ECE 327/627
Digital Hardware Systems

Tutorial 3

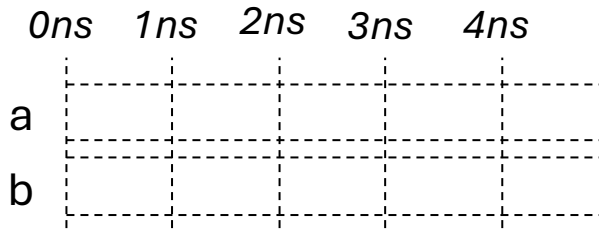
Q1

Draw the timing diagram associated with this code up to 4ns.

When you are presented with code that has inter-assignment delays between non-blocking assignments, best way to reason about it is to think in sections with the specified delays/pauses between them.

```
`timescale 1ns/1ns

module q2 ();

logic [3:0] a, b;

initial begin
      a <= 4'd7;
      b <= a;
      #2 b <= a + 1;
      b <= a + 3;
      #1 b <= a + 2;
end

endmodule
```

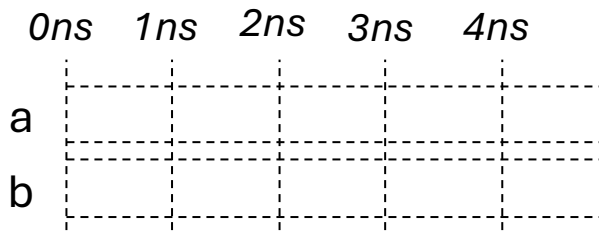
Inter-assignment delays



Q1

Draw the timing diagram associated with this code up to 4ns.

When you are presented with code that has inter-assignment delays between non-blocking assignments, best way to reason about it is to think in sections with the specified delays/pauses between them.




```
`timescale 1ns/1ns

module q2 ();

logic [3:0] a, b;

initial begin
      a <= 4'd7;
      b <= a;
      #2;
      b <= a + 1;
      b <= a + 3;
      #1;
      b <= a + 2;
end

endmodule
```

These happen concurrently

Wait for 2ns

Inter-assignment delays

These happen concurrently

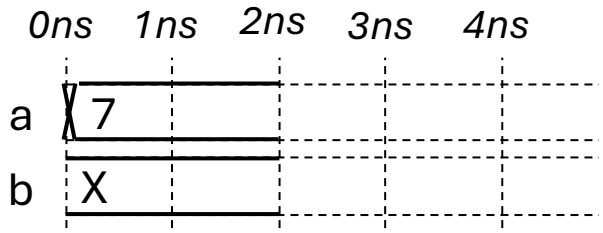
Wait for 1ns

Q1

Draw the timing diagram associated with this code up to 4ns.

When you are presented with code that has inter-assignment delays between non-blocking assignments, best way to reason about it is to think in sections with the specified delays/pauses between them.

For the first section "a" changes to 7 and at the same time "b" changes to the (old) value of "a" which is undefined. Then you wait for 2ns.




```
`timescale 1ns/1ns

module q2 ();

logic [3:0] a, b;

initial begin
      a <= 4'd7;
      b <= a;
      #2;
      b <= a + 1;
      b <= a + 3;
      #1;
      b <= a + 2;
end

endmodule
```

These happen concurrently

Wait for 2ns

Inter-assignment delays

These happen concurrently

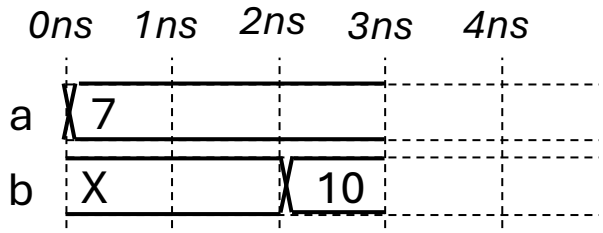
Wait for 1ns

Q1

Draw the timing diagram associated with this code up to 4ns.

When you are presented with code that has inter-assignment delays between non-blocking assignments, best way to reason about it is to think in sections with the specified delays/pauses between them.

For the second section "b" is changed using two consecutive non-blocking assignments that happen at the same time. Last assignment wins, so "b" is set to "a+3" @ 2ns. Then you wait for 1ns.




```
`timescale 1ns/1ns

module q2 ();

logic [3:0] a, b;

initial begin
      a <= 4'd7;
      b <= a;
      #2;
      b <= a + 1;
      b <= a + 3;
      #1;
      b <= a + 2;
end

endmodule
```

These happen concurrently

Wait for 2ns

These happen concurrently

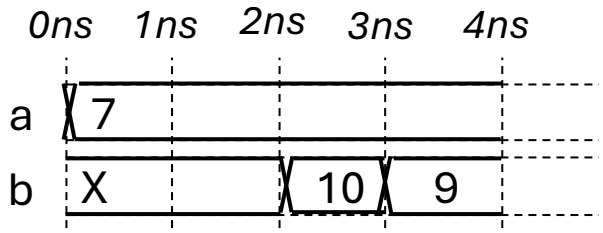
Wait for 2ns

Q1

Draw the timing diagram associated with this code up to 4ns.

When you are presented with code that has inter-assignment delays between non-blocking assignments, best way to reason about it is to think in sections with the specified delays/pauses between them.

For the third section, "b" is changed to "a+2" @ 3ns. Then nothing happens until 4ns.




```
`timescale 1ns/1ns

module q2 ();

logic [3:0] a, b;

initial begin
      a <= 4'd7;
      b <= a;
      #2;
      b <= a + 1;
      b <= a + 3;
      #1;
      b <= a + 2;
end

endmodule
```

These happen concurrently

Wait for 2ns

These happen concurrently

Wait for 2ns

Q2

Will the simulation of this code converge to a stable set of values for a, b, and c? Why or why not?

Initial block changes "b" to 1 at the beginning of time.

This triggers the first always_comb block, which is listening for a change in "b". It changes the value of "c" to 0+1 = 1.

This triggers the second always_comb block, which is listening for a change in "c". It changes the value of "a" to 1+1=2.

This then triggers the first always_comb block again, which is listening for a change in "a". It changes the value of "c" to 2+1=3.

This then triggers the second always_comb block again … and this keeps going indefinitely without reaching stable signal values to advance the simulation time step.

```
`timescale 1ns/1ns

module q3 ();

logic [3:0] a = 4'd0, b, c;

initial begin
     b = 4'd1;
end

always_comb begin
     c = a + b;
end

always_comb begin
     a = c + 1;
end

endmodule
```

Q3

Draw the waveform resulting from simulating this code.

This question plays with intra-assignment delays and using them with blocking vs. non-blocking assignments.

```
`timescale 1ns/1ns

module q4 ();

logic a = 1'b0;
logic b = 1'b0;
logic c = 1'b0;
logic d = 1'b0;
logic e = 1'b0;
logic f = 1'b0;

initial begin
     a = #10 1'b1;
     b = #20 1'b1;
     c = #30 1'b1;
     #10 $finish();
end

initial begin
     d <= #10 1'b1;
     e <= #20 1'b1;
     f <= #30 1'b1;
end

endmodule
```

Intra-assignment delays

Q3

Draw the waveform resulting from simulating this code.

This question plays with intra-assignment delays and using them with blocking vs. non-blocking assignments.

But first … what is the difference between inter- and intra-assignment delays?

#<delay> <LHS> = <RHS> // inter-assignment

The evaluation of the RHS and the update of the LHS are done after the delay expires.

<LHS> = #<delay> <RHS> // intra-assignment

The evaluation of the RHS is done (i.e., values of all signals on the RHS are captured) first. Then the update of the LHS is done after the delay expires.

```
`timescale 1ns/1ns

module q4 ();

logic a = 1'b0;
logic b = 1'b0;
logic c = 1'b0;
logic d = 1'b0;
logic e = 1'b0;
logic f = 1'b0;

initial begin
      a = #10 1'b1;
      b = #20 1'b1;
      c = #30 1'b1;
      #10 $finish();
end

initial begin
      d <= #10 1'b1;
      e <= #20 1'b1;
      f <= #30 1'b1;
end

endmodule
```

Q3

Draw the waveform resulting from simulating this code.

This question plays with intra-assignment delays and using them with blocking vs. non-blocking assignments.

<u>Remember that:</u>

- *The 2 initial blocks happen concurrently*
- *For blocking assignments (=), the evaluation of the RHS and the update of the LHS of one statement must be done to proceed to the next statement.*
- *For non-blocking assignments (<=), all evaluations of the RHS of all statements happen first. Then, all updates of the LHS happen at the end of the simulation time step*

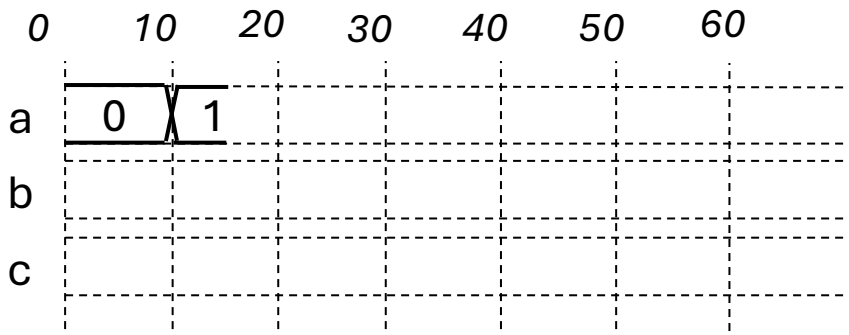
```
`timescale 1ns/1ns

module q4 ();

logic a = 1'b0;
logic b = 1'b0;
logic c = 1'b0;
logic d = 1'b0;
logic e = 1'b0;
logic f = 1'b0;

initial begin
      a = #10 1'b1;
      b = #20 1'b1;
      c = #30 1'b1;
      #10 $finish();
end

initial begin
      d <= #10 1'b1;
      e <= #20 1'b1;
      f <= #30 1'b1;
end

endmodule
```

Q3

Draw the waveform resulting from simulating this code.

This question plays with intra-assignment delays and using them with blocking vs. non-blocking assignments.

We will pick any initial block and focus on it first, since both happen concurrently. Let's start with the first initial block. At the beginning of time (0ns), evaluation of the RHS of the first statement happens, and the update of "a" is delayed by the intra-assignment #10. Only after the update happens @ 10ns, we can proceed to the next statement because this is a blocking assignment.



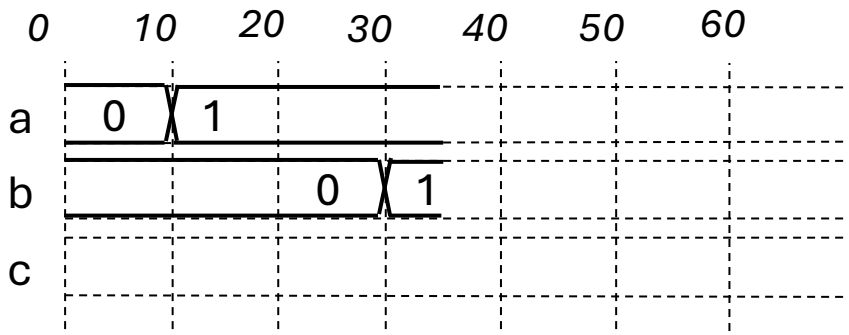

```
`timescale 1ns/1ns

module q4 ();

logic a = 1'b0;
logic b = 1'b0;
logic c = 1'b0;
logic d = 1'b0;
logic e = 1'b0;
logic f = 1'b0;

initial begin
      a = #10 1'b1;
      b = #20 1'b1;
      c = #30 1'b1;
      #10 $finish();
end

initial begin
      d <= #10 1'b1;
      e <= #20 1'b1;
      f <= #30 1'b1;
end

endmodule
```

Q3

Draw the waveform resulting from simulating this code.

This question plays with intra-assignment delays and using them with blocking vs. non-blocking assignments.

At 10ns (current time after 1st statement), evaluation of the RHS of the second statement happens, and the update of "b" is delayed by the intra-assignment #20 (from the current point in time, which is 10ns). Only after the update happens @ 30ns, we can proceed to the next statement because this is a blocking assignment.



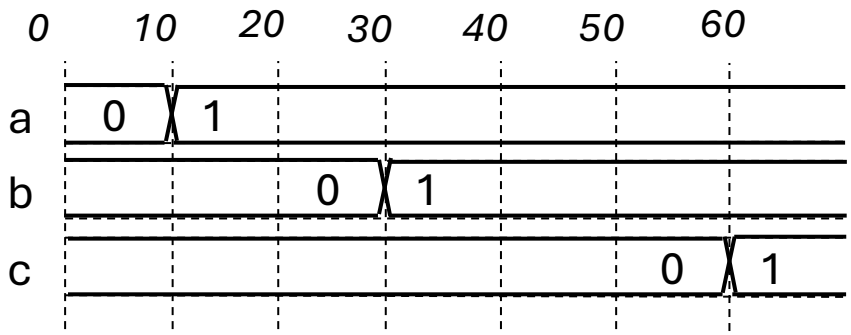

```
`timescale 1ns/1ns

module q4 ();

logic a = 1'b0;
logic b = 1'b0;
logic c = 1'b0;
logic d = 1'b0;
logic e = 1'b0;
logic f = 1'b0;

initial begin
      a = #10 1'b1;
      b = #20 1'b1;
      c = #30 1'b1;
      #10 $finish();
end

initial begin
      d <= #10 1'b1;
      e <= #20 1'b1;
      f <= #30 1'b1;
end

endmodule
```

Q3

Draw the waveform resulting from simulating this code.

This question plays with intra-assignment delays and using them with blocking vs. non-blocking assignments.

At 30ns (current time after 2nd statement), evaluation of the RHS of the third statement happens, and the update of the "c" is delayed by the intra-assignment #30 (from the current point in time, which is 30ns). Only after the update happens @ 60ns, we can proceed to the next statement because this is a blocking assignment. Then we wait for 10ns and end simulation.



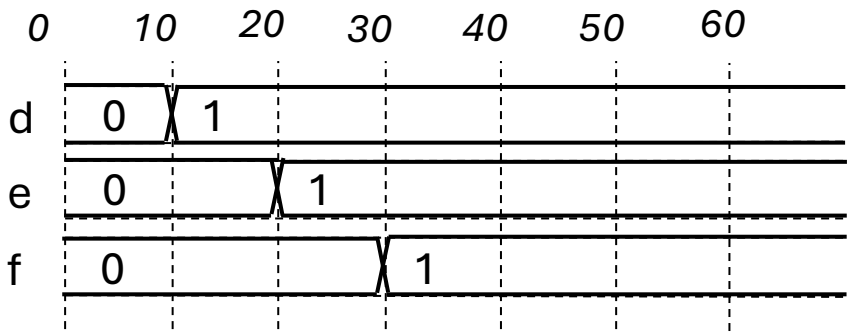

```
`timescale 1ns/1ns

module q4 ();

logic a = 1'b0;
logic b = 1'b0;
logic c = 1'b0;
logic d = 1'b0;
logic e = 1'b0;
logic f = 1'b0;

initial begin
      a = #10 1'b1;
      b = #20 1'b1;
      c = #30 1'b1;
      #10 $finish();
end

initial begin
      d <= #10 1'b1;
      e <= #20 1'b1;
      f <= #30 1'b1;
end

endmodule
```

Q3

Draw the waveform resulting from simulating this code.

This question plays with intra-assignment delays and using them with blocking vs. non-blocking assignments.

Now let's see how that is different for non-blocking assignments in the second initial block. At the beginning of time (0ns), the RHS of all 3 statements are evaluated at the same time. However, their assignments are delayed by 10ns, 20ns, and 30ns relative to the current point in time (0ns). This means that "d" will be set at 10ns, "e" will be set at 20ns, and "f" will be set at 30ns

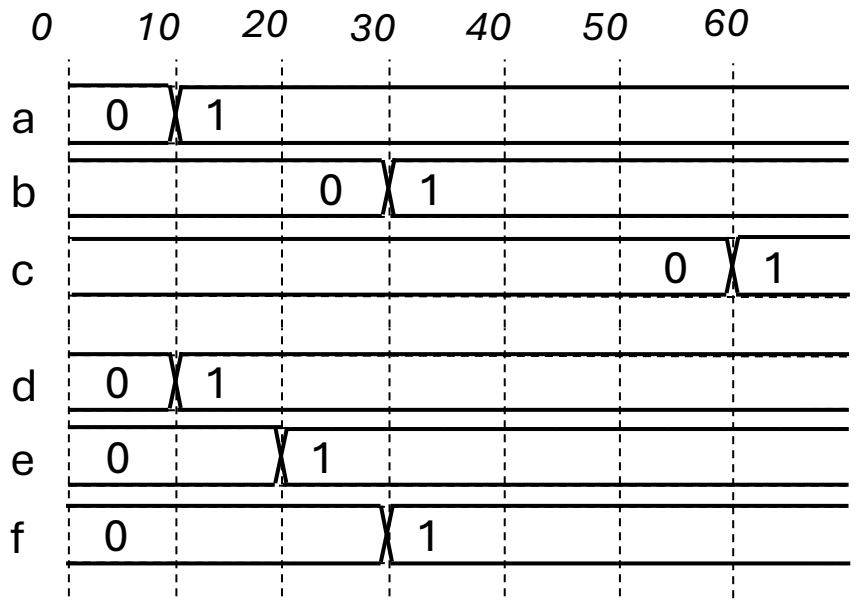



```
`timescale 1ns/1ns

module q4 ();

logic a = 1'b0;
logic b = 1'b0;
logic c = 1'b0;
logic d = 1'b0;
logic e = 1'b0;
logic f = 1'b0;

initial begin
      a = #10 1'b1;
      b = #20 1'b1;
      c = #30 1'b1;
      #10 $finish();
end

initial begin
      d <= #10 1'b1;
      e <= #20 1'b1;
      f <= #30 1'b1;
end

endmodule
```

Q3

Draw the waveform resulting from simulating this code.

So now, here is the full waveform




```
`timescale 1ns/1ns

module q4 ();

logic a = 1'b0;
logic b = 1'b0;
logic c = 1'b0;
logic d = 1'b0;
logic e = 1'b0;
logic f = 1'b0;

initial begin
     a = #10 1'b1;
     b = #20 1'b1;
     c = #30 1'b1;
     #10 $finish();
end

initial begin
     d <= #10 1'b1;
     e <= #20 1'b1;
     f <= #30 1'b1;
end

endmodule
```

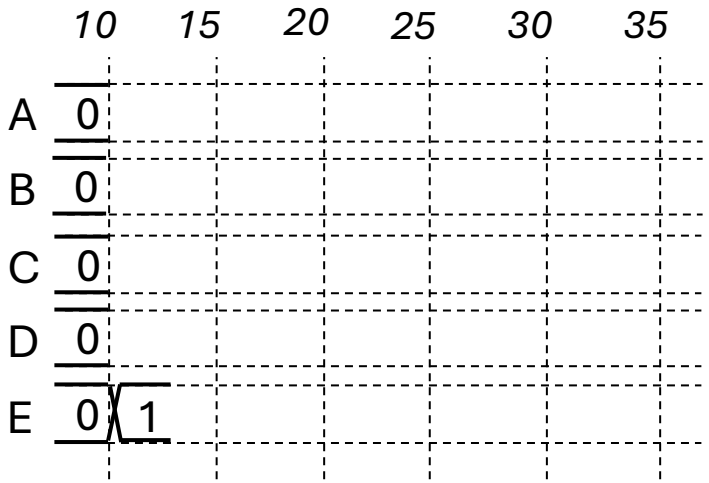
Q4

In the following always block: A, B, C, and D all have a value of ‘0’ at time = 10 ns. If E changes from ‘0’ to ‘1’ at time = 10 ns, specify the time(s) at which each signal will change and the value to which it will change.

```
always @ (E) begin
      A <= #5 1;
      B <= 1;
      C <= A;
      #10;
      D <= #2 A;
end
```

First of all, `always @ (E)` *means that this is a combinational block that is only listening for changes in the value of “E” (instead of any RHS changes as in* `always_comb`*).*

This means that the behavior in this always block will happen at 10 ns when “E” changes from 0 to 1 as specified in the question.



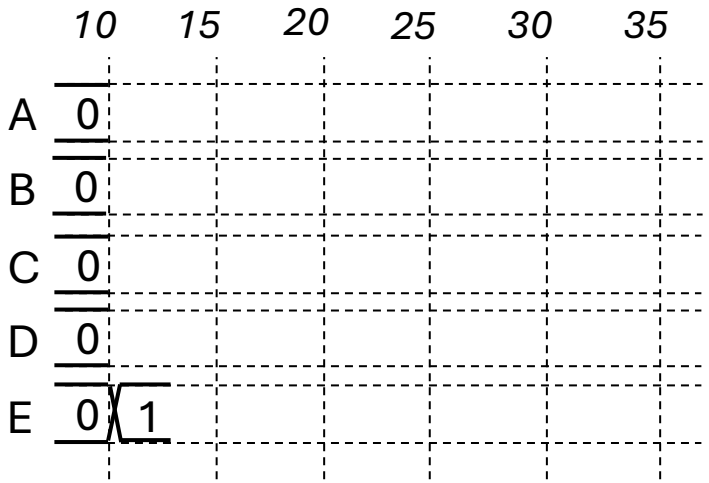
Q4

In the following always block: A, B, C, and D all have a value of ‘0’ at time = 10 ns. If E changes from ‘0’ to ‘1’ at time = 10 ns, specify the time(s) at which each signal will change and the value to which it will change.

```
always @ (E) begin
      A <= #5 1;
      B <= 1;
      C <= A;
      #10;
      D <= #2 A;
end
```

The code snippet has an inter-assignment delay that splits the behavior in two sections with a 10ns wait in between.

The three statements before the #10 delay are non-blocking, which means that the evaluation of their RHS happens first, then all three updates to “A”, “B”, and “C” are performed at the end of the time step.



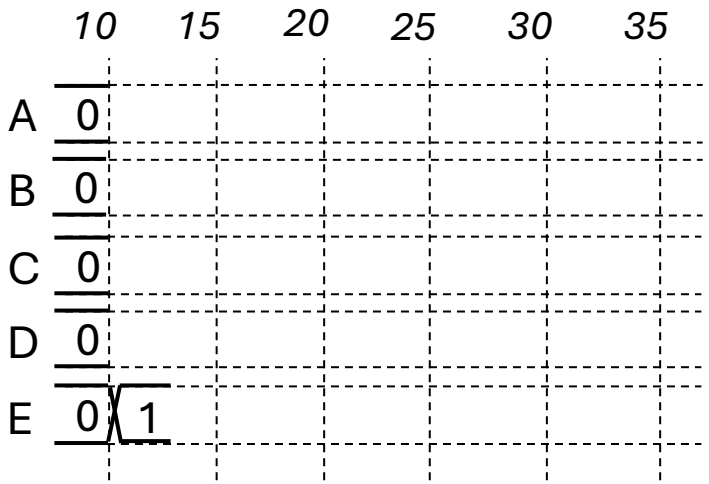
Q4

In the following always block: A, B, C, and D all have a value of ‘0’ at time = 10 ns. If E changes from ‘0’ to ‘1’ at time = 10 ns, specify the time(s) at which each signal will change and the value to which it will change.

```
always @ (E) begin
      A <= #5 1;
      B <= 1;
      C <= A;
      #10;
      D <= #2 A;
end
```

Current point in time is 10ns:

RHS of the 1st statement evaluates to value of 1 but the update is delayed by 5ns due to the intra-assignment delay (#5). This means that “A” is scheduled to be updated at 15ns.



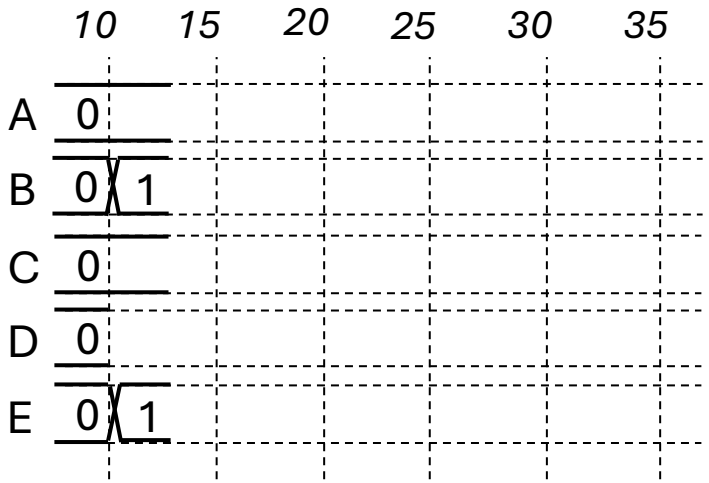
Q4

In the following always block: A, B, C, and D all have a value of ‘0’ at time = 10 ns. If E changes from ‘0’ to ‘1’ at time = 10 ns, specify the time(s) at which each signal will change and the value to which it will change.

```
always @ (E) begin
      A <= #5 1;
      B <= 1;
      C <= A;
      #10;
      D <= #2 A;
end
```

Current point in time is 10ns:

RHS of the 2nd statement evaluates to value of 1 and RHS of the 3rd statement evaluates to the value of “A” at current time (10ns), which is still 0. Then, both are updated at the same time (i.e., “B” changes from 0 to 1 and “C” stays 0).



Q4

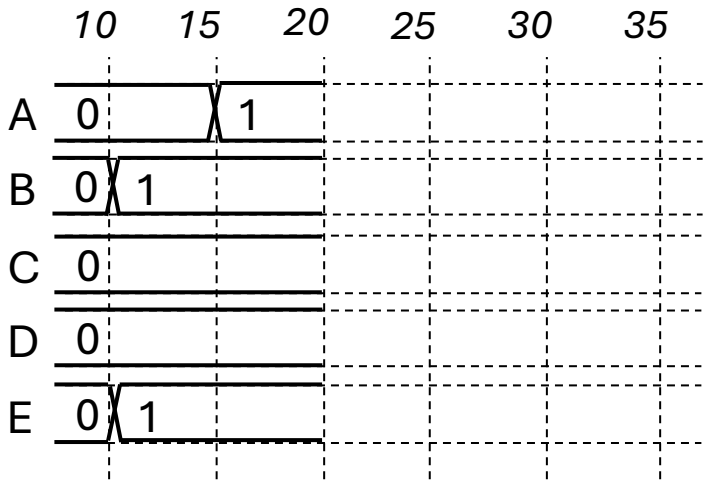
In the following always block: A, B, C, and D all have a value of ‘0’ at time = 10 ns. If E changes from ‘0’ to ‘1’ at time = 10 ns, specify the time(s) at which each signal will change and the value to which it will change.

```
always @ (E) begin
      A <= #5 1;
      B <= 1;
      C <= A;
      #10;
      D <= #2 A;
end
```

Current point in time is 10ns:

Then, the delay statement (#10) advances time by 10ns from the current time, which is 10ns → advance to 20ns.

But remember, along the way, “A” was scheduled to be updated at 15ns (from the first statement).



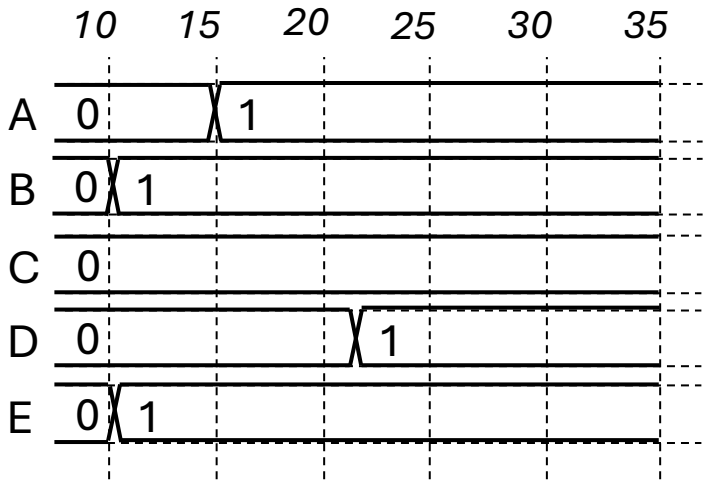
Q4

In the following always block: A, B, C, and D all have a value of ‘0’ at time = 10 ns. If E changes from ‘0’ to ‘1’ at time = 10 ns, specify the time(s) at which each signal will change and the value to which it will change.

```
always @ (E) begin
      A <= #5 1;
      B <= 1;
      C <= A;
      #10;
      D <= #2 A;
end
```

Current point in time is 20ns:

RHS of the 4th statement evaluates to value of “A” at current point in time (20ns), which is 1. The update of “D” is delayed by 2ns due to the intra-assignment delay (#2). This means that “D” is scheduled to be updated at 22ns.



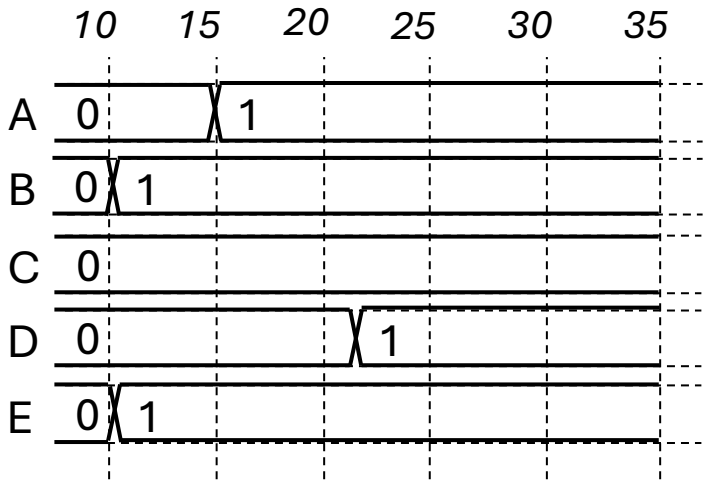
Q4

In the following always block: A, B, C, and D all have a value of ‘0’ at time = 10 ns. If E changes from ‘0’ to ‘1’ at time = 10 ns, specify the time(s) at which each signal will change and the value to which it will change.

```
always @ (E) begin
      A <= #5 1;
      B <= 1;
      C <= A;
      #10;
      D <= #2 A;
end
```

Current point in time is 20ns:

RHS of the 4th statement evaluates to value of “A” at current point in time (20ns), which is 1. The update of “D” is delayed by 2ns due to the intra-assignment delay (#2). This means that “D” is scheduled to be updated at 22ns.



So, the final answer is:

- *“A” changes from ‘0’ to ‘1’ @ 15ns*
- *“B” changes from ‘0’ to ‘1’ @ 10ns*
- *“C” stays ‘0’*
- *“D” changes from ‘0’ to ‘1’ @ 22ns*

Q5

Implement a 4-bit counter that increments by 2 at every positive clock edge, when enable is 1. Also provide an active high reset signal (when reset is 1, then the counter is set to 0 synchronously). Draw a waveform of the counter simulation using the provided testbench code.

```
module counter (
     input clk,
     input rst,
     input en,
     output logic [3:0] count
);

// Write your code here

endmodule
```

```
`timescale 1ns/1ns

module counter_tb();

logic clk, rst, en;
logic [3:0] count;

counter dut (clk, rst, en, count);

initial begin
     clk = 1'b0;
     forever #5 clk = ~clk;
end

initial begin
     rst = 1'b1; en = 1'b0;
     #22;
     rst = 1'b0; en = 1'b1;
     #100;
     $finish();
end

endmodule
```

Q5

Implement a 4-bit counter that increments by 2 at every positive clock edge, when enable is 1. Also provide an active high reset signal (when reset is 1, then the **counter is set to 0 synchronously**). Draw a waveform of the counter simulation using the provided testbench code.

What does "counter is set to 0 synchronously"?

Reset logic happens at +ve clock edge if rst is 1 (synchronous reset)

```
always_ff @ (posedge clk) begin
     if (rst) begin
          // Reset logic
     end else begin
          // Other logic
     end
end
```

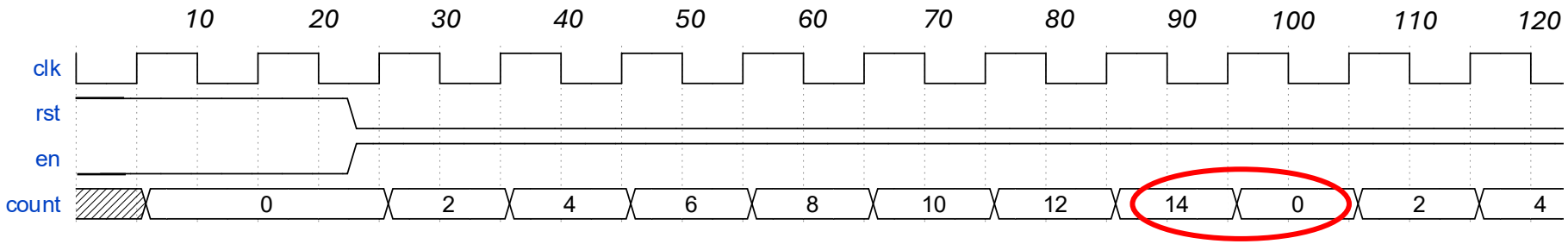
Reset logic happens at the rising edge of rst signal regardless at the clock edge or not (asynchronous reset)

```
always_ff @ (posedge clk or posedge rst) begin
     if (rst) begin
          // Reset logic
     end else begin
          // Other logic
     end
end
```

Q5

Implement a 4-bit counter that increments by 2 at every positive clock edge, when enable is 1. Also provide an active high reset signal (when reset is 1, then the counter is set to 0 synchronously). Draw a waveform of the counter simulation using the provided testbench code.

```
module counter (
      input clk,
      input rst,
      input en,
      output logic [3:0] count
);

always_ff @ (posedge clk) begin
  if (rst) begin
    count <= 0;
  end else begin
    if (en) count <= count + 2;
  end
end

endmodule
```



Count overflows to zero because it is 4 bits (highest possible value is 4'b1111 = 15)