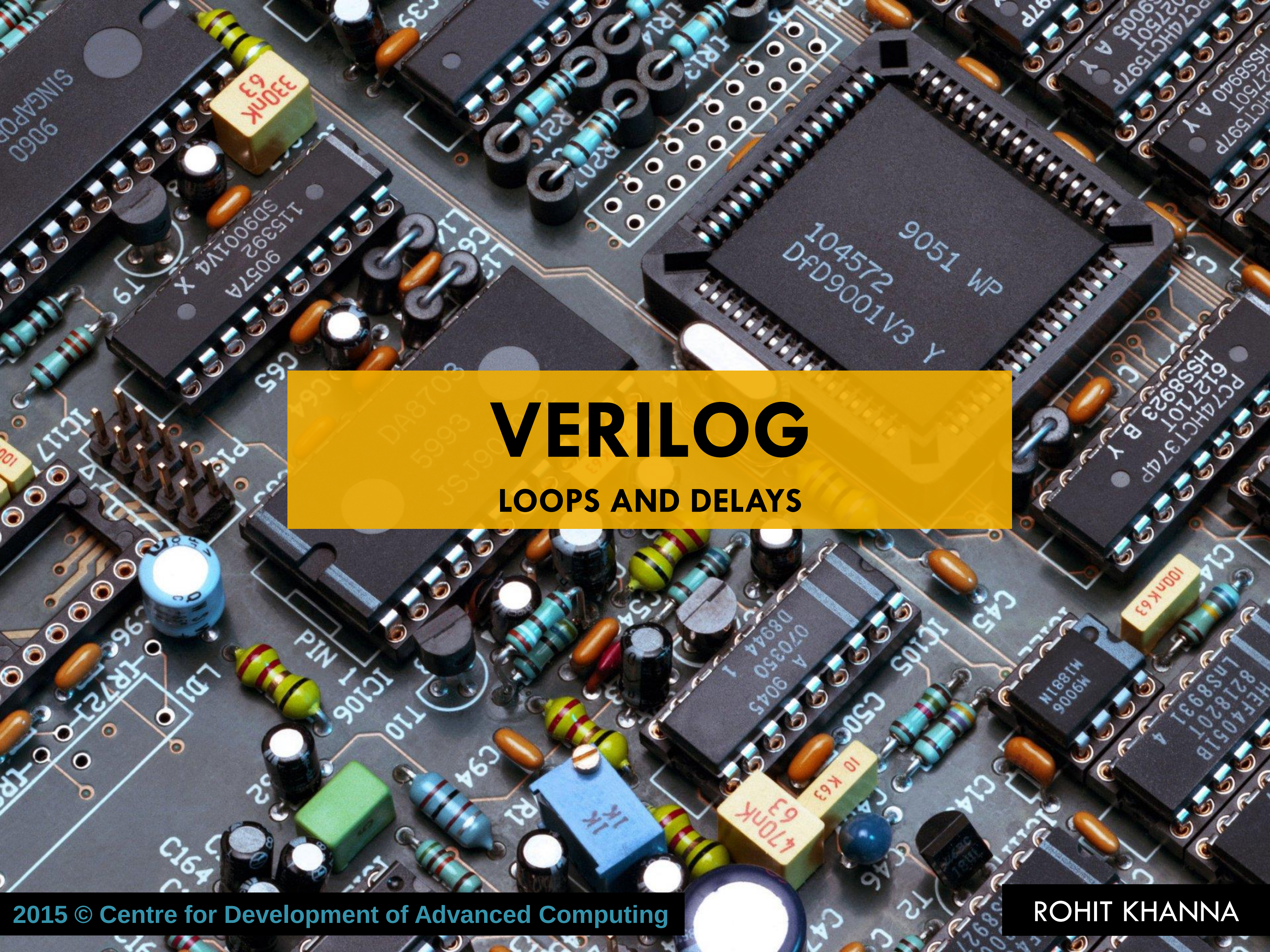


VERILOG

LOOPS AND DELAYS

ROHIT KHANNA

Loops

- Verilog provides four type of loop statements
  - while
  - for
  - repeat
  - forever
- These loops can appear only inside always or initial block.
- Loop may contain delay statements.

while

- while loop executes until the while expression is not true.

Syntax

```
while (<expression>)
begin
statements;
end
```

Example

```
integer count=0;

initial
while (count != 7)
#1 count= count + 1;
```

Result

count=7

for

- for loop executes till the condition is true.

Syntax

```
for (initialization; condition check; updating )
begin
statements;
end
```

Example

```
integer count=0, i;

initial
for (i=0; i<=7; i= i + 1)
#1 count= count + 1;
```

Result

count=8

repeat

- Repeat loop executes the loop a fixed number of times

Syntax

```
repeat (<no of times>)
begin
statements;
end
```

Example

```
integer count=0;

initial
repeat (10)
#1 count= count + 1;
```

Result

count=10

forever

- forever is an infinite loop that executes without any condition

Syntax

```
forever
begin
statements;
end
```

Example

```
reg clock=0;

initial
forever #5 clock= ~clock;
```

Block Statements

- Block statements are used to group multiple statements together.

- There are two type of Block Statements
  - Sequential Blocks
  - Parallel Blocks

- Sequential Blocks:
  - begin and end keywords are used to group statements.
  - Statements are executed in the order they are specified.
  - Delay or event control is relative to the time when previous statement got executed.

Block Statements

- Parallel Blocks:
  - fork and join keywords are used to group statements.
  - Ordering of statements is controlled by delay or event control assigned to each statement.
  - Delay or event control is relative to the time when the block was entered.
- Sequential blocks can be used for synthesis and simulation.
- Parallel blocks can only be used for simulation.
- Nesting of Block statements is allowed.

Block Statements

Example: Sequential Block

```
integer a=0, b=0, c=0, d=0, e=0;

initial
begin                         // Sequential block
a=5;                          // a=5 occurs at time 0
#3 b=7;                       // b=7 occurs at time 3
#5 c=4;                       // c=4 occurs at time 8
#8 d=10;                      // d=10 occurs at time 16
#2 e=9;                       // e=9 occurs at time 18
end
```

Block Statements

Example: Parallel Block

```
integer a=0, b=0, c=0, d=0, e=0;

initial
fork                          // Parallel block
a=5;                          // a=5 occurs at time 0
#3 b=7;                       // e=9 occurs at time 2
#5 c=4;                       // b=7 occurs at time 3
#8 d=10;                      // c=4 occurs at time 5
#2 e=9;                       // d=10 occurs at time 8
join
```

Nested Block Statements

Example1

```
integer a=0, b=0, c=0, d=0, e=0;

initial
begin              //sequential Block
a=5;
#5 b=7;
#6 c=4;
fork  #1 a=9;  #4 d=3;
#5 c=8; join   //Parallel Block
#8 d=10;
#2 e=9;
end
```

```
// a=5 occurs at time 0
// b=7 occurs at time 5
// c=4 occurs at time 11
// a=9 occurs at time 12
// d=3 occurs at time 15
// c=8 occurs at time 16
// d=10 occurs at time 24
// e=9 occurs at time 26
```

Nested Block Statements

Example2

```
integer a=0, b=0, c=0, d=0, e=0;

initial
fork                    //Parallel Block
a=2;
#6 b=7;
#3 c=4;
begin  #1 a=9;  #4 d=3;
#5 c=1; end    //Sequential Block
#9 d=10;
#2 e=9;
join
```

```
// a=2 occurs at time 0
// a=9 occurs at time 1
// e=9 occurs at time 2
// c=4 occurs at time 3
// d=3 occurs at time 5
// b=7 occurs at time 6
// d=10 occurs at time 9
// c=1 occurs at time 10
```

Named Block

- Verilog allows giving names to block statements. These block are called named blocks.
- Local variables can be declared for named blocks.
- Variables in a named block can be accessed by using hierarchical name referencing.
- It is possible to stop execution (disable) of named blocks from within the same block or different block.

Named Block

Declaration example:

```
initial
begin : block1
integer count; //count variable local to block1
 statements;
end

initial
begin : block2
reg a;              // a variable local to block1
statements;
end
```

Named Block

Example1:

```
initial
begin : block1
integer count; //count variable local to block1
count=0;
    forever
    begin
    #5 count = count + 1;
    if (count==8) disable block1;
    end
end
```

Named Block

Example2:

```
integer count=0, a=0, b=0;

initial
begin : block1
b=2;
     forever
     begin
     #1 count = count + 1;
     end
end
```

```
initial
begin
a=3;
#5 a=7;
#4 a=2;
#2 disable block1;
#3 a=9;
end
```

Named Block

Example3:

```
integer count=0, a=0, b=0;

initial
begin
b=2;
     forever
     begin : block1
     #1 count = count + 1;
     end
end
```

```
initial
begin
a=3;
#5 a=7;
#4 a=2;
#2 disable block1;
#3 a=9;
end
```

Named Block

Example4:

```
integer count=0, a=0, b=0;

initial
begin
b=2;
     begin: block1
     forever #1 count = count + 1;
     end
#2  b=4;
end
```

```
initial
begin
a=3;
#5 a=7;
#4 a=2;
#2 disable block1;
#3 a=9;
end
```

Named Block

Example5:

```
integer count=0, a=0, b=0;

initial
begin : block2
b=2;
     begin: block1
     forever #1 count = count + 1;
     end
end
```

```
initial
begin
a=3;
#5 a=7;
#4 a=2;
#2 disable block2.block1;
#3 a=9;
end
```

Continuous Assignment Delays

- Continuous assignment delays are used to delay assignment of net by specified time.
- There are three ways for assigning delay in Continuous assignment
  - Regular Assignment Delay
  - Implicit Continuous Assignment Delay
  - Net Declaration Delay
- Regular Assignment Delay

```
wire c;
assign #10 c = a & b;
```

Continuous Assignment Delays

- Implicit Continuous Assignment Delay

```
wire #10 c = a & b;
```

- Net Declaration Delay

```
wire #10 c;
assign c = a & b;
```

- All three ways of assigning delay gives same result.
- They these delays are inertial in nature and are used to model a digital circuit.

Procedural Timing Controls

- The time at which the Procedural Statements has to executed can be controlled with help of Procedural Timing control.
- There are three type of Procedural Timing Controls
  - Delay based Timing control
  - Event based timing control
  - Level Sensitive Timing control

Delay Based Timing Control

- There are three type of Delay based Timing Controls
  - Regular delay control
  - Intra assignment delay control
  - Zero Delay control

Regular Delay Control

- In Regular delay control, the execution of statement is delayed by a specified time.
- First delay is performed and then register is updated with the value at current time.
- Any event on the sensitivity list is ignored till the all the delays are executed.
- Both Blocking and Non-Blocking shows the similar behavior with regular delay.

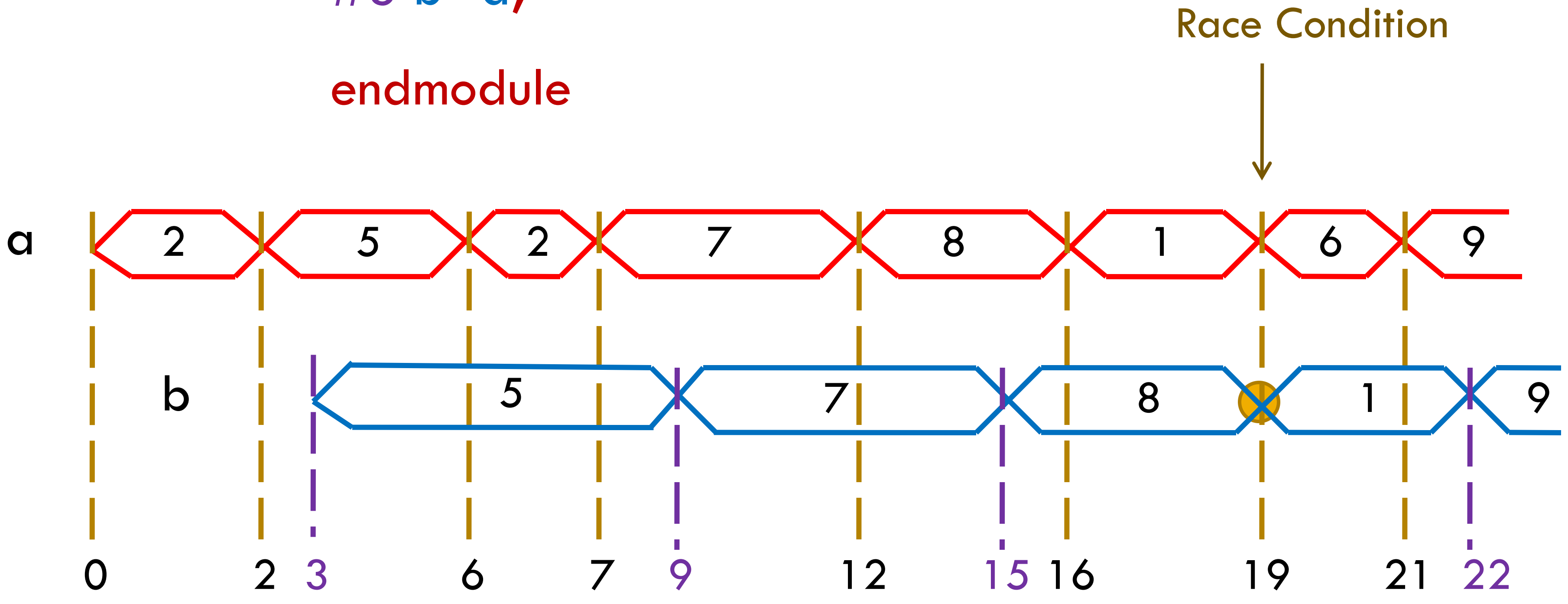
Regular Delay Control

Declaration example:

```
parameter m=5;
integer a=0, b=0, count=0;

initial
begin
a=0;
#10 a=5;
#m b=2;                        //Delay specified by parameter
#b count= count + 1;
#(4:5:8) b=4;                  //Min, Typ and Max delay
end
```

Regular Delay Control

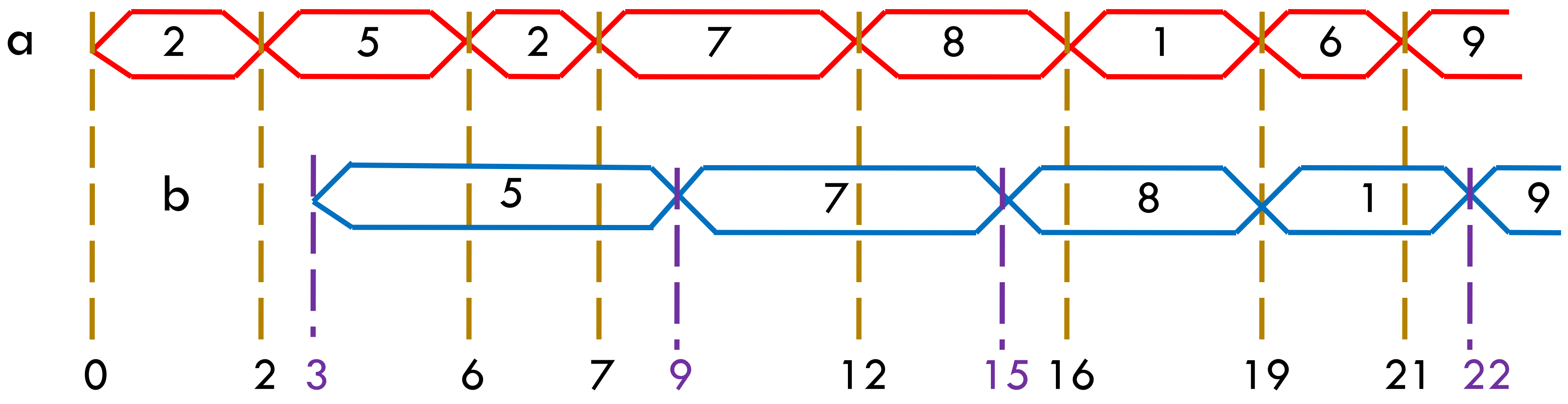
Example1:
```
module delay(input signed [31:0] a, output integer b);

always @ (a)
#3 b=a;

endmodule
```



Regular Delay Control

Example2:

```
module delay(input signed [31:0] a, output integer b);

always @ (a)
#3 b<=a;

endmodule
```



Intra Assignment Delay Control

- In Intra assignment delay control, the assignment of statement is delayed by a specified time.
- Current values are read and are assigned to the register after specified delay.
- Any event on the sensitivity list is ignored till the all the delays are executed for blocking statement.
- Blocking and Non-Blocking does not show same behavior with this type of delay.

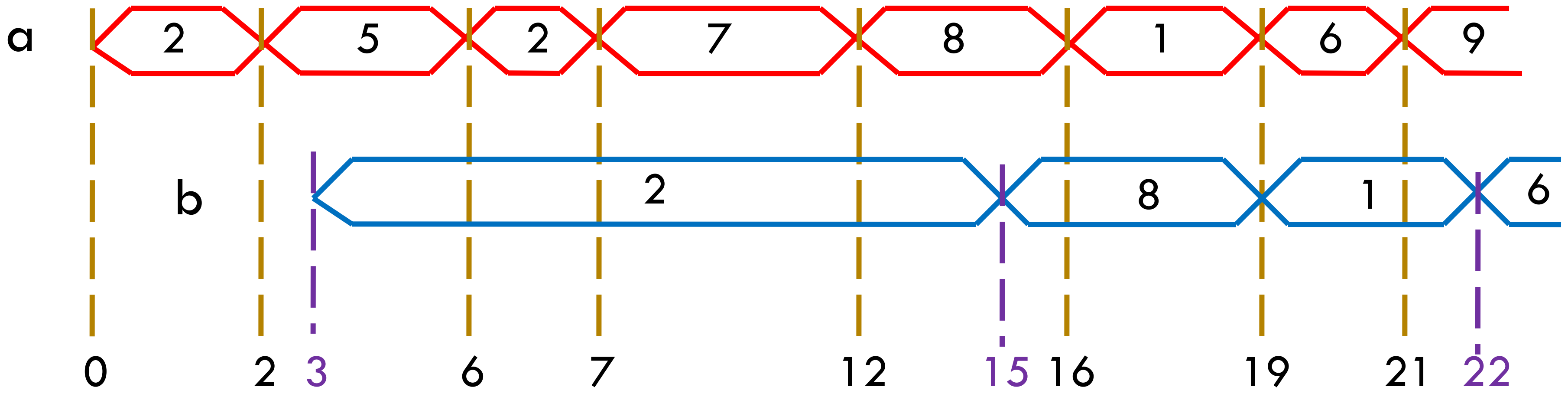
Intra Assignment Delay Control

Declaration example:

```
integer x=0, y=0;

initial
begin
x=2;
y=#4 x;                    //intra Assignment Delay
// For blocking assignment it is equivalent to
// temp=x;
// #4 y=temp;
end
```

Intra Assignment Delay Control

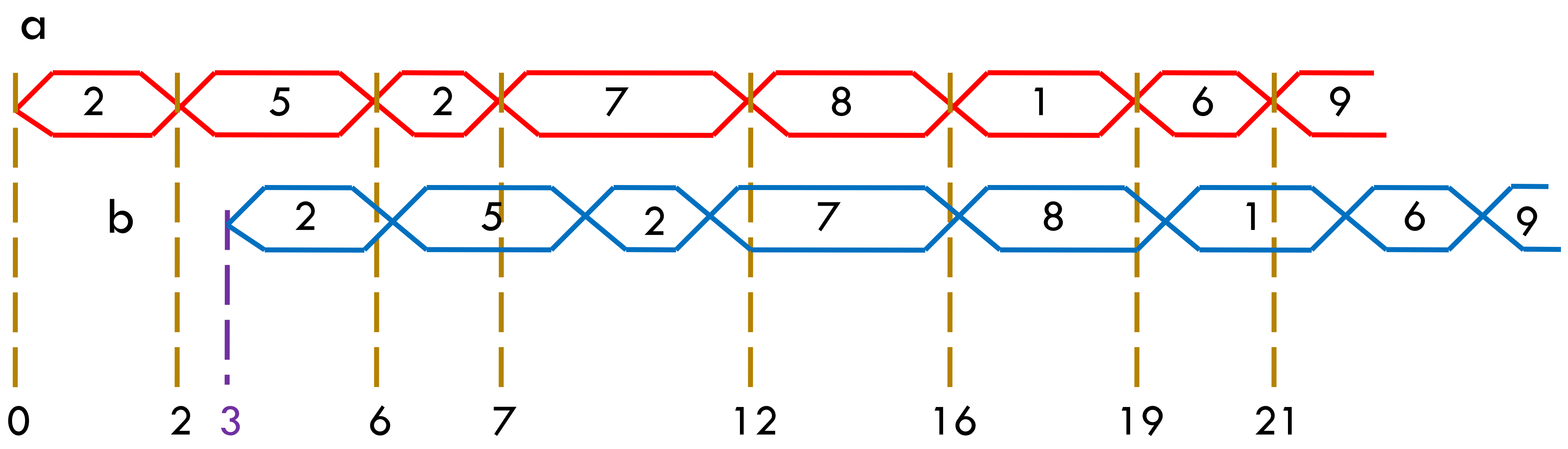
Example1:
```
module delay(input signed [31:0] a, output integer b);

always @ (a)
b= #3 a;

endmodule
```



Intra Assignment Delay Control

Example2:
```
module delay(input signed [31:0] a, output integer b);

always @ (a)
b<= #3 a;

endmodule
```

Transport Delay



Zero Delay Control

- Procedural statements in multiple always/initial block are evaluated at same time.
- In such a case the order of execution of the statement is non-deterministic.
- Zero delay is a method to ensure that a statement is executed at the last.
- If there are multiple zero delay statements then order between then is non-deterministic.

Zero Delay Control

Example:

```
integer x, y;

initial
begin
x=0; y=0;
end

initial
begin
#0 x=1; y=2;                    //#0 ensures this executes at last
end
```

Event Based Timing Control

- Change in value of a variable is called a event.
- Events can be used to trigger execution of statements or block of statements.
- There are three types of Event based Timing Control
  - Regular Event control
  - Named Event control
  - Event OR control

Regular Event Control

- @ is used to specify event control.
- Statements are executed if variable changes or posedge (positive transition) or negedge (negative transition) occur on a variable.

```
@ (clk)  q=d;               //executed whenever clk changes

@ (posedge clk) q=d; //executed whenever positive transition occurs on clk
                     //Delays execution till posedge occurs on clock
@ (negedge clk) q=d; //executed whenever negative transition occurs on clk

q= @(posedge clk) d; //value is read immediately and assigned after
                     //positive transition occur on clock
```

Named Event Control

- Named event allows user to declare event and trigger that event.

```
integer a=0, count=0;
event myevent;

initial
begin
a=4;
#3 ->myevent;
// -> is used to trigger event myevent
#2 count=6;
#4 ->myevent;
end
```

```
always @ (myevent)
begin
count= count + 1;
end
```

Event OR Control

- In may be a requirement that transition on one or more variable may trigger execution of statement or block of statements.
- This is expressed as OR of events also know as sensitivity list.
- Sensitivity list can also be specified with help of , instead of or operator

```
always @ (d or en)
begin
if (en) q=d;
end
```

```
always @ (posedge clk, posedge rst)
begin
if (rst) q<=0;
else q<=d;
end
```

Level sensitive Timing Control

- Level sensitive control waits for a condition to be true before a statement or block of statements is executed.
- wait keyword is used to provide level sensitive control.

```
integer count=0;

always
wait (Enable)
#5 count= count + 1;
```

- Enable is monitored continuously, if it is 0 then statement is not executed. If it is 1 then count is incremented by 1 after 5 time units.

Generate Blocks

- Generate statements are used to replicate hardware at elaboration time before simulation.
- All generate instantiations (hardware to be replicated) are written inside generate and endgenerate keywords.
- Generate instantiations can be combination of any of the following
  - Module instances
  - Gate Primitives
  - User defined Primitives
  - Continuous Assignments
  - initial and always blocks

Generate Blocks

- Verilog permits declaration of nets, registers and events in generate scope.

- Declaration of parameters, local parameters, input, output, inout port and specify block is not allowed.

- There are three types of generate statements
  - Generate Loop
  - Generate Conditional
  - Generate Case

Generate Loop

- genvar is the keyword used to declare variables uses only in evaluation of generate block.
- Value of genvar variable can only be modified by generate loop.
- Nesting of generate loop is allowed.
- Block identifiers are required for generate loop in most cases.

Generate Loop

Example1:

```
module fa_4 (output [3:0] sum, output carry,
                                        input [3:0] a, b, input c);
genvar i;                                  //variable for generate loop
wire [4:0] c_temp;
assign carry= c_temp[4];
assign c_temp[0]= c;
    generate for (i=0; i<=3; i=i +1)
        begin: loop1
        fa u (sum[i], c_temp[i+1], a[i], b[i], c_temp[i]);
        end
    endgenerate
endmodule
```

Generate Loop

Example2:

```
module and_4 (output reg [3:0] c, input [3:0] a, b);

genvar i;                              //variable for generate loop

   generate for (i=0; i<=3; i=i +1)
      begin : loop2
      always @ (a[i], b[i])
      c[i]= a[i] & b[i];
      end
   endgenerate

endmodule
```

Generate Loop

Example3:
```
module pipo (output reg [3:0] op , input [3:0] d, input clk);

genvar i;                          //variable for generate loop

        generate for (i=0; i<=3; i=i +1)
                begin : loop3

                always @ (posedge clk)
                op[i]<=d[i];
                end
        endgenerate
endmodule
```

Generate Loop

Example4:

```
module nested (output [3:0] op1, op2, op3 ,
                        input [3:0] a1, a2, a3 , b1, b2, b3 );

genvar i, k;

   wire [1:4] a [1:3];
   wire [1:4] b [1:3];
   wire [1:4] op [1:3];

   assign op1= op [1];          assign a[1] = a1;        assign b[1] = b1;
   assign op2= op [2];          assign a[2] = a2;        assign b[2] = b2;
   assign op3= op [3];          assign a[3] = a3;        assign b[3] = b3;
```

Generate Loop

```
generate
        for (i=1; i<=3; i=i +1)
          begin : loopi
            for (k=1; k<=4; k=k +1)
              begin : loopk
              assign op[i] [k]= a[i] [k] & b[i] [k] ;
              end
          end
    endgenerate

endmodule
```

Generate Conditional- Example 1

```
module logic_univ (output [3:0] op , input [3:0] a, b);
parameter logic_type="orgate";

     generate
          if (logic_type=="orgate")
               or u0 [3:0] (op, a, b);
          else if (logic_type=="andgate")
                and u1  [3:0] (op, a, b);
           else
                xor u2 [3:0] (op, a, b);
     endgenerate

endmodule
```

Generate Conditional-Example 2

```
module fa_4 (output [3:0] sum, output carry, input [3:0] a, b, input c);
genvar i;                          //variable for generate loop

wire [2:0] c_temp;
    generate for (i=0; i<=3; i=i +1)
        begin: loop1
        if (i==0)  fa u (sum[i], c_temp[i] , a[i], b[i], c);
        else  if (i>0 && i<3)
        fa u (sum[i], c_temp[i], a[i], b[i], c_temp[i-1]);
        else   fa u (sum[i], carry, a[i], b[i], c_temp[i-1]);
        end
    endgenerate
endmodule
```

Generate Case-Example

```
module adder (sum, carry, a, b, c);
parameter N=3;
output [N-1: 0] sum;  output carry;
input [N-1: 0] a, b; input c;

    generate
      case(N)
      1: adder_1bit u0 (sum, carry, a, b, c);
      2: adder_2bit u1 (sum, carry, a, b, c);
      default: adder_custom #(N) u2 (sum, carry, a, b, c);
      endcase
    endgenerate
endmodule
```