorithm would like like this.

If (wr ptr[1:0] == rd ptr[1:0]) and (wr ptr[2] ~= rd ptr[2]) then FIFO is full and assert ‘fifo full’.

If (wr ptr[1:0] == rd ptr[1:0]) and (wr ptr[2] == rd ptr[2]) then FIFO is empty and assert ‘fifo empty’.

In other words when lower bits of ‘rd ptr’ and ‘wr ptr’ are equal, compare the MSB bit and if MSB bits of both ‘rd ptr’ and ‘wr ptr’ are equal , then FIFO is empty otherwise FIFO is full.

Going back to the waveforms, at the beginning of the ‘wrclk’ cycle 4, we find that FIFO is full and ‘fifo full’ signal is asserted. This would block further write attempt from the ‘wrclk’ domain as ‘rdclk’ hasn’t caught up and there is no point in writing to full FIFO.

As you can see when we compare the ‘wr ptr’ and ‘rd ptr’ in ‘wrclk’ domain, we are actually comparing synchronized version of ‘rd ptr’, which is compared to the original ‘rclk’, is 2 ‘wrclk’ cycles late, because we use two flops as part of synchronization circuit to synchronize ‘rdclk’ domain ‘rd ptr’ signals to ‘wrclk’. Because of this synchronization delay, we add pessimism in our circuit. This is how it happens. As you can see that after FIFO becomes full in ‘wrclk’ cycle 4, immediately on the next ‘rdclk’ cycle ‘rdclk’ domain performs a FIFO read, and we know that as soon as something is read off of a full FIFO, ideally we should turn off ‘fifo full’ signal to ‘wrclk’ domain right away, as shown with the dotted line on the ‘fifo full’ waveform, so that someone can write into FIFO in next cycle if needed. But because our communication from ‘rdclk’ domain to ‘wrclk’ domain is only through ‘wrclk’ based synchronizer, the ‘fifo full’ signal will be deasserted only 2 ‘wrclk’ cycles later. We do this, because we don’t have any better, trustable way to pass signals from one independent domain to another one. This is just pessimism, which is preferable compared to errors and functional failure.

Similarly we can analyze the case of FIFO getting empty.

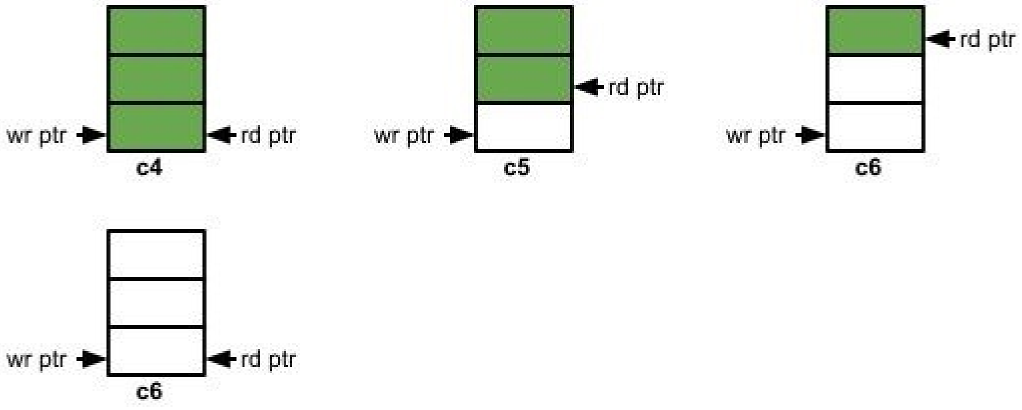


Figure D374: FIFO getting empty.

Lets say somewhere along the line FIFO got full and it happened in the ‘rdclk’ cycle 4. In subsequent clock cycles, at each clock cycle ‘rdclk’ domain read from the FIFO and given the FIFO depth is 3 entries, it got empty at ‘rdclk’ cycle 6. Following is how the waveforms for the same would look like.

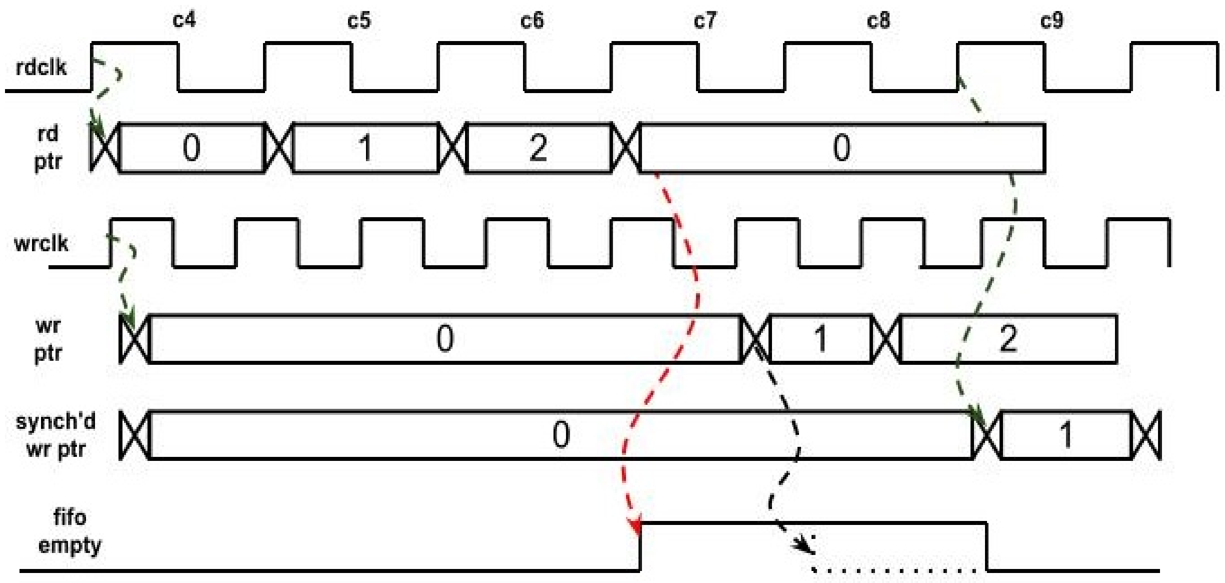


Figure D375: FIFO getting empty waveforms.

As you can see the waveforms look identical to the ‘fifo full’ case and there is again pessimism in the ‘fifo empty’ signal of two ‘rdclk’ cycles, whereby ‘rdclk’ domain is blocked for two extra cycles before it can start reading from the FIFO.

Question D38): Why are gray code used in FIFO pointers and not binary codes ?
Answer D38):
FIFO pointers don’t use binary codes, but they use gray codes. In gray codes, the hamming distance between successive codes is only one, in other words successive values only differ by 1 as can be seen in the following example.

One of the major purpose of the gray codes is to address the problem that arises when you are trying to sample a multiple bits code value and if more than one bits of the code could be changing for successive values of code, like binary code, you can get erroneous sampling if all bits of the code are not changing simultaneously and you happened to sample while bits are transitioning.

Dec	Gray	Binary
0	000	000
1	001	001
2	011	010
3	010	011
4	110	100
5	111	101
6	101	110
7	100	111

To clarify this point, lets take an example, where you are sampling 3 bit binary code(above example), basically a binary counter(address pointer) and in practical circuits based on several factors like, device delay, wire delay and clock skew, different bits will transition at different times. If the value is changing from decimal 3(binary ‘011’) to decimal 4(binary ‘100’) all three bits of the code change values and actual transition can happen like any of the following 6 combinations.

011 -> 111 -> 101 -> 100
011 -> 111 -> 110 -> 100
011 -> 001 -> 101 -> 100
011 -> 001 -> 000 -> 100
011 -> 010 -> 110 -> 100
011 -> 010 -> 000 -> 100

And depending upon what instance you sampled all three bits you can possibly detect ‘111’, ‘101’, ‘110’, ‘001’, ‘010’ or ‘000’, which are all wrong combinations. If you use gray codes, you avoid this issue, at the most you will be off by one number.

In FIFO, we are synchronizing read pointers in write clock domain and write pointers in read clock domain all the time, hence it is crucial we avoid such potential spurious switching issue by using gray codes.

Circuits Questions and Answers

Question C1): what is CMOS stand for ?
Answer C1) :
Complementary Metal Oxide Semiconductor.

Question C2): What does a delay of a cell/gate depend upon ?
Answer C2):
It mainly depends upon the input pin slope/slew rate and output load.

Question C3): Draw a CMOS transistor circuit for a 2 input NAND gate.
Answer C3):

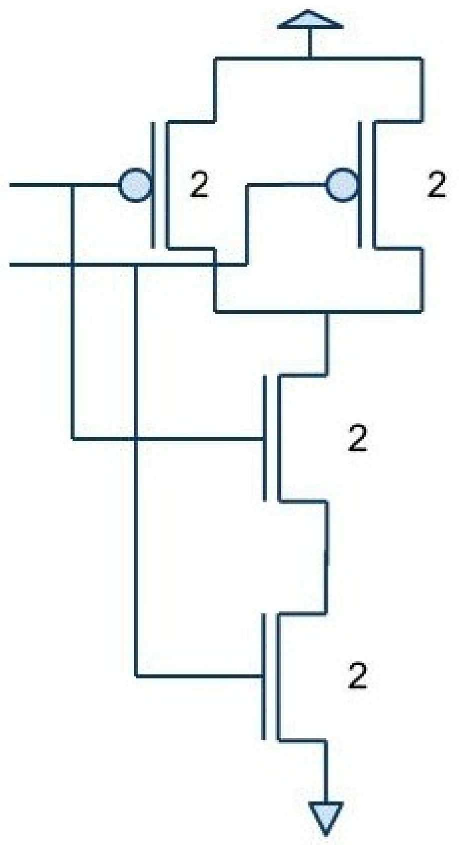


Figure C3. Two input NAND gate.

Question C4): Which input of 2 input NAND gate is faster. ? Why ?
Answer C4):
Input that is closes to the output node of NAND gate is faster. It's input A in above figure. This is because of we assume that the NMOS closest to the Vss is already turned on than Vss has effectively moved to the source of the NMOS near the output of the NAND gate (between A & B NMOS) and hence there is less resistive path now.

Question C5): When is the 2 input NAND gate slowest ? Why ?
Answer C5):
2 input NAND gate is slowest when both of the inputs A & B are switching high at the same time. This is when you can imagine the two NMOS in series combined form a bigger effective NMOS which is twice as resistive compared to single NMOS and as it's more resistive it charges the output slower and is slowest compared to just one of them turning on.

Question C6): Size transistors of a 2 input NAND gate for equal rise & fall delay assuming a P/N skew of 2/1

Answer C6):

In general you need to find out which transistors(s) are responsible for switching output. E.g in following NAND gate, only one of the PMOS would be responsible for output switching high as opposed to both NMOS needing to turn on for switching output low. So if PMOS size is 2, we're looking for NMOS equivalent size of 1 as P/N ratio is 2/1 in out example. As both NMOS have to turn on, the effective strength of NMOS is half(assuming both are equal in size). So get effective strength of 1, we need to have each individual NMOS sized 2. Think of two resistors in series. Same applies if there were 3 NMOS transistors in series. The equivalent strength would be ⅓ the single NMOS size.

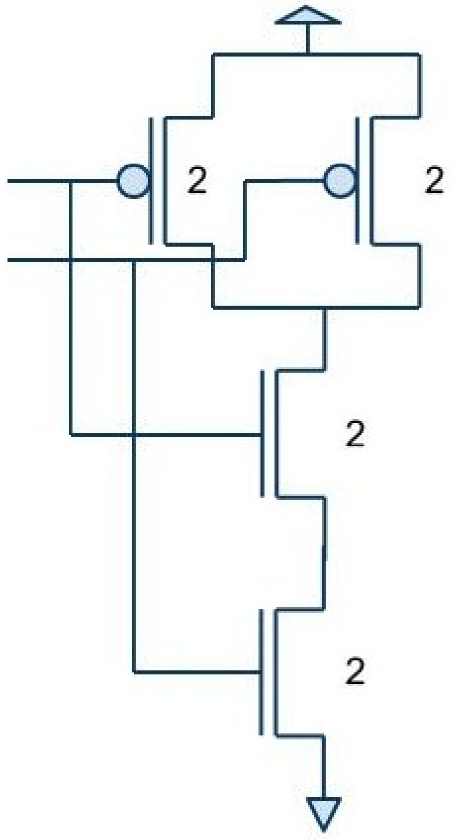


Figure C6. Two input NAND gate with P/N skew of 2/1.

Question C7): Draw the cross section of a MOSFET. What is saturation region ? How to bias device as such. Where is the bulk connection? What does it do?
Answer C7):

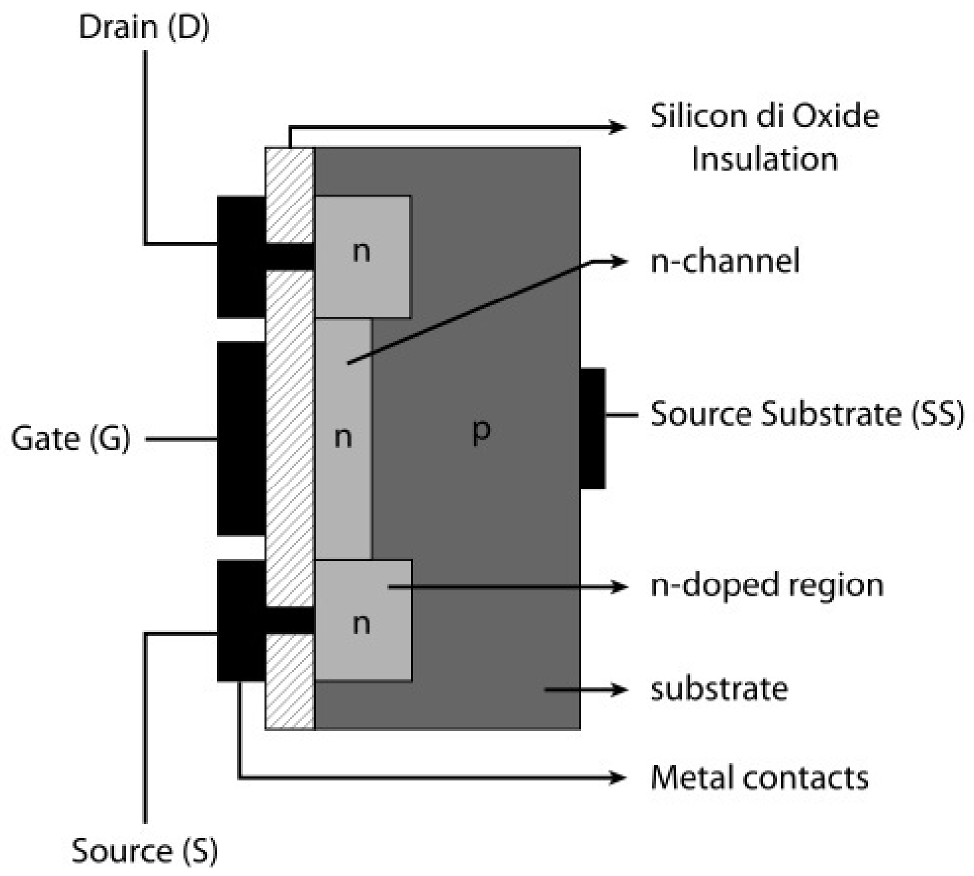


Figure C7. MOSFET cross section.

Image from wikipedia(http://en.wikipedia.org/wiki/File:N-channel_mosfet.svg)

In saturation, Vgs > Vt and Vds > Vgs - Vt. One way to bias a device into saturation is to tie the gate and drain together and then raise the gate voltage above Vt. That way, you're guaranteed to satisfy both conditions. The bulk connection is to the substrate. You can use a bulk connection to adjust the threshold voltage, via the body effect

Question C8): Which layers are best used for routing? Why?
Answer C8):

Higher layers are best used for routing. Because higher layers have less RC delay per micron of length. This is because higher layers have wider wider wires, which are more capacitive, but have much less resistance, resulting in overall less RC delay.

Question C9): How does a substrate bias (a.k.a. back-gate bias) on a MOS transistor affect Vt?
Answer C9):
A back-gate(body) bias increases the magnitude of Vt. The mechanism is an increase in the depletion width of the induced p-n junction under the gate. This uncovers more fixed charge in the channel region. (The mobile charge gets "pulled" to the substrate contact.)

Since the uncovered fixed charge has the same sign as the channel inversion charge, not as much channel inversion charge is needed to balance the charge on the gate. As a result, some of the inversion charge flows out the source terminal, so the channel isn't as inverted as it was prior to applying the substrate bias.

Therefore, the gate voltage needs to increase in magnitude to restore the previous level of channel inversion. For NMOS, with the decrease of Vb, Vt increases.

Question C10): Why power routes are routed in the top metal layers?
Answer C10):
This is not always true, but in general top metal layers are less resistive and hence IR drop is less in power distribution network. Also routing power wires in higher layers in top layers can free up lower metal layers to ease routing congestion in lower layers. Bigger the concern of IR drop more power routes are needed in all layers.

Question C11): What are the ways to speed up a standard cell ?
Answer C11):
Delay of the standard cell depends upon three main factors. Input slope, output load and drive strength. If you increase input slope by increasing the drive strength of the driver, the cell speeds up. If you reduce load at the output of the cell, cell speeds up. If you increase

the width of the cell, cell speeds up. A cell with lover sub-threshold voltage is faster than a nominal cell.

Question C12): How do you reduce noise or glitch ?
Answer C12):
First thing you ask when you’ve noise or glitch issue is whether you can tolerate the glitch or not. If you receiver is non-storage element, e.g. static gate like inverter or buffer, they attenuate the glitch/pulse(based on their skew) and noise glitch becomes tolerable by the time it reaches flop/latch.

Normally cross coupling from wires is biggest contributor of the noise glitch so increasing spacing to attacker helps. If the driver of the attacker is over-sized it helps to reduce the cross coupling, Also if the victim node driver is poorly sized, up-sizing the driver helps. Many times logical filtering, i.e. realizing the logical mutex conditions between attacker and victim helps filter out attackers.

Question C13): Explain short circuit current.
Answer C13):
In practice, because of finite input signal rise and fall times, there results a direct current path between the supply and ground. This exists for a short period of time during switching of the gate.

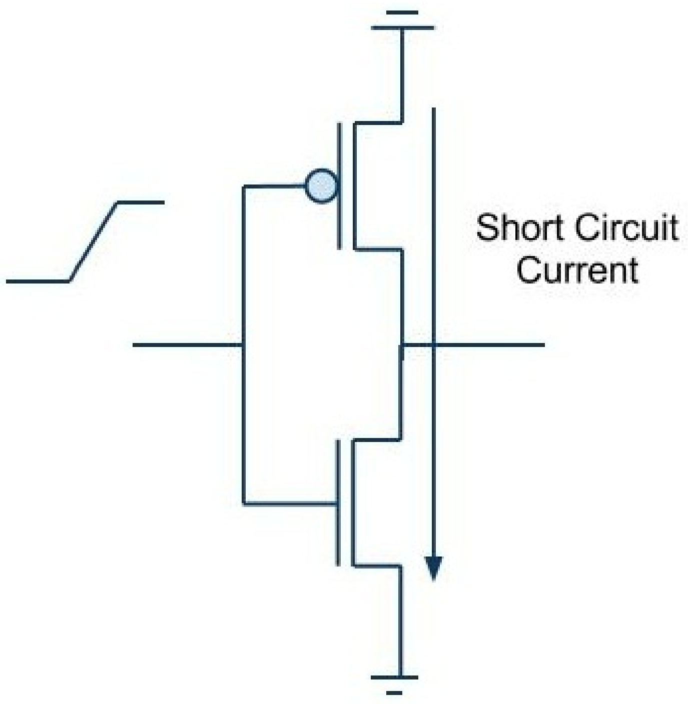


Figure C13. Short-circuit current.

Question C14): How do you size a ring oscillator ?
Answer C14):
In a simple ring oscillator design all inverters are sized same for achieve equal delay through them. Frequency of the ring oscillator is given by following equation which depends upon the delay through the inverters.

To speed up frequency we can increase the size of inverters to speed them up and to reduce the frequency we can down size the inverters to slow them down. Here f is the ring oscillator frequency, n is the number of inverters(stages) and td is the delay through inverter, which is assumed same for all.
$f = 1/(2 * n * td)$

Question C15): Why does digital gates run slower at higher temperature.
Answer C15):
As temperature increases the mobility of the carrier decreases, which causes active/switching current (Ion) to decrease, hence gates run slower at higher temperature.

Question C16): What happens to gate leakage at higher temperature ? Why ?
Answer C16):
Gate leakage is exponentially dependent upon temperature.

Question C17) How to reduce short channel effects ?
Answer C17):
Performance improvement requirement drive aggressive technology scaling, resulting in shortening channel length. Shrinking channel length causes loss of control-ability of gate over the channel depletion region, mainly because of the charge sharing from source or drain.

Short channel effect causes reliability issues, as channel length reduction means absence of pinchoff, shift in threshold voltage(Vt), drain induced barrier lowering and increased hot electron sensitivity.

Thinning gate oxide and using shallow source/drain junctions are effective mechanisms for preventing short channel effect. Thin-film SOI MOSFET are known to have less sensitivity to short channel effect. Recently tri-gate has emerged as a completely new way of fabricating transistors, which one can say addresses short channel length effect.

Question C18) What are DCAP cells ?
Answer C18)
Circuits are sensitive to noise glitches or spikes. Global distribution networks like power grid are major source of glitches. When a large number of locally clustered device switch simultaneously, a large amount of current is drawn from the power network.

This instantaneous large current causes voltage to droop for a small amount of time while devices are switches, this is called dynamic IR drop or Instantaneous Voltage Drop (IVD). It is very common in high speed memories, which have potentially thousands of cells switching at a time.

If capacitance is increased between VDD & VSS (rail to rail), VDD node becomes more resistant to the effects of IVD, as the capacitance acts as a charge reserve supplying local current sinks briefly for the short time when large number of devices are switching.

As such, DCAP cells which are nothing but capacitors are added to the areas of an IC that otherwise have no cells or the area where large simultaneous switching is expected(memory). However, DCAP cells normally come with a serious down-side. They are leaky devices and causes extra power dissipation, hence they need to be carefully used.

Question C19) Why is NAND preferred over NOR in CMOS design/fabrication ?
Answer C19)
NANDs are faster than NOR because NAND has NMOS devices in series where as NOR has PMOS devices in series. NMOS devices are faster because of higher mobility of electrons.

Side effect of this is that rising and falling delays of NOR are skewed because slower PMOS in series makes rising transition much slower and faster NMOS in parallel makes falling transition much faster.

Where NAND has balanced rise and fall delays. Because of stackability of NMOSes in NAND structure leakage can be controlled by adding extra device in pull down stack.

Question C20) What happens when PMOS and NMOS devices are switched in an inverter ?
Answer C20)
As you can see if we swap PMOS and NMOS devices in an inverter, input ‘1’ means NMOS on and hence output is pulled up, similarly input ‘0’ means PMOS on and output is pulled down, the circuit behaves as an buffer and not inverter anymore.

NMOS on top would not be able to pass complete VDD to output so output will not rise above VDD-Vth(Threshold Voltage) similarly PMOS would not pull down output all the way to VSS, but it would only pull it down to VSS+Vth.

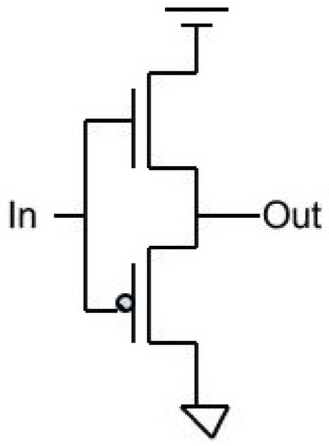


Figure C20. PMOS & NMOS swapped.

Question C21) What is charge sharing?
Answer C21)

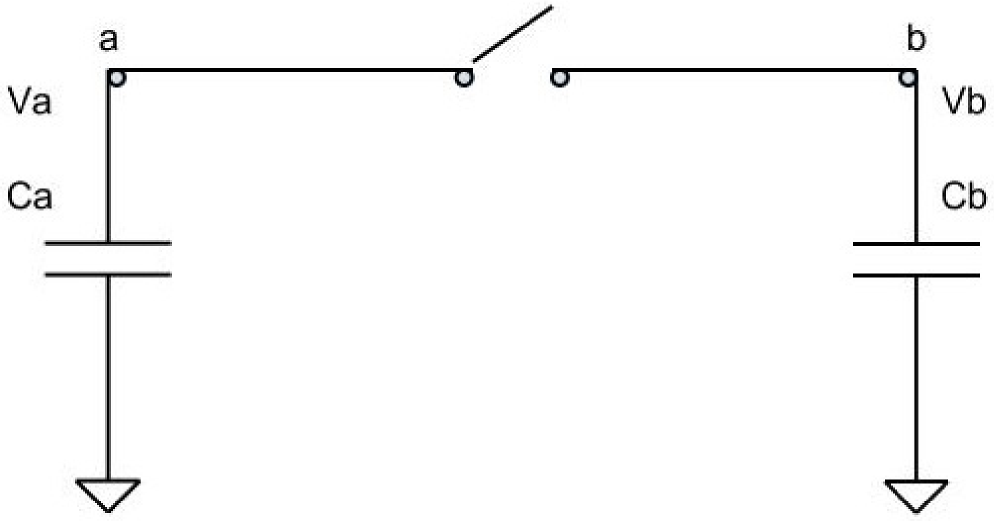


Figure C21. Charge sharing.

In the above mentioned picture, let’s say that initially Va=0, so charges stored on Ca capacitor is 0 as charge Q = CV. Let’s say that Vb > 0 and hence there is some charge stored on Cb capacitor, which will be Qb = CbVb. When switch is closed, nodes a & b are shorted and depending on the value of Ca compared to Cb, charge transfers from Cb to Ca. New capacitance is Ca+Cb, because Ca and Cb are in parallel now. Because total charge remains the same after switch is closed, we can say following :

CbVb = (Ca+Cb)Vb' [Previously node 'a' had zero charge, and node 'b' had CbVb charge]

Here Vb' is the voltage on node b after switch is closed and Vb is voltage on node 'b' before switch is closed.

If Ca =~ 0 or Cb >>> Ca, then Vb' = Vb, in all other cases Vb' << Vb.

This means unless Ca is very small compared to Cb node 'b' voltage will drop. This is the effect of charges sharing. When two capacitors are shorted depending on the capacitor values, charge is shared or transferred from one capacitor to another and voltage can droop on one of the node. Bigger the value of Ca, more charge will move(transfer) from Cb to Ca and when charges moves away from a node, the voltage at that node drops because voltage at a node is nothing but the charge(potential) difference between the node in question and charge at ground(which is zero). In other words a node can glitch up or down and if downstream there is a sequential element, it can capture glitch and cause false state to be captured and your circuit to malfunction.

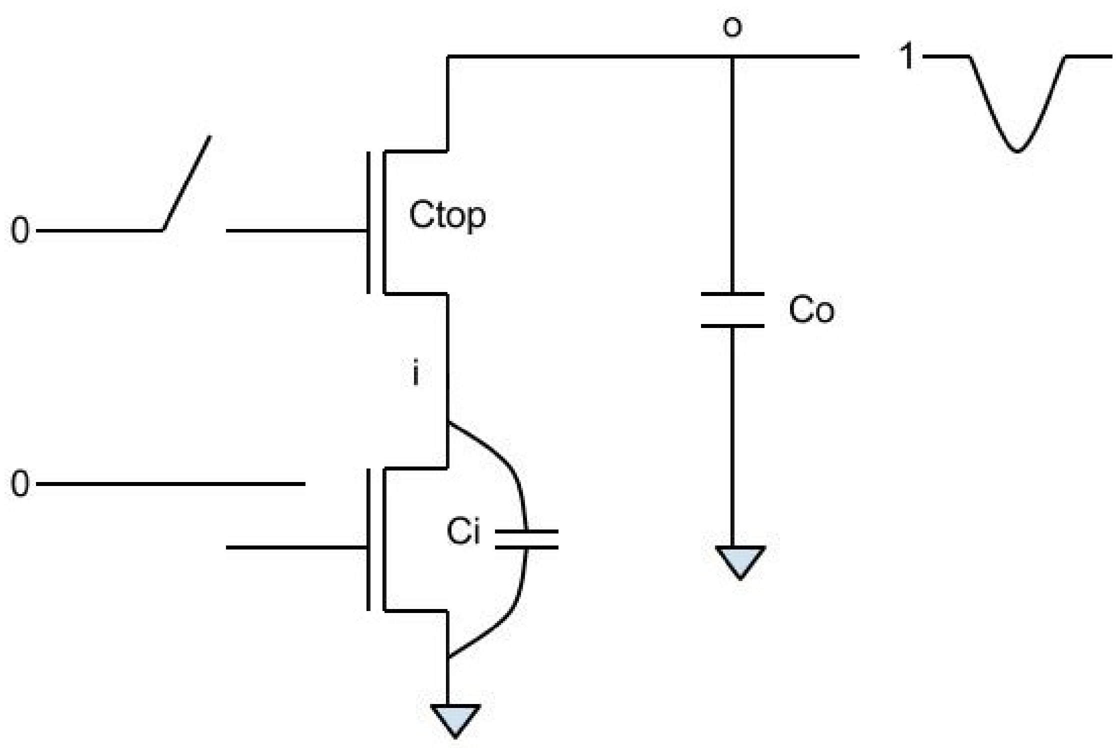


Figure C21. Charge sharing in stacked CMOS gates.

Question C22): What can you do to save power ?

Answer C22):

This is a generic question. In CMOS design following are the main components of power.

- Leakage Power Components.
    - Power caused by current flow that occurs regardless of voltage transitions, mostly from sub-threshold current of “off” transistors. Scales exponentially with voltage and temperature.
- Leakage Power reduction technique.
    - This is usually trade off with performance. You can use long channel devices to reduce leakage. High threshold voltage (Vt) transistors are less leaky. Both degrade performance of the device.
    - Downsize devices to reduce leakage. Hurts performance.
    - Process techniques/Dielectric properties help reduce leakage.

- Dynamic Power components
    - This is the the power dissipation caused by switching signals.
    - Power = C*V^2*f
    - More accurately Dynamic Power = Activity Factor * C * V^2 * f + Short Circuit Power + Glitch Power.
        - Sometimes a more effective measure of dynamic power is dynamic capacitance.
        - Dynamic Capacitance = ∑(Activity factor * C)/net
- Dynamic power reduction techniques (This is essentially minimize : Activity Factor * C * V^2 * f + Short Circuit Power + Glitch Power.)
    - Reduce Activity Factor: Mainly by Gating Clocks & Data. Don't let them toggle when they're not needed to be active. This is sometimes called toggle filtering as well.
    - Reduce Capacitance : Reduce capacitance on high activity nodes. Do topological optimization to minimize wire length. Increase spacing of the wires. In general you want to shoot for 50% wire capacitance and 50% device capacitance (gate + diffusion)
    - Short circuit and Glitch power are generally marginal in good design. You want to make sure slew rates are good and avoid other situation that cause short circuit current.
    - Downsize devices where you can afford to reduce device capacitance. Enable downsizing.
- Static Power components.
    - This is the power dissipation because of the bias current in IO or Analog circuits. It's independent of frequency and it's not the leakage power.
- Static Power reduction techniques.
    - Analog techniques to address bias current.

Static Timing Analysis Questions and Answers.

Question T1) What is setup time ?
Answer T1)
For any sequential element e.g. latch or flip-flop, data needs to be stable when clock-capture edge is active. Actually data needs to be stable for a certain time before clock-capture edge activates, because if data is changing near the clock-capture edge, sequential element (latch or flip-flop) can get into a metastable state and can capture unintended value at the output.

The time requirement that data be stage for before the clock capture edge activates is called the setup time of that sequential element.

Question T2) What is hold time ?
Answer T2)
For any sequential element e.g. latch or flip-flop, data needs to be held stable when clock-capture edge is active. Actually data needs to be held stable for a certain time after clock-capture edge deactivates, because if data is changing near the clock-capture edge, sequential element can get into a metastable state and can capture unintended value at the output.

This time requirement that data needs to be held stable for after the clock capture-edge deactivates is called hold time requirement for that sequqntial.

Question T3): What does the setup time of a flop depend upon ?
Answer T3):

Setup time of a flip-flop depends upon the Input data slope, Clock slope and Output load.

Question T4): What does the hold time of a flip-flop depend upon ?
Answer T4):
Hold time of a flip-flop depends upon the Input data slope, Clock slope and Output load.

Question T5): Describe a timing path.
Answer T5):
For standard cell based designs, following figure illustrates basic timing path. Timing path typically starts at one of the sequential (storage element) which could be either a flip-flop or a latch.

The timing path starts at the clock pin of the flip-flop/latch. Active clock edge on this element triggers the data at the output of such element to change. This is the first stage delay which is also called clock -> data out(Q) delay.

Then data goes through stages of combinational delay and interconnect wires. Each of such stage has it's own timing delay that accumulates along the path. Eventually the data arrives at the sampling storage element, which is again a flip-flop or a latch.

That's where data has to meet setup and hold checks against the clock of the receiving flip-flop/latch. Also notice for the timing paths in the same clock domain, generating flip-flop clock and sampling flip-flop clocks are derived from a single source, which is called the point of divergence.

In reality, actual start point for a synchronous clock based circuits is the first instance where clocks branch off to generating path and sampling path as shown here in the picture, which is also called point of divergence.

To simplify analysis we agree that clock will arrive at very much a fixed time at the clock pin of all sequentials in the design. This simplified the analysis of the timing path. from one sequential to another sequential.

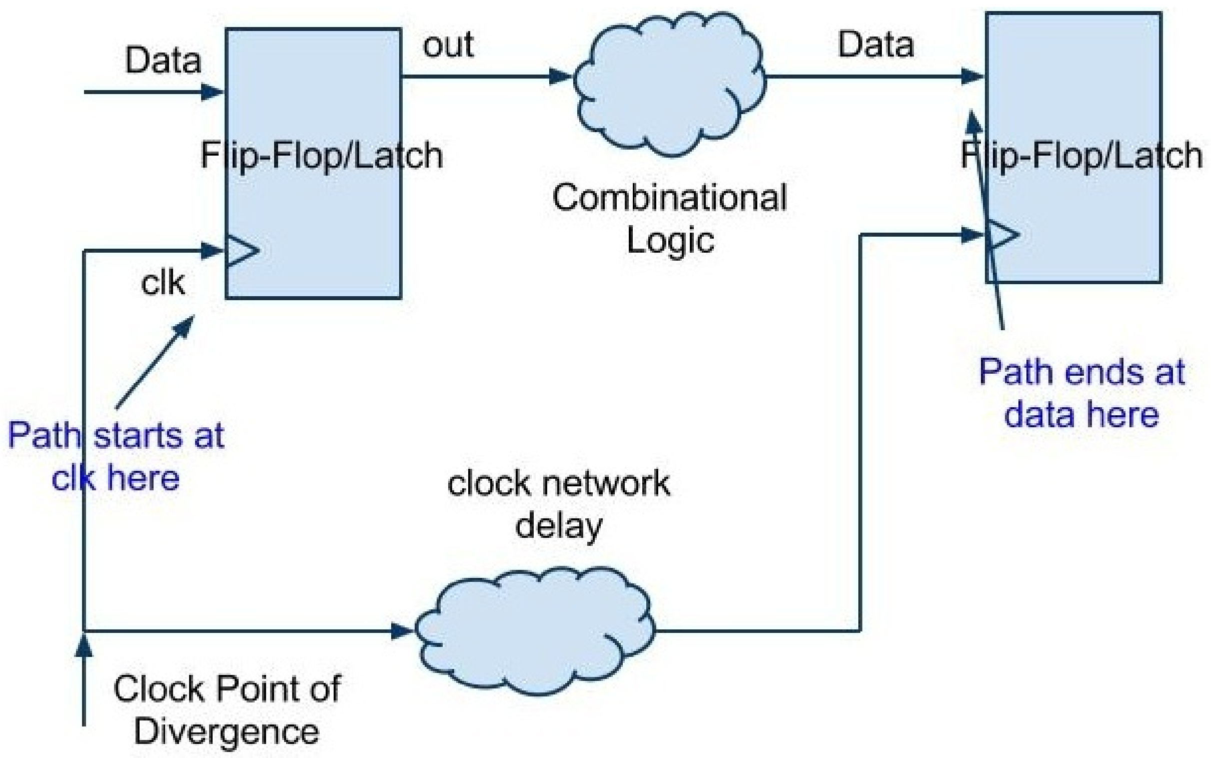


Figure T15. Timing path from one Flipflop to another Flipflop.

Question T6): How do you fix timing path from latch to latch ?
Answer T6):
Latch to latch setup time violation is fixed just like flop to flop path setup time violation, where you either speed up the data from latch to latch by either optimizing logic count, or speeding up the gate delays and/or speeding up wire delays.

You can speed up gates by upsizing them and you can speed up wires by either promoting them to higher layers, or widening their width or increasing spacing or shielding them.

You can also fix the timing issues by delaying the sampling clock or speeding up the generating clock. Latch to latch hold violations have inherent protection of a phase or half a clock cycle.

Question T7): If hold violation exists in design, is it OK to sign off design? If not, why?
Answer T7):
No, hold violations are functional failures. Unlike setup violations, which go away with reduced frequency, hold violations are frequency

independent and are functional failures as mentioned earlier.

Question T8): Explain CTS (Clock Tree Synthesis) flow.
Answer T8):
The goal of the CTS flow is to minimize the clock skew and the clock insertion delay. This is the flow where actual clock distribution tree is synthesized. Before CTS timing tools use ideal clock arrival times. After CTS real clock distribution tree is available so real clock arrival times are used.

Question T9): What is metastability and what are its effects ?
Answer T9):
Whenever there is setup or hold time violations in a flip-flop, it enters a state where its output is unpredictable. This state of unpredictable output is known as metastable state.

It's also called quasi stable state. At the end of metastable state, the flip-flop settles down to either '1' or '0'. The whole process is known as metastability.

Question T10) What is the difference between a latch and a flip-flop.
Answer T10):
Latch is level sensitive device, while flip-flop is edge sensitive. Actually a D flip-flop is made from two back to back latches, in master-slave configuration.

A low level master latch is followed by a high level slave latch to form a rising edge sensitive D flip-flop. Latch is made using fewer devices hence lower power compared to flip-flop, but flip-flip is immune to glitches while latch will pass through glitches.

Question T11) What is clock skew ?
Answer T11):
In synchronous circuit design, usually a gridded clock is used. Gridded clock means, at least for parts of the design clock has to arrive at the same time. In reality clock arrives at different times at different clock receivers in the design. This phenomenon of clock arriving at different times at different places is called 'clock skew'.

This could happen because of several reasons, device delay variation because of threshold voltage and channel length variation, on chip device variation, differing interconnect/wire delays, interconnect delay variation, temperature variation, capacitive coupling, varying receiver load, bad clock distribution tree design.

Skew could help or hurt in your design. If in reality clock arrives later than expected at a sampling element, and if there is minimal data delay from previous sampling element, new data can race through from the previous sampling element and can get inadvertently captured at the sampling element where clock arrives late.

Or if there is enough of data delay from previous sampling element to the current element, the late arriving clock compared to data can help meet setup requirement at the sampling element. Following figure describes the false data capture becuase of clock being late.

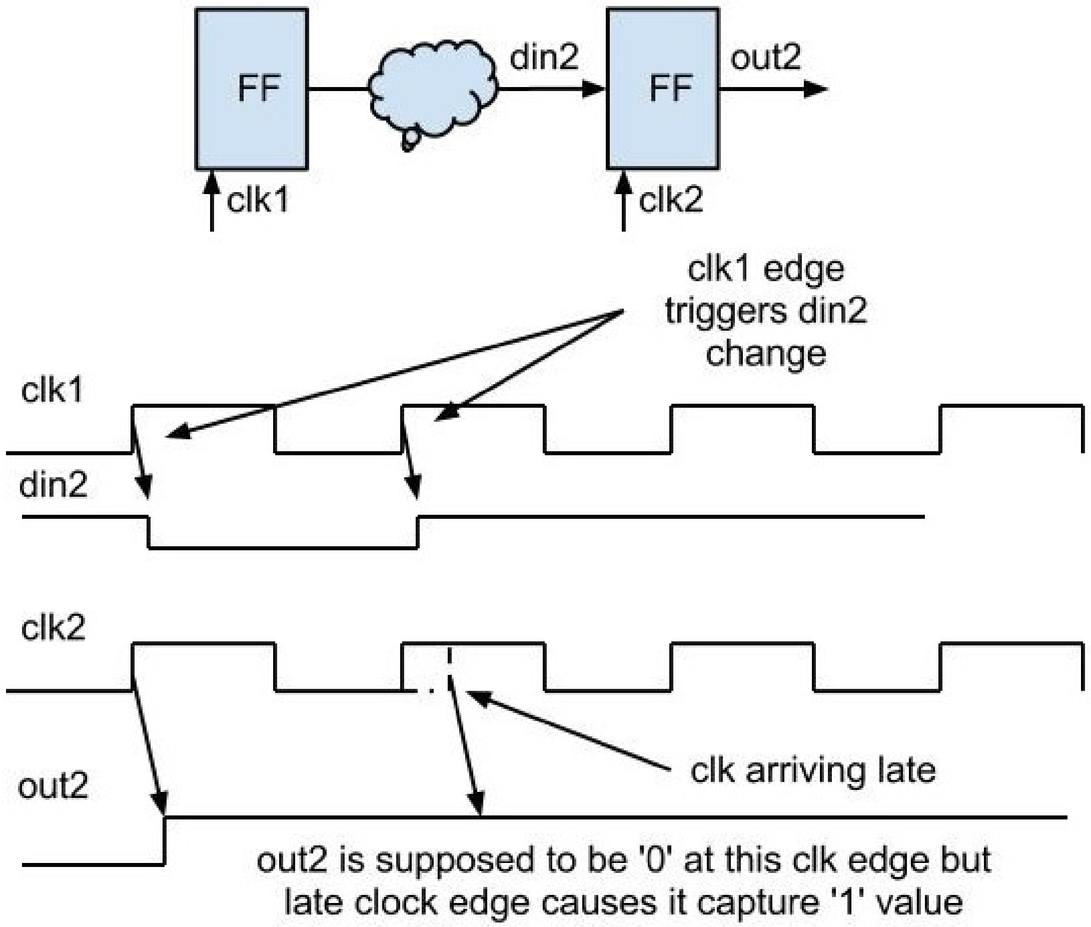

FF
din2
FF
out2
clk1
clk2
clk1 edge triggers din2 change
clk1
din2
clk2
clk arriving late
out2
out2 is supposed to be '0' at this clk edge but late clock edge causes it capture '1' value

Figure T11. False data capture because of late clock (clock skew)

Question T12) What happens to delay if you increase load capacitance?
Answer T12):
Usually device slows down if load capacitance is increased. Device delay depends on three parameters, 1) the strength of device, which usually is the width of the device 2) input slope or slew rate and 3) output load capacitance. Increasing strength or the width increases self load as well.

Question T13) What is clock-gating ?
Answer T13)
Clock gating is a power saving technique. In synchronous circuits a logic gate (AND) is added to the clock net, where other input of the AND gate can be used to turn off clock to certain receiving sequentials which are not active, thus saving power because of toggling clock.

Question T14) How to avoid metastability ?
Answer T14)
If we ensure that input data meets setup and hold requirements, we can guarantee that we avoid metastability. Sometimes it's not possible to guarantee to meet setup/hold requirements, especially generating signal is coming from a different clock domain compared to sampling clock.

In such cases, what we do is place back to back flip-flops and allocate extra timing cycles of clocks to sample the data. Such a series of back to back flops is called a metastability hardened flop.

Essentially what we're doing is that we allow first flip-flop to potentially go metastable, during first sampling clock cycle and we give first flop a full sampling clock cycle to recover from metastability. If within first cycle first flop recovers to correct value, we capture correct value at output second flip-flop at beginning of second clock cycle. If first flop recovers to wrong stage we've to wait for one more cycle i.e. beginning of 3rd cycle of sampling clock to capture the correct value.

Sometimes it’s possible that first flop takes longer than one sampling clock cycle to recover to stable value, in which case 3 flip-flops in series can be used. More flops in series reduces the failure in capturing the correct value at output at expense of more number of cycles.

Question T15) Explain signal timing propagation from one flip-flop to another flip-flop through combinational delay.
Answer T15)
Following is a simple structure where output of a flop goes through some stages of combinational logic, represented by pink bubble and is eventually samples by receiving flop. Receiving flop, which samples the FF2_in data, poses timing requirements on the input data signal.

The logic between FF1_out to FF2_in should be such that signal transitions could propagate through this logic fast enough to be captured by the receiving flop. For a flop to correctly capture input data, the input data to flop has to arrive and become stable for some period of time before the capture clock edge at the flop.

This requirement is called the setup time of the flop. Usually you'll run into setup time issues when there is too much logic in between two flop or the combinational delay is too small. Hence this is sometimes called max delay or slow delay timing issue and the constraints is called max delay constraint.

In figure there is max delay constraint on FF2_in input at receiving flop. Now you can realize that max delay or slow delay constraint is frequency dependent. If you are failing setup to a flop and if you slow down the clock frequency, your clock cycle time increases, hence you've larger time for your slow signal transitions to propagate through and you'll now meet setup requirements.

Typically your digital circuit is run at certain frequency which sets your max delay constraints. Amount of time the signal falls short to meet the setup time is called setup or max, slack or margin.

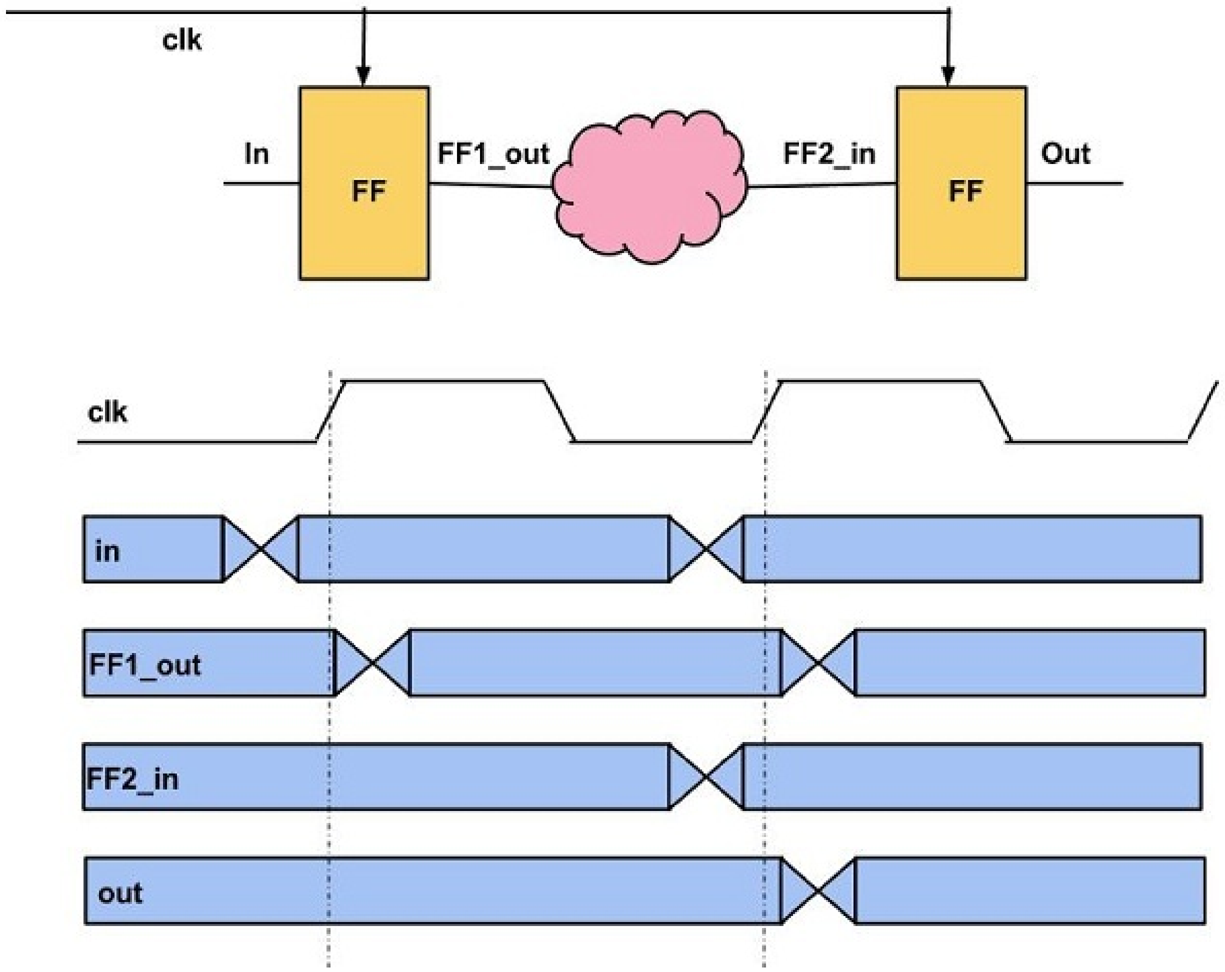


Figure T15. Signal timing propagation from flip-flop to flip-flop

Question T16) Explain setup failure to a flip-flop.

Answer T16)

Following figure describes visually a setup failure. As you can see that first flop releases the data at the active edge of clock, which happens to be the rising edge of the clock. FF1_out falls sometime after the clk1 rises.

The delay from the clock rising to the data changing at output pin is commonly referred to as clock to out delay. There is finite delay from FF1_out to FF2_in through some combinational logic for the signal to travel.

After this delay signal arrives at second flop and FF2_in falls. Because of large delay from FF1_out to FF2_in, FF2_in falls after the setup requirement of second flop, indicated by the orange/red vertical dotted line. This means input signal to second flop FF2_in, is not held stable for setup time requirement of the flop and hence this flop doesn't correctly capture this data at it's output.

As you can see one would've expected 'Out' node to go high but it doesn't because of setup time or max delay failure at the input of the second flop and it goes metastable. Setup time requirement dictates that input signal be steady during the setup window (which is a certain time before the clock capture edge).

As mentioned earlier if we reduce frequency, our cycle time increases and eventually FF2_in will be able to make it in time and there will not be a setup failure. Also notice that a clock skew is observed at the second flop. The clock to second flop clk2 is not aligned with clk1 anymore and it arrives earlier, which exacerbates the setup failure.

This is a real world situation where clock to all receivers will not arrive at same time and designer will have to account for the clock skew. We'll talk separately about clock skew in details

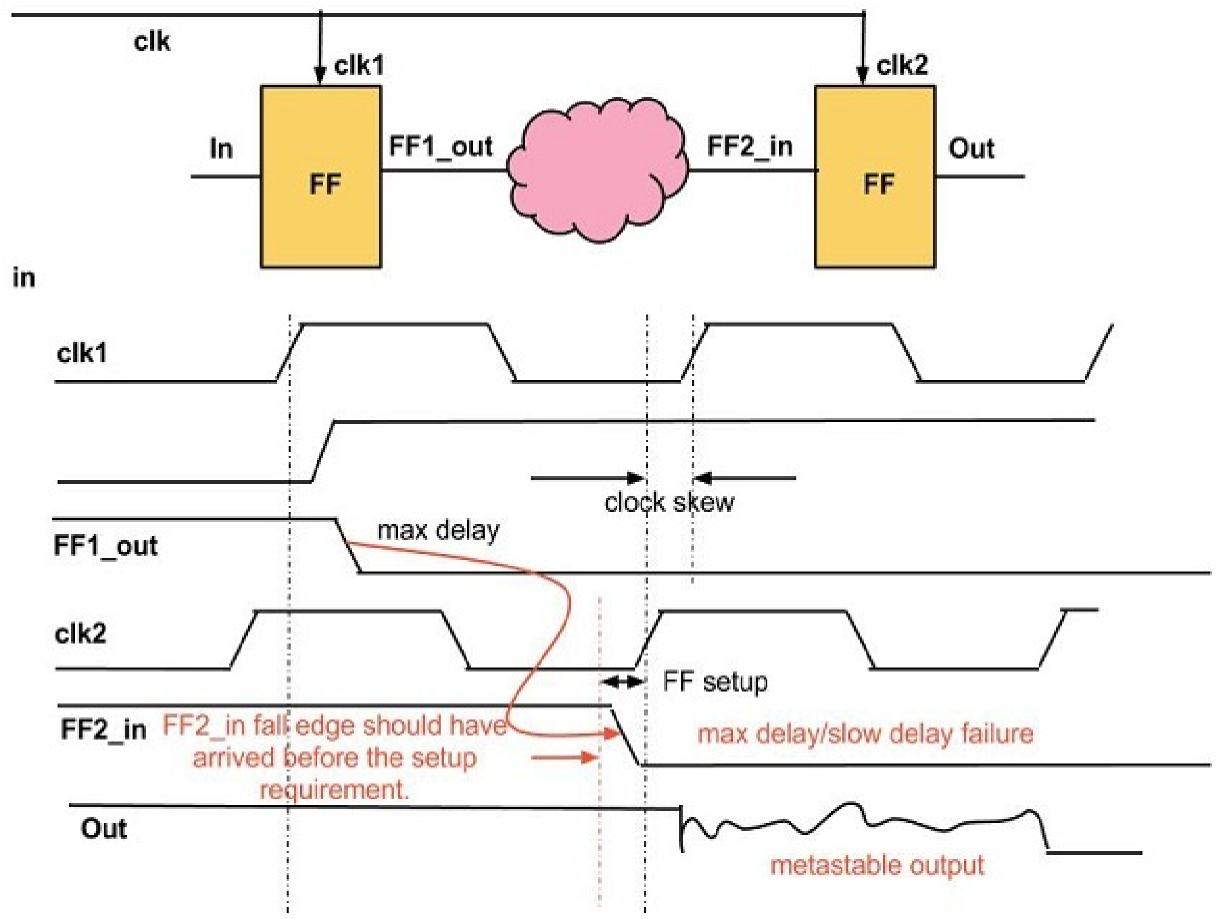


Figure T16. Setup/Max delay failure to a flip-flop.

Question T17) Explain hold failure to a flip-flop.

Answer T17)

Like setup, there is a 'Hold' requirement for each sequential element (flop or a latch). That requirement dictates that after the assertion of the active/capturing edge of the sequential element input data needs to be stable for a certain time/window.

If input data changes within this hold requirement time/window, output of the sequential element could go meta-stable or output could capture unintentional input data. Therefor it is very crucial that input data be held till hold requirement time is met for the sequential in question.

In our figure below, data at input pin 'In' of the first flop is meeting setup and is correctly captured by first flop. Output of first flop 'FF1_out' happens to be inverted version of input 'In'.

As you can see once the active edge of the clock for the first flop happens, which is rising edge here, after a certain clock to out delay output FF1_out falls. Now for sake of our understanding assume that combinational delay from FF1_out to FF2_in is very very small and signal goes blazing fast from FF1_out to FF2_in as shown in the figure below.

In real life this could happen because of several reasons, it could happen by design (imagine no device between first and second flop and just small wire, even better think of both flops abutting each-other), it could be because of device variation and you could end up with very very fast device/devices along the signal path, there could be capacitance coupling happening with adjacent wires, favoring the transitions along the FF1_out to FF2_in, node adjacent to FF2_in might be transitioning high to low(fall) with a sharp slew rate or slope which couples favorably with FF2_in going down and speeds up FF2_in fall delay.

In short in reality there are several reasons for device delay to speed up along the signal propagation path. Now what ends up happening because of fast data is that FF2_in transitions within the hold time requirement window of flop clocked by clk2 and essentially violates the hold requirement for clk2 flop.

This causes the the falling transition of FF2_in to be captured in first clk2 cycle where as design intention was to capture falling transition of FF2_in in second cycle of clk2.

In a normal synchronous design where you have series of flip-flops clocked by a grid clock(clock shown in figure below) intention is that in first clock cycle for clk1 & clk2, FF1_out transitions and there would be enough delay from FF1_out to FF2_in such that one would ideally have met hold requirement for the first clock cycle of clk2 at second flop and FF2_in would meet setup before the second clock cycle of clk2 and when second clock cycle starts, at the active edge of clk2 original transition of FF1_out is propagated to Out.

Now if you notice there is skew between clk1 and clk2, the skew is making clk2 edge come later than the clk1 edge (ideally we expect clk1 & clk2 to be aligned perfectly, that's ideally !!). In our example this is exacerbating the hold issue, if both clocks were perfectly aligned, FF2_in fall could have happened later and would have met hold requirement for the clk2 flop and we wouldn't have captured wrong data !!

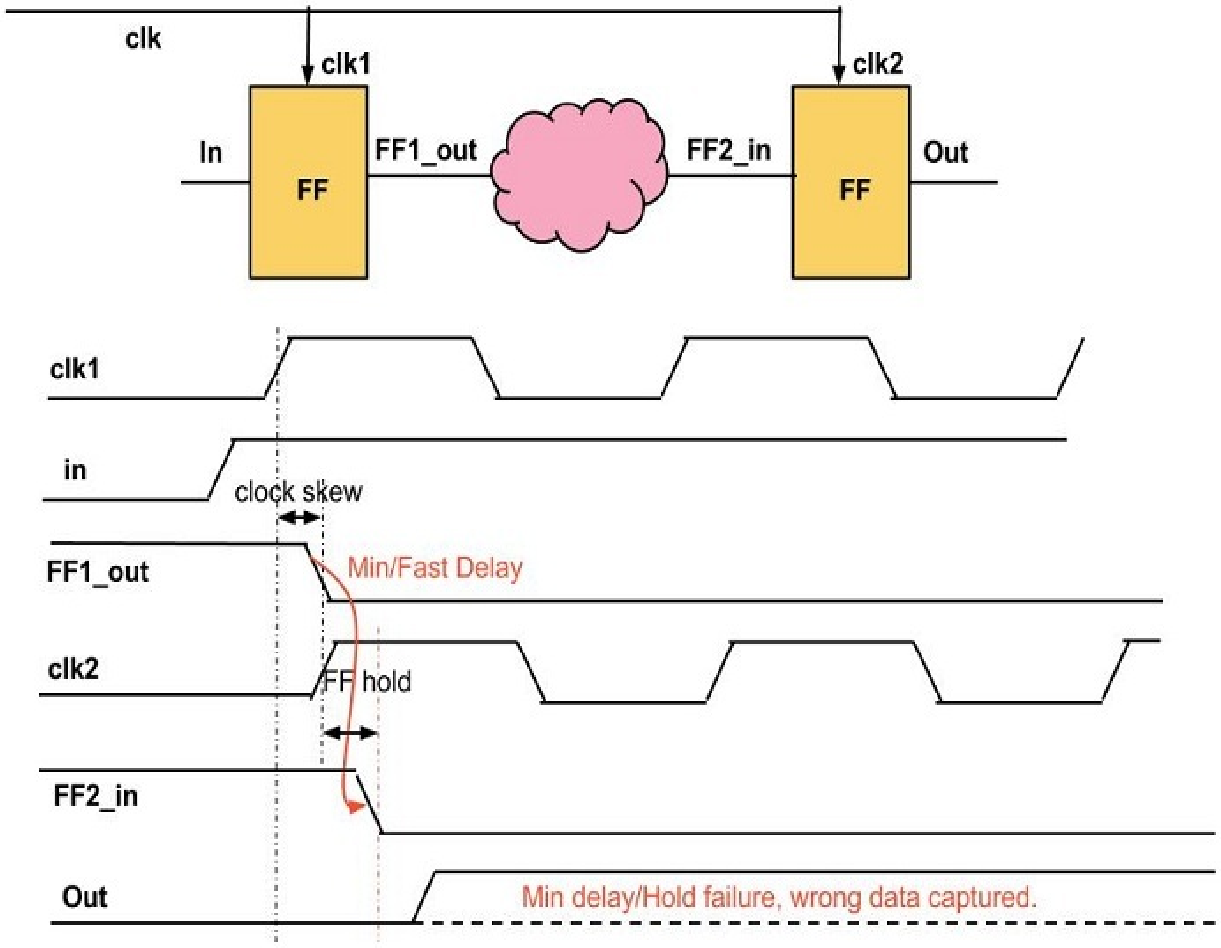


Figure T17. Hold/Min delay requirement for a flop.

Question T18): How do you synchronize between 2 clock domains?

Answer T18):

There are two ways to do this.

1) Asynchronous FIFO,

2) Synchronizer.

Question T19) STA tool reports a hold violation on following circuit. What would you do ?

Answer T19)

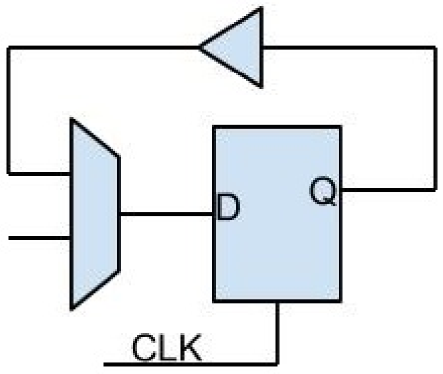


Figure T19. Hold/Min delay requirement for a flop.

If you go through previous question about hold violation failure, you'll realize that hold violation normally arises when generating clock and sampling clock are physically separate clocks and because usually there is large clock skew between the clock to the generating flop and the clock to sampling flop.

In this example we're referring to the hold violation reported by tool where the timing path starts at the CLK pin goes through the Q pin through buffer, MUX and comes back to D input pin of the same flop and ends there.

It is obvious that generating CLK edge and sampling CLK is essentially the same edge. This path would never have a real hold violation as we're referring to the same CLK edge. Many times STA tools have limitations and it doesn't realize this situation.

Because the data is released by the very active edge of CLK, against which the hold check is performed, we'll never have a hold violation as long as combined delay for CLK -> Q, buffer and MUX is more than the intrinsic hold requirement of the flop. Remember from the previous question about hold time, that sequentials(flop or latch) have intrinsic hold time requirement which would be more than zero ps in most of the cases.

The key to understand here is that we're referring to the same CLK edge hence no CLK skew and no hold violation.

Question T20) : Why does the delay of MOS device decreases with increasing temperature at high voltage, but decreases with increasing temperature at lower voltages ?

Answer T20) :
This effect is also referred to as low voltage Inverted Temperature Dependence.

Lets first see, what does the delay of a MOS transistor depend upon, in a simplified model.

Delay = (Cout * Vdd)/ Id   [approx]

Where
Cout = Drain Cap
Vdd = Supply voltage
Id = Drain current.

Now lets see what drain current depends upon.

$Id = \mu(T) * (Vdd - Vth(T))^{\alpha}$

Where
μ = mobility
Vth = threshold voltage
α = positive constant (small number)

One can see that Id is dependent upon both mobility μ and threshold voltage Vth. Let examine the dependence of mobility and threshold voltage upon temperature.

$\mu(T) = \mu(300) (300/T)^{m}$
$Vth(T) = Vth(300) - \kappa(T - 300)$
here ‘300’ is room temperature in kelvin.

Mobility and threshold voltage both decreases with temperature. But decrease in mobility means less drain current and slower device, whereas decrease in threshold voltage means increase in drain current and faster device.

The final drain current is determined by which trend dominates the drain current at a given voltage and temperature pair. At high voltage mobility determines the drain current where as at lower voltages threshold voltage dominates the darin current.

This is the reason, at higher voltages device delay increase with temperature but at lower voltages, device delay increases with temperature.

Circuit Modelling Questions and Answers.

Question M1): How does CAD tool model standard cell behavior ?
Answer M1):
There are several model available, e.g. constant voltage source, constant current source. Many times it uses a look up table method which extrapolates based on available data.

Question M2): How is RC delayed modelled by tools ? What are the RC delay models ?
Answer M2):
RC delay is modeled as Pi model of varying degree of accuracy. Most popular model for RC network is Elmore delay model. If you assume that your RC is network is composed of Pi segment of R resistance and C capacitance, following represents the RC structure.

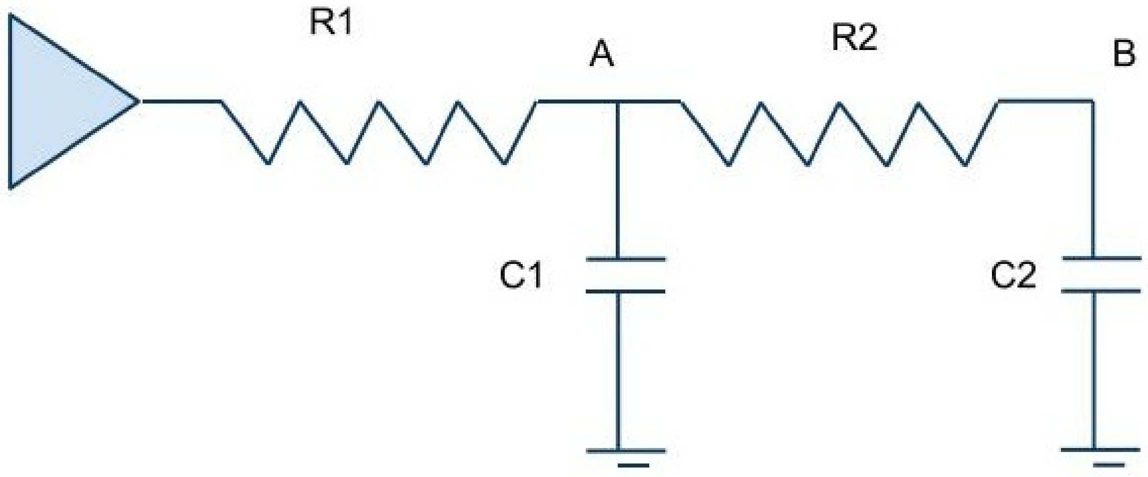


Figure M2. Elmore delay model.

Question M3): What’s the equation for Elmore RC delay ?
Answer M3):
For the above mentioned picture, Elmore delay is following
Total Delay at node B = R1C1 + (R1+R2)C2
If this modular structure is extended than delay at the last node can be represented as following.

Total Delay at node N = R1C1 + (R1+R2)C2 + (R1+R2+R3)C3 + …. (R1+R2+....+RN)CN

Verilog Questions and Answers

Question V1) What is the difference between blocking and non-blocking statements in Verilog ?
Answer V1):
Verilog has two types of procedural assignment statements
1) blocking and
2) non-blocking.
Corresponding assignment operators are = and <=.

1) blocking assignments :
- The blocking assignment statement (=) behaves much like in traditional programming languages. The whole statement is done before execution passes on to the next statement.
- Execution flow within the active procedure is blocked until the assignment is completely finished.
- Evaluations of concurrent statements within the same time step are blocked until the assignment is completely finished.

2) non-blocking assignment is a two step process.
Step 1) At the beginning of the current time unit or the simulation time step, it evaluates all the right-hand side(RHS) of the statement. It comes up with the evaluated value for the left hand side(LHS) and schedules the evaluated value into the non-blocking assign updates events queue.

Step 2) At the end of the current time unit, the LHS value is assigned at the end of the time unit.

In between these two steps of evaluation and update, it does not block assignments of other statement, which is the reason, such assignment is called “non-blocking” assignment.

In reality all Verilog assignments be it blocking or non-blocking, are two step process. First step is the evaluation of the RHS and second step is update of LHS. In blocking statements both steps are done as an atomic operation, hence nothing else can block the assignment, whereas in non-blocking assignments evaluate and update are not atomic operation, but are done at different times and they do not block other assignments.

It is illegal to use non-blocking assignment in a continuous assignment statement or in a net declaration.

```
// Blocking and non-blocking assignment

module blocking;
reg [0:2] X, Y;
initial begin: init1
X = 5;
#1 X = X + 1; // blocking procedural assignment
Y = X + 1;

$display("Blocking: X= %b Y= %b", X, Y );
X = 5;
#1 X <= X + 1; // non-blocking procedural assignment
Y <= X + 1;
#1 $display("Non-blocking: X= %b Y= %b", X, Y );
end
endmodule
```

produces the following output:

```
Blocking: X= 110 Y= 111
Non-blocking: X= 110 Y= 110
```

The effect is that for all non-blocking assignments use the old values of the variables at the beginning of the current time unit and assigns 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 sequentials.

Question V2) Assuming all variables have initial value of ‘0’, when simulation starts, what will be the value of ‘x’ at the end of simulation time unit 4 ?

```
initial begin
        x = 2
        #4
        y <= #9 x
        x = 1
end
always @(x,y)
begin
        z = #2 x + y
end
```

Answer V2):
In order to get a comprehensive understanding of the execution order, one has to look at what is called ‘Verilog stratified event queue’, which is the Verilog IEEE standard spec algorithm describing the event queues.

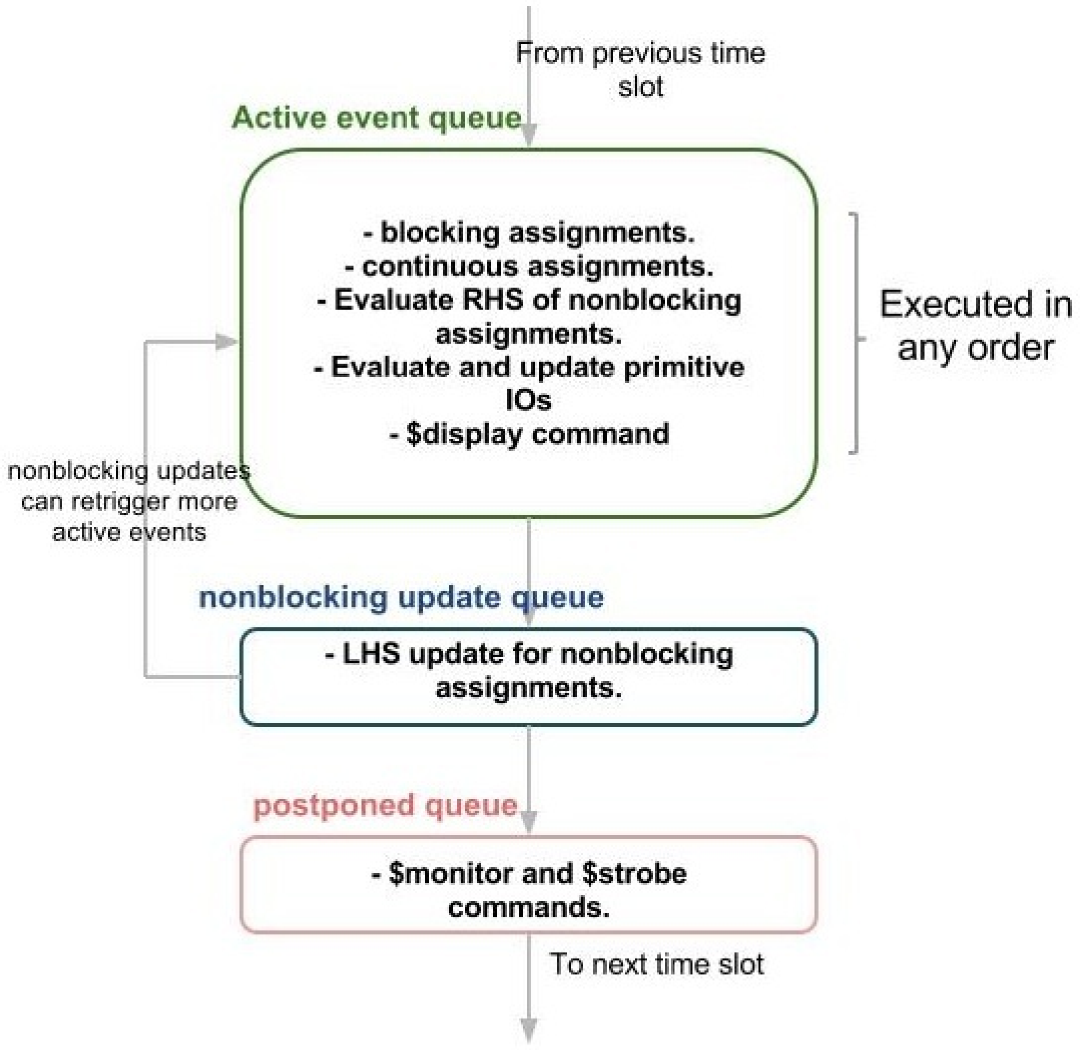

From previous time slot
Active event queue
- blocking assignments.
- continuous assignments.
- Evaluate RHS of nonblocking assignments.
- Evaluate and update primitive IOs
- $display command
Executed in any order
nonblocking updates can retrigger more active events
nonblocking update queue
- LHS update for nonblocking assignments.
postponed queue
- $monitor and $strobe commands.
To next time slot

Figure V2. Verilog Event Queues.

As per standard the event queue is logically segmented into four different regions. For sake of simplicity we're showing the three main event queues. The "Inactive" event queue has been omitted as #0 delay events that it deals with is not a recommended guideline.

As you can see at the top there is 'active' event queue. According to the IEEE Verilog spec, events can be scheduled to any of the event queues, but events can be removed only from the "active" event queue. As shown in the image, the 'active' event queue holds blocking assignments, continuous assignments. primitive IO updates and $write commands. Within "active" queue all events have same

priority, which is why they can get executed in any order and is the source of nondeterminism in Verilog.
There is a separate queue for the LHS update for the nonblocking assignments. As you can see that LHS updates queue is taken up after “active” events have been exhausted, but LHS updates for the nonblocking assignments could re-trigger active events.

Lastly once the looping through the “active” and non blocking LHS update queue has settled down and finished, the “postponed” queue is taken up where $strobe and $monitor commands are executed, again without any particular preference of order.

At the end simulation time is incremented and whole cycle repeats.

Getting back to our original question.

- Initial block starts execution. Inside initial block we have a begin, end procedural block, which means statements will be activated as per the order they appear.
- At beginning of simulation unit time 0, the variable ‘x’ is assigned value of 2 through blocking assignment. ‘x’ is now 2.
- This new value of ‘x’ at time 0 triggers always block and RHS of the equation is evaluated and value if ‘z’ is slated to be updated at simulation time unit 2 with the value of 2(x + y = 2 + 0 = 2)
- Control is back inside initial statement.
- We encounter #4 so, execution in initial block is suspended.
- At simulation time unit 2, ‘z’ is updated with the previously evaluated value of 2.
- Back in initial block at time unit 4, execution proceeds and non blocking statement is encountered, the RHS is evaluated, which is ‘x’ with value 2.
- LHS of non blocking statement is scheduled to be updated 9 time units later.
- Execution proceeds inside initial block at simulation time unit 4 and next is blocking update of ‘x’ with value 1.
- At the end of simulation time unit value of ‘x’ is 1.

Question V3): What will be the final value of x in following example ?

```
always @(posedge clk) x = 0
always @(posedge clk) x = 1
```

Answer V3): This is a race condition and final value of x can not be predefined as two blocking assignments could execute in either order.

Sensitivity list for both always blocks is same. When clock rises, for that current time slot, both blocking assignments are queued in the active event queue, but their order is not predefined. It could be either order, which is why they could execute in either order and during different simulations you get different results based on the order they got executed.

Question V4): What hardware will following piece of code infer ?

part 1)

```
reg x, y, z
always @(posedge clock)
begin
        x=0
        y=x
        z=y
end
```

part 2)

```
reg x, y, z
always @(posedge clock)
begin
        x<=0
        y<=x
        z<=y
end.
```

Answer V4)

Part 1) is using blocking assignment, hence 0 is propagated through ‘x’ and ‘y’ to ‘z’ and we get following single flop !

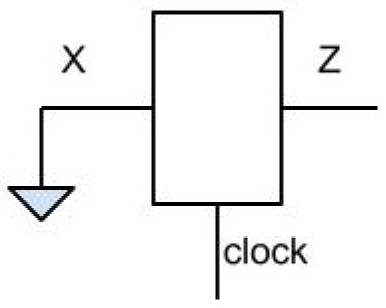


Figure V41: Blocking assignment synthesis.

Part 2) is non-blocking and because of which, value of 0 is not propagated through but each register output is dependent upon register input, which means we infer proper shift register.

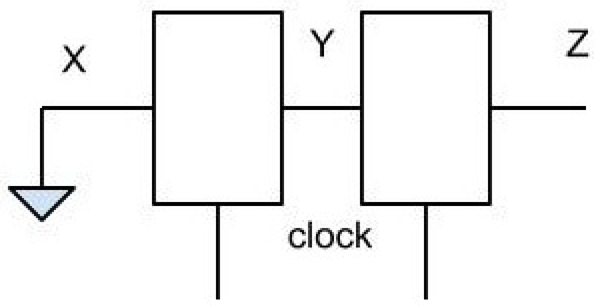


Figure V42: Non-blocking synthesis.

Question V5): Will following procedural block get triggered if value of ‘c’ goes from high to low ?

```
always @(c or d)
begin
        x = c + d
        y = w
end
```

Answer V5)
Yes it will be triggered, because when none of the posedge or negedge is specified in the @(..) sensitivity list, default behavior is to trigger the procedural block at either posedge or negedge event.

Question V6): For the following buffer in the figure and input waveform, show the buffer output waveform for each case of verilog code snippet representing buffer?

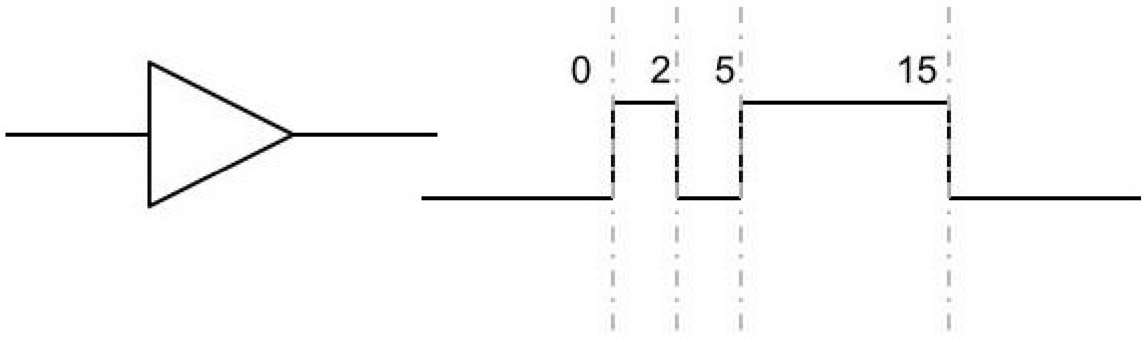


Figure V61: Buffer and input waveform.

code snippet 1)

```
always @(input)
 output = #5 input ;
```

code snippet 2)

```
always @(input)
 #5 output = input ;
```

Answer V6)

This question attempt to check for your understanding of hardware delay modelling.

There are two primary hardware delay propagation methods : Inertial delay and Transport delay.

Inertial delay : This delay models devices with finite switching speeds. Input glitches shorter than the device input to output delay are not propagated.

One can put the delay time control before the assignment to model inertial delay.
For example :
#<delay> output = input
This will cause the whole assignment to be evaluated after <delay> number of time units. As you can see if there is glitch on ‘input’ which is smaller than the <delay> value, it will be filtered as assignment evaluation will be suspended for <delay> number of time units. This is

also called inter assignment delay. Key to remember is that while assignment is suspended for <delay> number of time units, if ‘input’ changes in the interim, those changes are lost as assignment is suspended during that time !

Transport delay : This delay models devices with close to infinite switching speeds. Input glitches are propagated to the output of the device, because of infinite switching speeds.

One can use intra assignment delay control to model transport delay. You will put the delay control after assignment.
For example :
output = #<delay> input.
This will cause the ‘input’ to be evaluated first and update to happen <delay> units later. Even if there is glitch on ‘input’ which is smaller than <delay> value, the changes to ‘input’ are registered and propagated to ‘output’ after <delay> time units. Key to intra assignment delay with blocking assignment is that while LHS update is waiting on the <delay> time units, if ‘input’ changes in the interim, those changes are lost ! Which really means this is not a pure transport delay example. As we’ll shortly see that, at output we will not get the same waveform as input. If we truly want the intermediate changes to the ‘input’ to be registered, while assignment is suspended on delay control, we need to use non-blocking assignment.

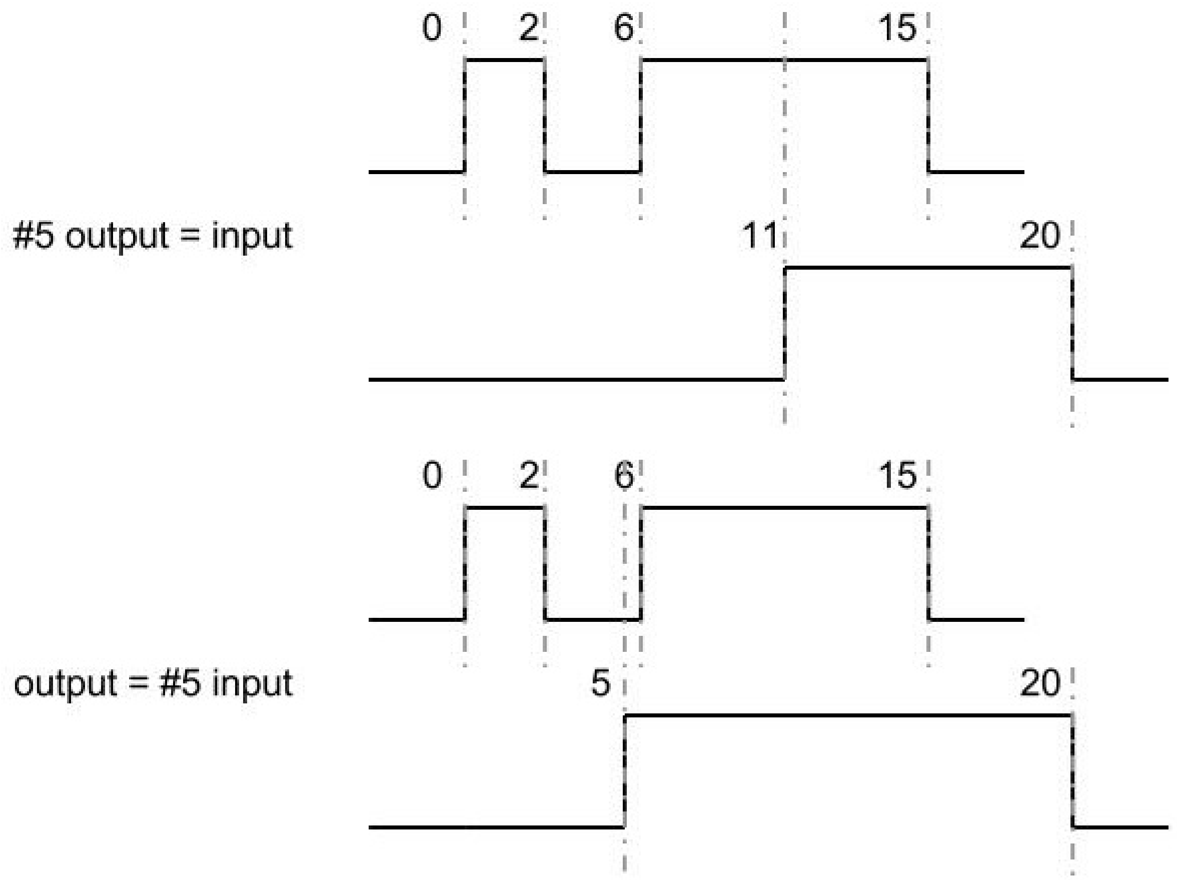


Figure V62: Blocking inter & intra assignment delay waveforms.

We can easily see that code snippet 2) looks to be representing inertial delay, where initial glitch will get filtered. At time unit ‘0’, ‘input’ changes, hence blocking assignment ‘output = input’ is scheduled to be executed 5 time units later. 5 units later when assignment executes it checks current value of ‘input’ which is ‘0’ now, so no change in output. In the meantime ‘input’ change at time unit 2 is lost. Next ‘input’ changes at time unit 6, which is propagated to output at 11 and ‘input’ change from 15 is propagated to output at time unit 20. We lost the glitch.

Code snippet 1) execution is similar to previous one, except that when LHS update happens it uses older value of ‘input’ and not the current value of ‘input’. At time 0, ‘input’ change, which is reflected at time unit 5 on output. ‘input’ changes at time unit 2 lost, next ‘input’ changes to high at time unit 6, which means no change in output waveform at time unit 11, as output was already high and so on. This can represent transport delay if we change assignment to nonblocking type, because in nonblocking assignment as mentioned earlier, while assignment is waiting on delay control, it still registers changes on ‘input’.

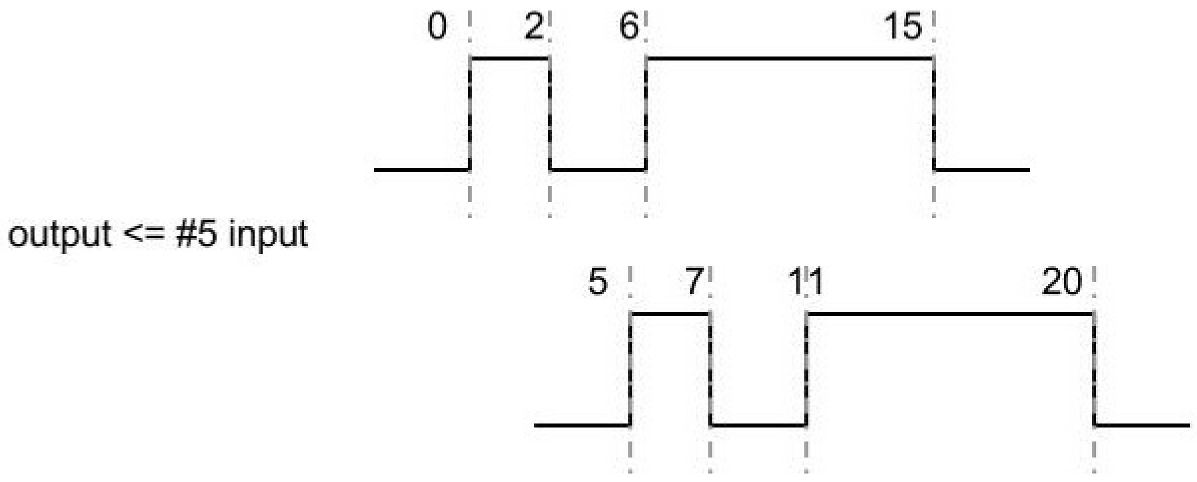


Figure V63: Non-blocking intra assignment and transport delay

Question V7): Why would you use a Verilog Pre-processor?
Answer V7):
It’s mainly used for certain loop constructs for multiple instantiations

Misc. Questions with Answers

Question S1): What is a low pass filter circuit ?
Answer S1):
It is a filter that passes low-frequency signals. It attenuates frequencies that go beyond the cut-off frequency

Reference:
http://en.wikipedia.org/wiki/Low-pass_filter

Question S2): What is a boost converter ?
Answer S2):
A boost converter is an electronic system that allows attaining a greater DC level output compared to the level of its input.

Reference
http://en.wikipedia.org/wiki/Boost_converter

Question S3): Why is common collector configuration not used for amplifier?
Answer S3):
Because it doesn't amplify

Question S4): How to improve the linearity of the common source amplifier?
Answer S4):
One way is to tie source to the body node to remove the body effect from changing the threshold voltage.