

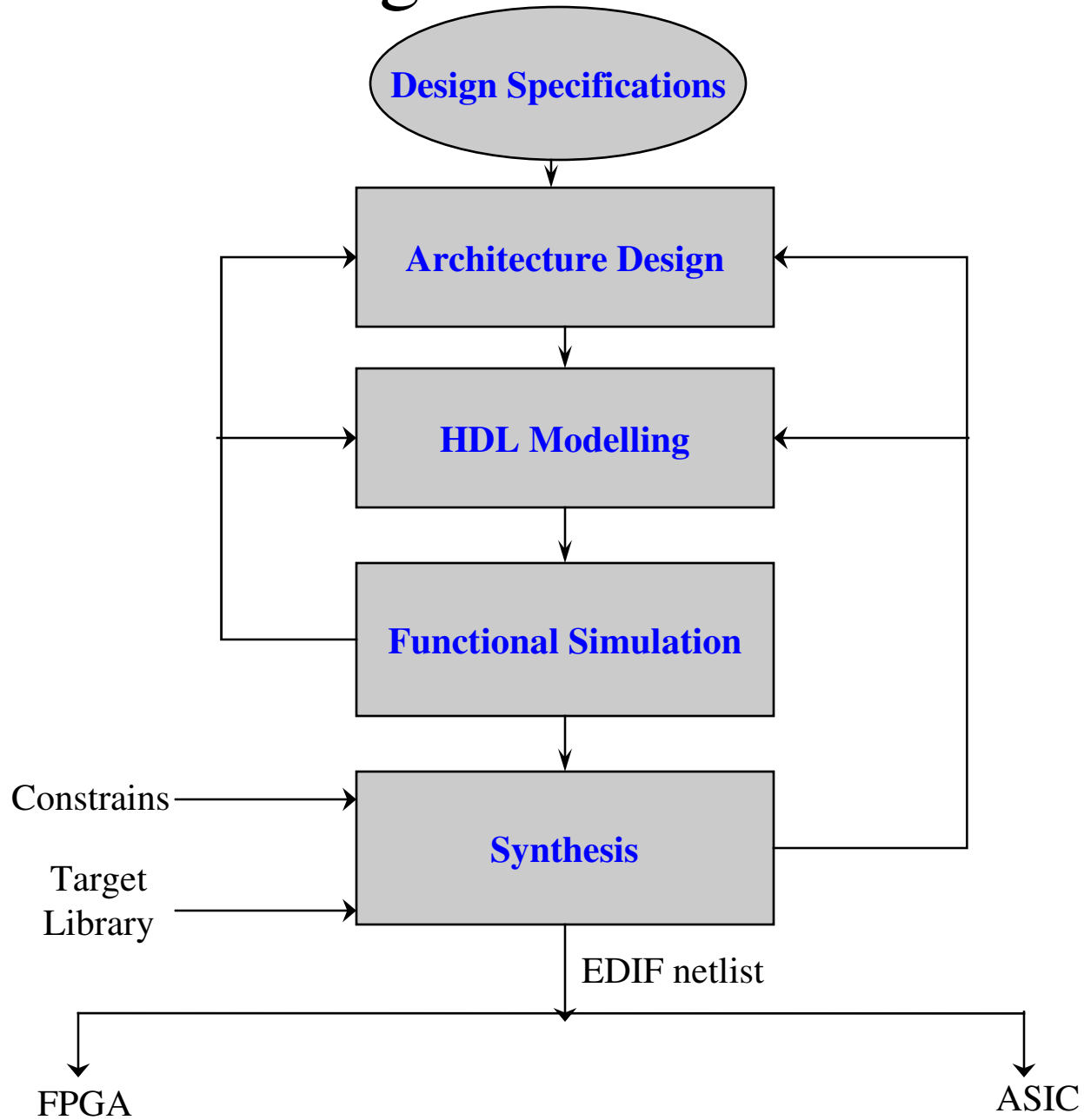
Verilog

Simulation & Synthesis

By
Vijay
Design Engineer



Design flow of an IC



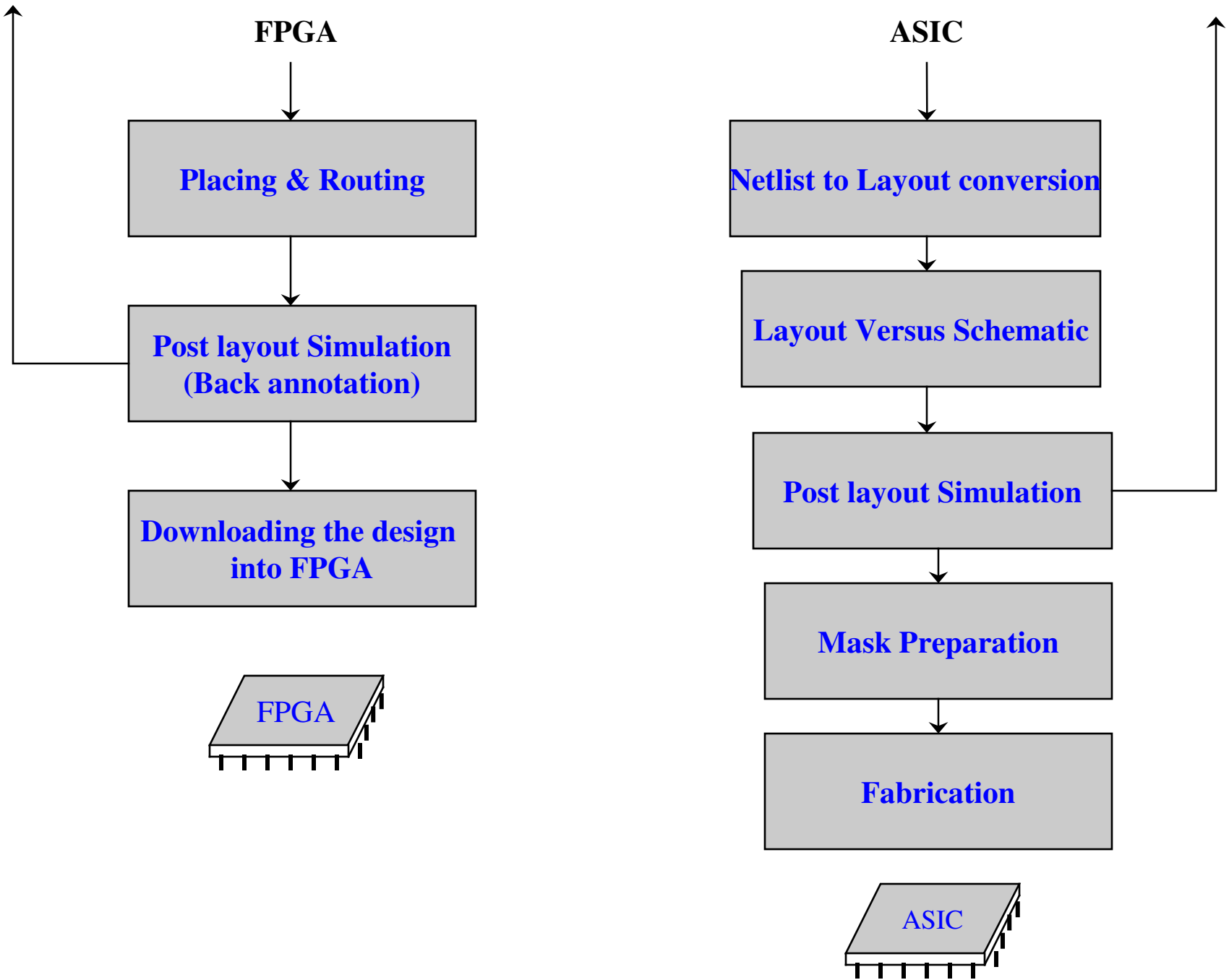
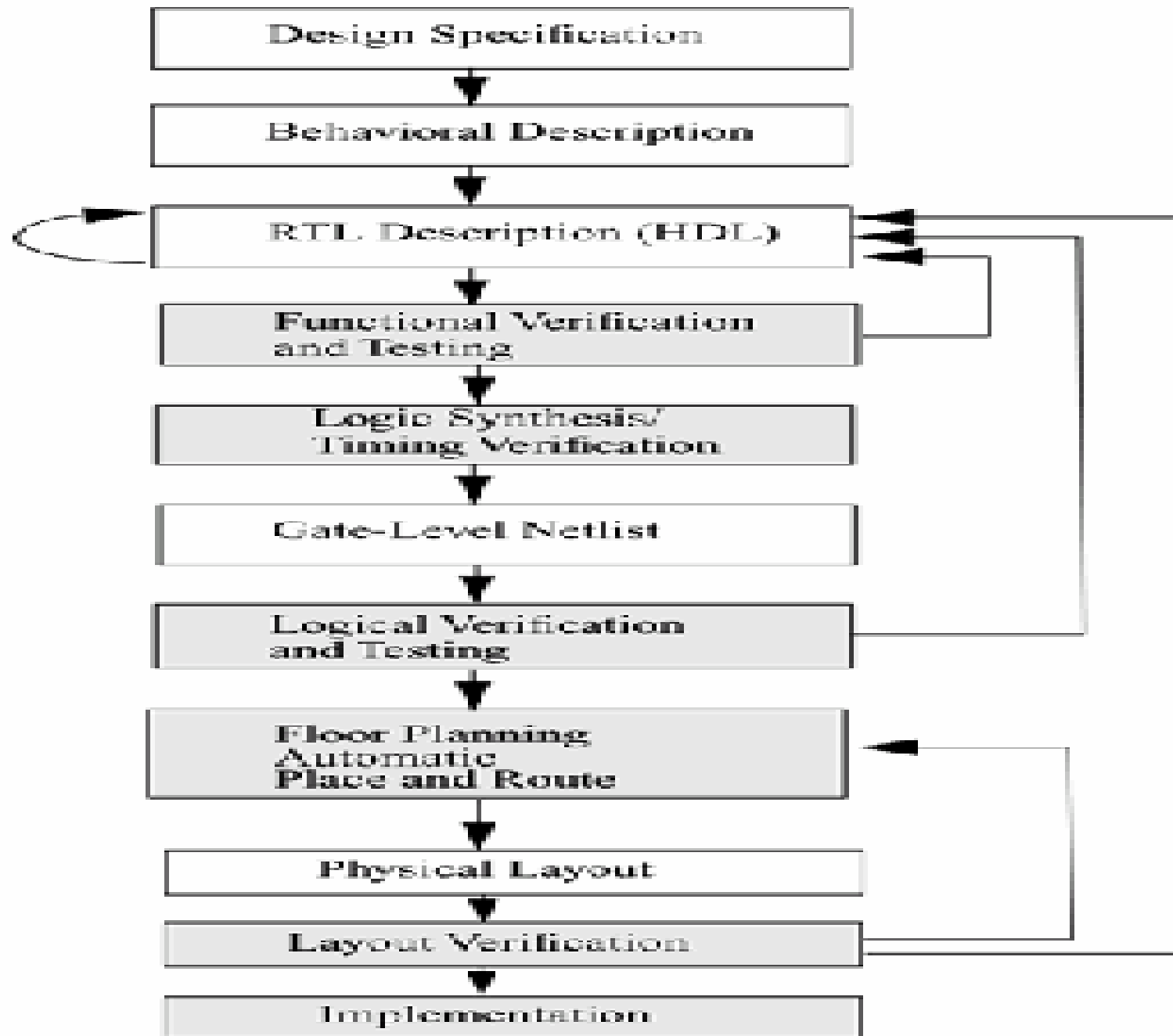


Figure 1-1. Typical Design Flow



Modern Design Methodology

Simulation and Synthesis are components of a design methodology

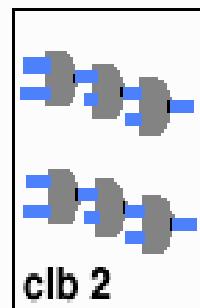
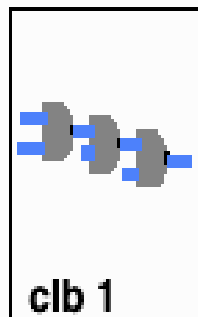
```
always
mumble
mumble
blah
blah
```

Synthesizable Verilog

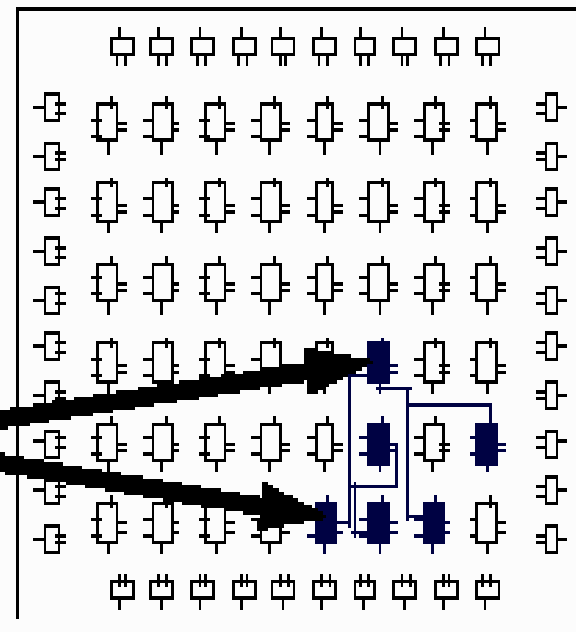
Synthesis



Technology Mapping



Place and Route



Importance of HDLs

- ◆ Designs can be described at a very abstract level by use of HDLs. Designers can write their RTL description without choosing a specific fabrication technology.

Brief History of Verilog

The verilog HDL was developed in 1984 by Gateway Design Automation and it was proprietary language

Later Verilog was placed in the public domain in 1990 by Cadence Design Systems

Became IEEE standard in 1995

And became the most widely used Hardware Description Language

Types of Modeling

- Behavioral
 - Structural
 - Gate level
 - Switch level
- RTL
-

Ports of a Module

Interfaces to the external world

```
module test(out,p,in_1,in_2);  
input in_1,in_2;  
output out;  
inout p;
```

Structure of Verilog Module

```
module test (port list);  
input ....; //Input declaration  
output ...; //Output declaration  
reg .....; } Intermediate Signal Declaration  
wire .....; }  
  
// Primitive or Module instantiation  
assign //Continuous assignments  
always @(sensitivity list) //Behavioral Block  
function //Description  
task // Description  
specify //Specify Block  
endmodule
```

Model of 'and' gate

```
module myand(a,o);  
input [1:0]a;  
output o;  
always @ (a)  
begin  
    o = a[1] & a[0];  
end  
endmodule
```

Number Specification

Sized Numbers :

4'b1010, 9'o451, 8'ha4
7'd16

Unsize Numbers

234, 'h 5ac, 'o17

Value Set

Any Verilog signal can have one of the below values

0	--	Logical Zero
1	--	Logical One
X	--	Unknown Value
Z	--	High Impedance

Data Declarations

<i>input</i>	}	Define direction and size of the port
<i>output</i>		
<i>inout</i>		
<i>net</i>		Used as interconnection between modules or components
<i>reg</i>		Used as internal signal or a memory
<i>parameter</i>		Are are the constants for the simulation
<i>integer</i>		Integer type
<i>real</i>		Floating point type
<i>event</i>		Flag that can trigger a behavioral block
<i>time</i>		Contains Delays and simulation time

Net Types

wire

Establishes connectivity

wand

A net with multiple drivers and resolution of 'wired and'

wor

A net with multiple drivers and resolution of 'wired or'

tri

Establishes connectivity

It has same functionality as wire.

Indicate that net will be tri stated in hardware

Net Types

tri0 A tri net with default value '0'

tri1 A tri net with default value '1'

trior A tri net uses resolution for non-z values using 'or'

triand

trireg A tri net with storage capability

supply0 *supply1* -- Used for switch level modelling

Differences between register and net

Operators

• Arithmetic

multiply(*) divide(/) add (+)
subtract(-) modulus (%)

• Logical

negation(!) and(&&) or(||)

• Relational

greater than (>) less than(<)
greater than or equal(>=)
less than or equal(<=)

Operators

Equality

equality(==)

inequality (!=)

case equality(===)

case inequality (!==)

Bitwise

negation(~) and (&) or (|) xor (^)

xnor (~^) nor (~|)

Operators

• Reduction

and(&) nand (~&) or (|) nor (~|)
xor (^) xnor (~^)

• Shift

right (>>) left (<<)

• Concatenation

{ , }

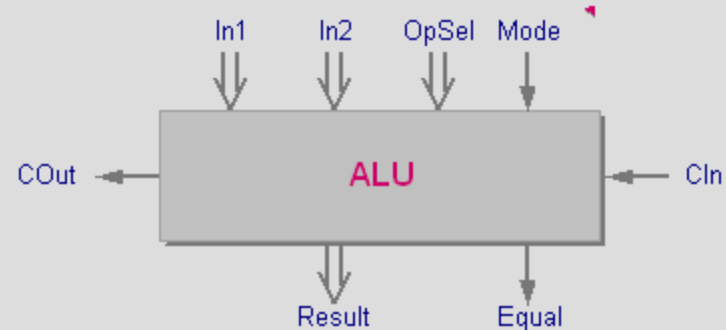
• Conditional ? :

eg: *assign* o = (en == 1'b1)? a : b ;

The Module Interface

◆ Port List

◆ Port Declaration



```
module ALU (Result, COut, Equal, In1, In2,
             OpSel, CIn, Mode);

output [3:0] Result;    // operation result
output     COut;       // carry out
output     Equal;     // when 1, In1 = In2
input  [3:0] In1;      // first operand
input  [3:0] In2;      // second operand
input  [3:0] OpSel;    // operation select
input  CIn;           // carry in
input  Mode;          // mode arithm/logic;
                        // arithm when 0
    . . .

endmodule
```

Let Us Solve

1. A register can be output of a module (T/F)
2. A net may be assigned in the behavioral block of the module (T/F)
3. A Verilog register is a hardware memory element (T/F)
4. The initial block will be executed only once (T/F)

Let Us Solve

5. Inputs to the module can be registers (T/F)
6. Default size of 'reg' is 32 – bits (T/F)
7. What are the differences between 'reg' and 'net'
8. Write a module for a 2:1 MUX using Conditional Operator (?).

Let Us Solve

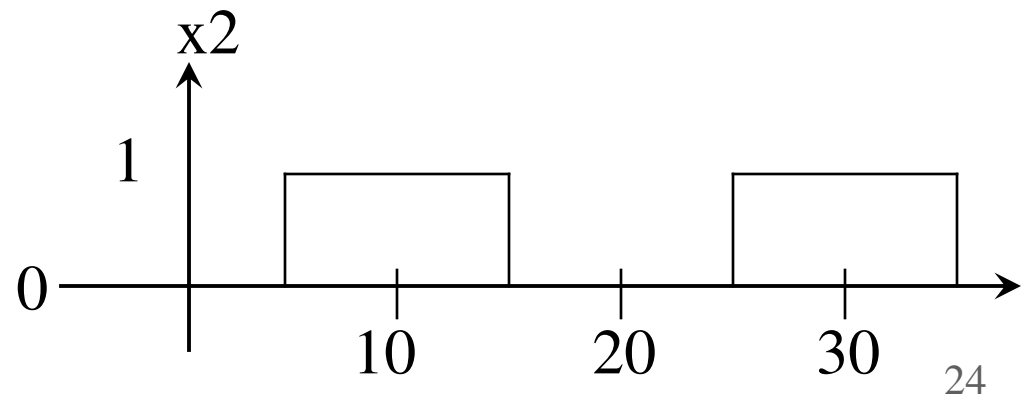
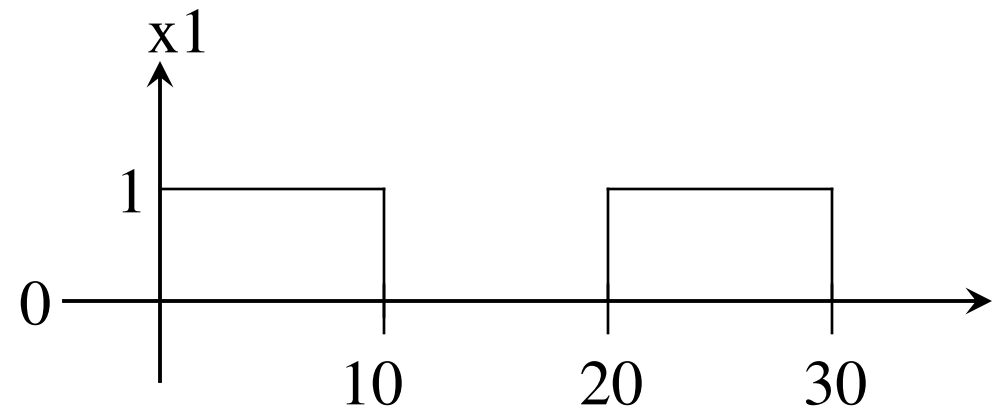
9. Find the error(s) in the code

```
module task(x1,x2,y);  
input [8:0] x1,x2;  
output [8:0] y;  
wire x1;  
net n1,n2;  
reg [-5:0] r;  
events e1,e2;  
assign y = x1 & x2;  
endmodule
```

Let Us Solve

10. Draw the waveform of 'y' when x1 and x2 are as shown below, for the following code.

```
module test (y,x1,x2);  
output y;  
input x1,x2;  
wand y;  
assign y = ~x1;  
assign y = ~x2;  
endmodule
```



Structural Modelling

```
module fulladd (a,b,cin,sum,cout);
```

Instantiation of the above module :

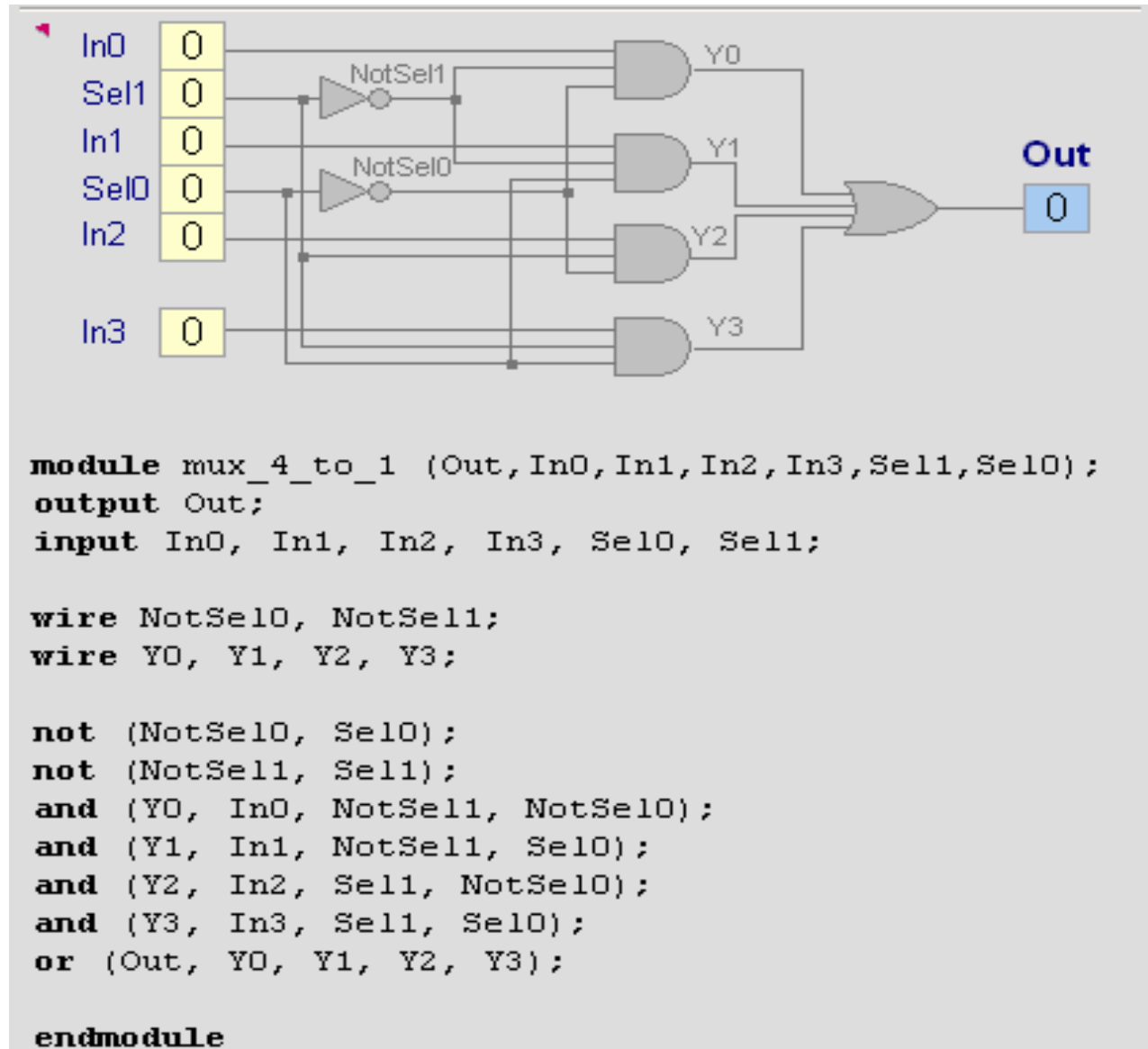
-- Connecting by Ordered List

```
    fulladd u1(in1,in2,in3,SUM,COUT);
```

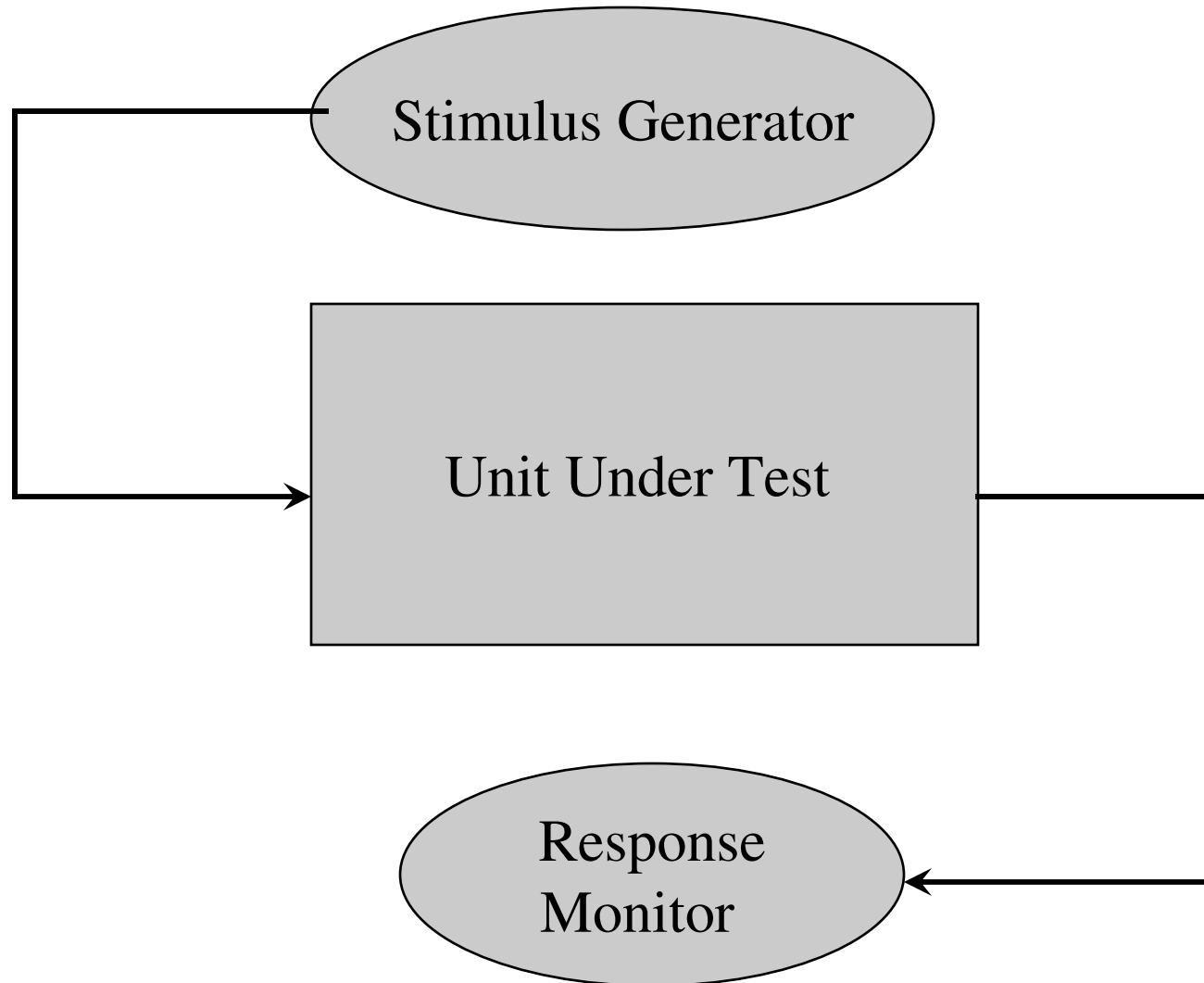
-- Connecting ports by name

```
    fulladd u1(.a(in1),.b(in2), .cin(in3),  
              .sum(SUM),.cout(COUT));
```

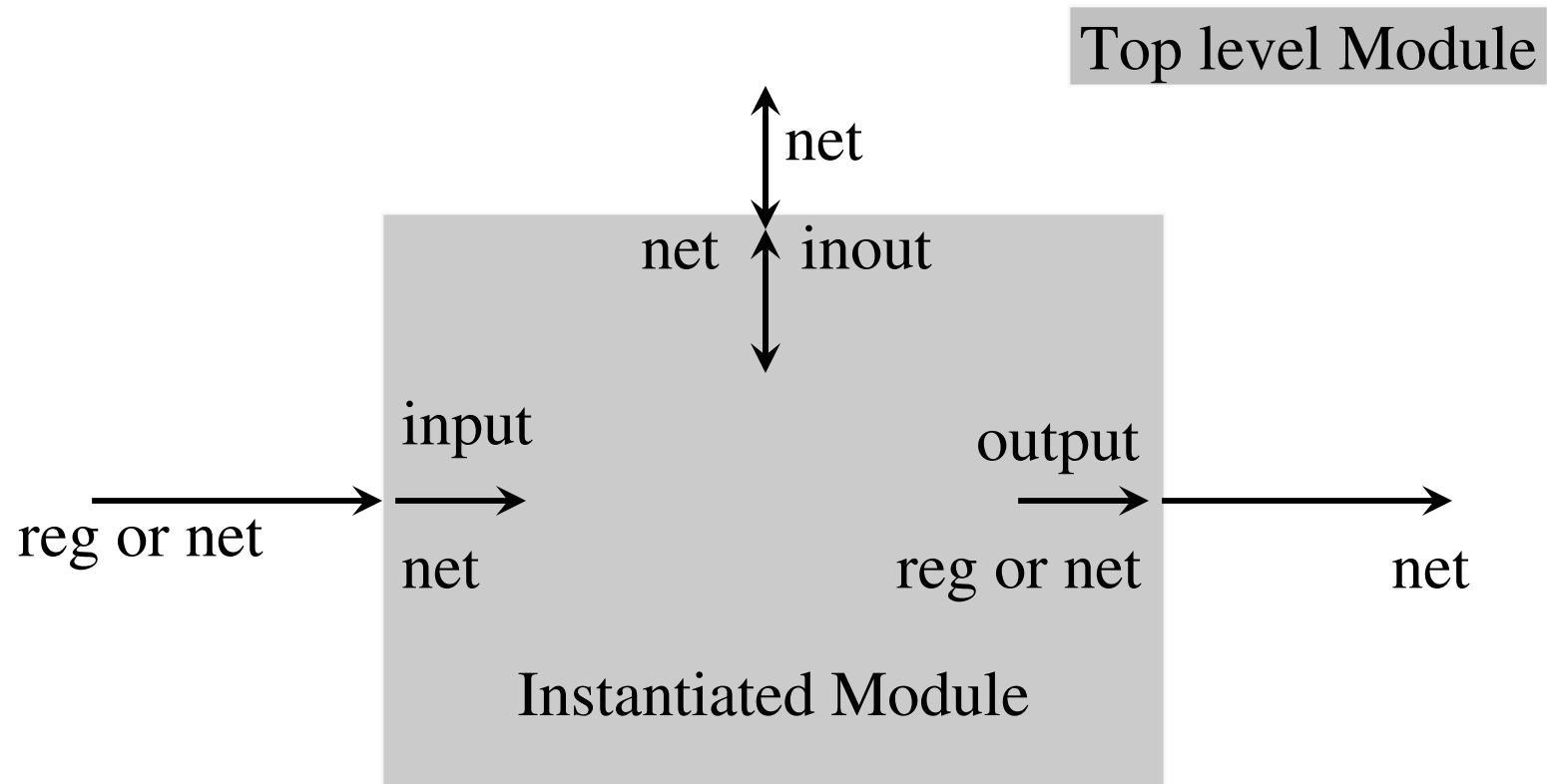
Structural style: Verilog Code



Stimulus Modelling



Instantiation Port Mapping Rules



Stimulus Modelling

```
module test;  
reg a1,b1;  
wire y1;  
  myand u1(y1,a1,b1);  
initial  
begin  
  a1 = 1'b0;  
  b1 = 1'b1;  
  #10 a1 = 1'b1;  
  #10 b1 = 1'b0;  
end  
endmodule
```

In built primitives

Instantiation label is not required

Two state :

and

nand

or

nor

xor

xnor

not

buf

Tri state :

bufif0

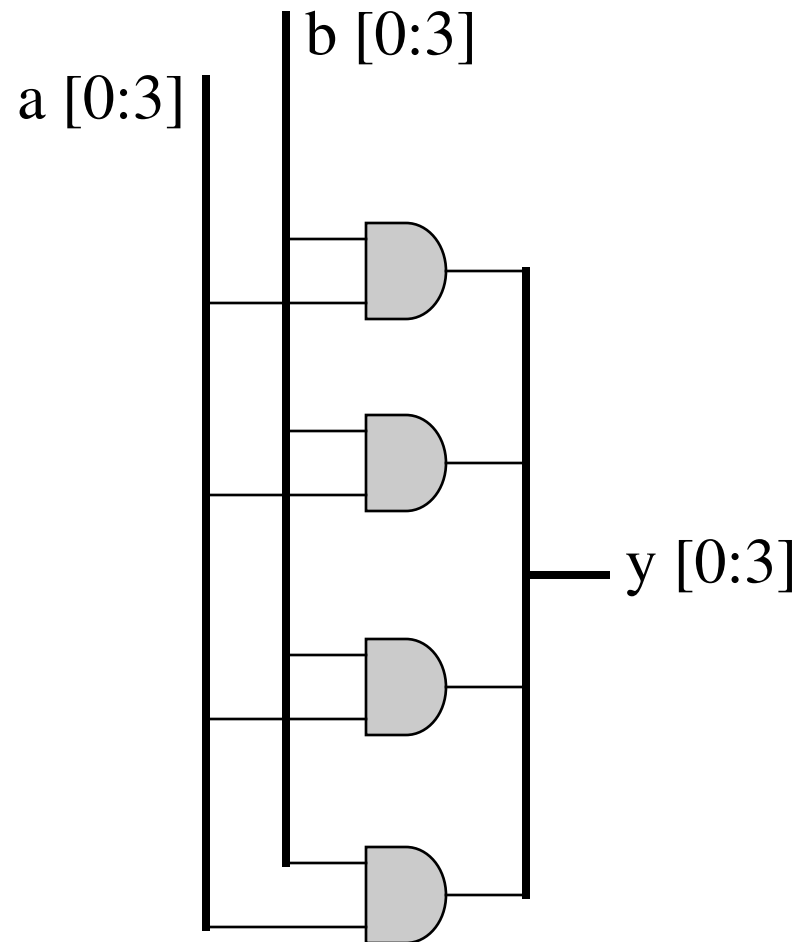
bufif1

notif0

notif1

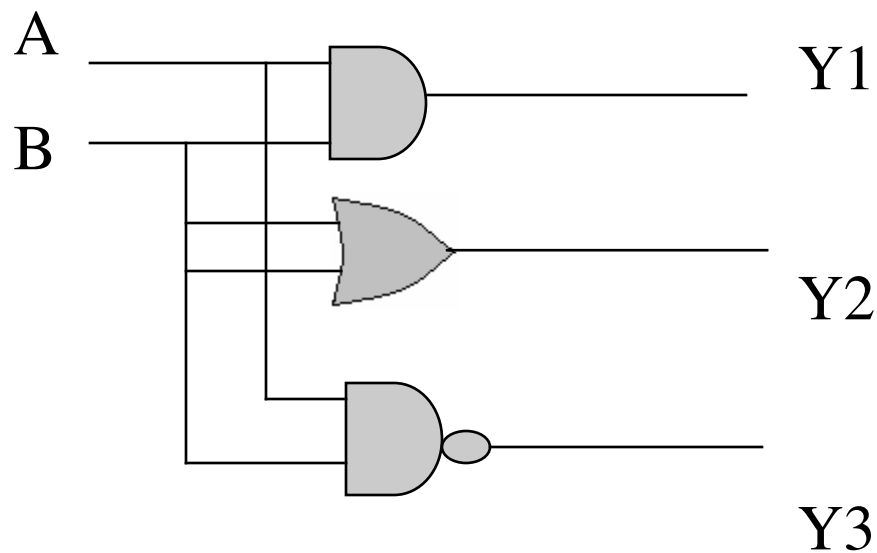
Array of Instances

```
module array (y,a,b);  
input [0:3] a,b;  
output [0:3] y;  
and [0:3] (y,a,b);  
endmodule
```



Let Us Solve

1. Write the Verilog structural model for the circuit given below using primitive gates.



Let Us Solve

2. Write the test bench for the above model.
3. Write the test bench for a 3-bit parallel adder.

Behavioral Modelling

All Behavioral statements should be in "always" or "initial" block.

Types of Assignment :

Blocking assignment (=)

Non blocking assignment (<=)

Blocking Assignments

- Completes the assignment before going to the next assignment. That is how it blocks the execution of other assignments in the same always block.
- Computing RHS and assigning it to LHS is single step operation.
- Race condition may Occur.
- Recommended for combinational logic modelling

Race Condition in Blocking statements

```
module fbosc1 (y1, y2, clk, rst);  
output y1, y2;  
input clk, rst;  
reg y1, y2;  
always @ (posedge clk or posedge rst)  
if(rst) y1 = 1'b0; // reset  
else y1 = y2;  
always @ (posedge clk or posedge rst)  
if(rst) y2 = 1'b1; // preset  
else y2 = y1;  
endmodule
```

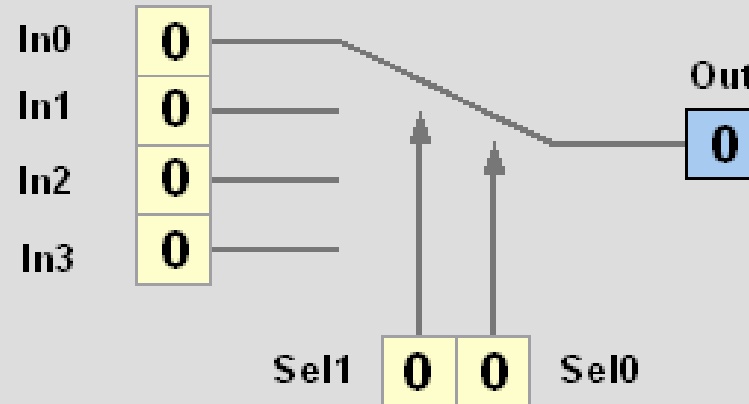
Non Blocking Assignments

- Goes to the next assignment before completing the current assignment. That is how it does not block the execution of other assignments in the same always block.
- Computing RHS and Assigning it to LHS is a two step operation.
- Race condition can be avoided.
- Recommended for sequential logic modelling

Module Using Non blocking Assignments

```
module fbosc1 (y1, y2, clk, rst);  
output y1, y2;  
input clk, rst;  
reg y1, y2;  
always @(posedge clk or posedge rst)  
if (rst) y1 <= 0; // reset  
else y1 <= y2;  
always @(posedge clk or posedge rst)  
if (rst) y2 <= 1; // preset  
else y2 <= y1;  
endmodule
```

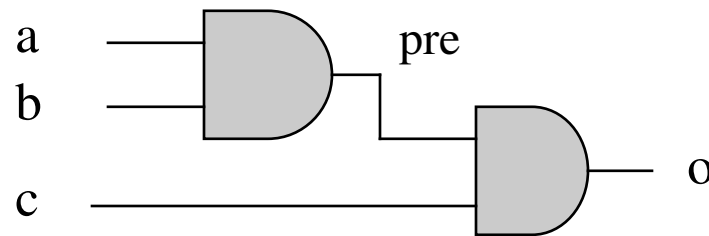
Behavioral style: Verilog Code



```
module mux_4_to_1 (Out, In0, In1, In2, In3, Sel1, Sel0);  
output Out;  
input In0, In1, In2, In3, Sel0, Sel1;  
reg Out;  
  
always @(Sel1 or Sel0 or In0 or In1 or In2 or In3)  
  begin  
    case ({Sel1, Sel0})  
      2'b00 : Out = In0;  
      2'b01 : Out = In1;  
      2'b10 : Out = In2;  
      2'b11 : Out = In3;  
      default : Out = 1'bx;  
    endcase  
  end  
  
endmodule
```

What if non blocking assignment is used for combinational logic ?

```
module and2 (o, a, b, c);  
input a, b, c;  
output o;  
reg pre, o;  
always @ (a or b or c)  
begin  
pre <= a & b;  
o <= pre & c;  
end  
endmodule
```



Let Us Solve

1. Draw the waveform of 'clk' if the code is

a) `begin`

`clk = 0;`

`#5 clk = 1;`

`#7 clk = 0;`

`#10 clk = 1;`

`end`

b) `begin`

`clk <= 0;`

`#5 clk <= 1;`

`#7 clk <= 0;`

`#10 clk <= 1;`

`end`

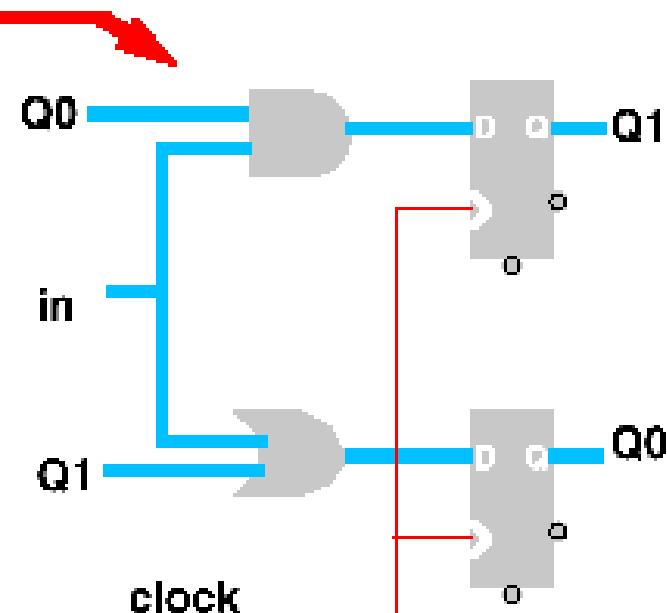
They are same

Non-Blocking Concurrent Assignment

Concurrent Assignment — primary use of `<=`

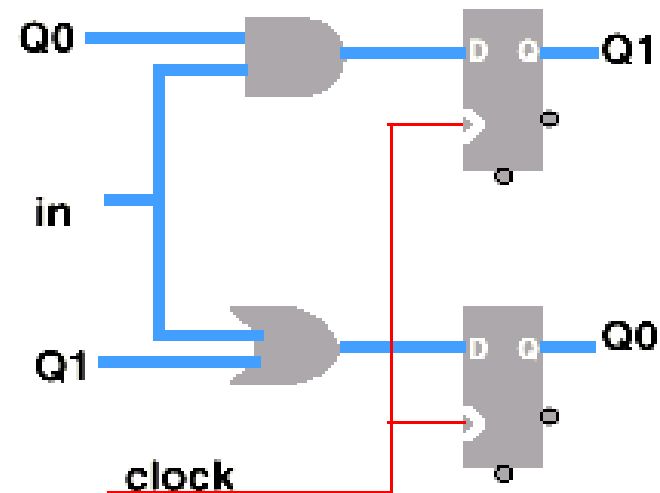
- The assignment is “guarded” by an edge
- All assignments guarded by the edge happen concurrently
 - All right-hand sides are evaluated before any left-hand sides are updated
 - Like this

```
module fsm (Q1, Q0, in, clock);  
  output  Q1, Q0;  
  input   clock, in;  
  reg     Q1, Q0;  
  
  always @(posedge clock) begin  
    Q1 <= in & Q0;  
    Q0 <= in | Q1;  
  end  
endmodule
```



Edges in time – concurrent assignment

```
module fsm (Q1, Q0, in, clock);  
  output  Q1, Q0;  
  input   clock, in;  
  reg     Q1, Q0;  
  
  always @(posedge clock) begin  
    Q1 <= in & Q0;  
    Q0 <= in | Q1;  
  end  
endmodule
```



Values after the clock edge (t^+)

— calculated in response to the clock edge, using values at the clock edge

Values at the clock edge. (At t)

Alternates – not all equivalent

```
module fsm (Q1, Q0, in, clock);
  output  Q1, Q0;
  input   clock, in;
  reg     Q1, Q0;

  always @(posedge clock) begin
    Q1 <= in & Q0;
    Q0 <= in | Q1;
  end
endmodule
```

```
module fsm (Q1, Q0, in, clock);
  output  Q1, Q0;
  input   clock, in;
  reg     Q1, Q0;

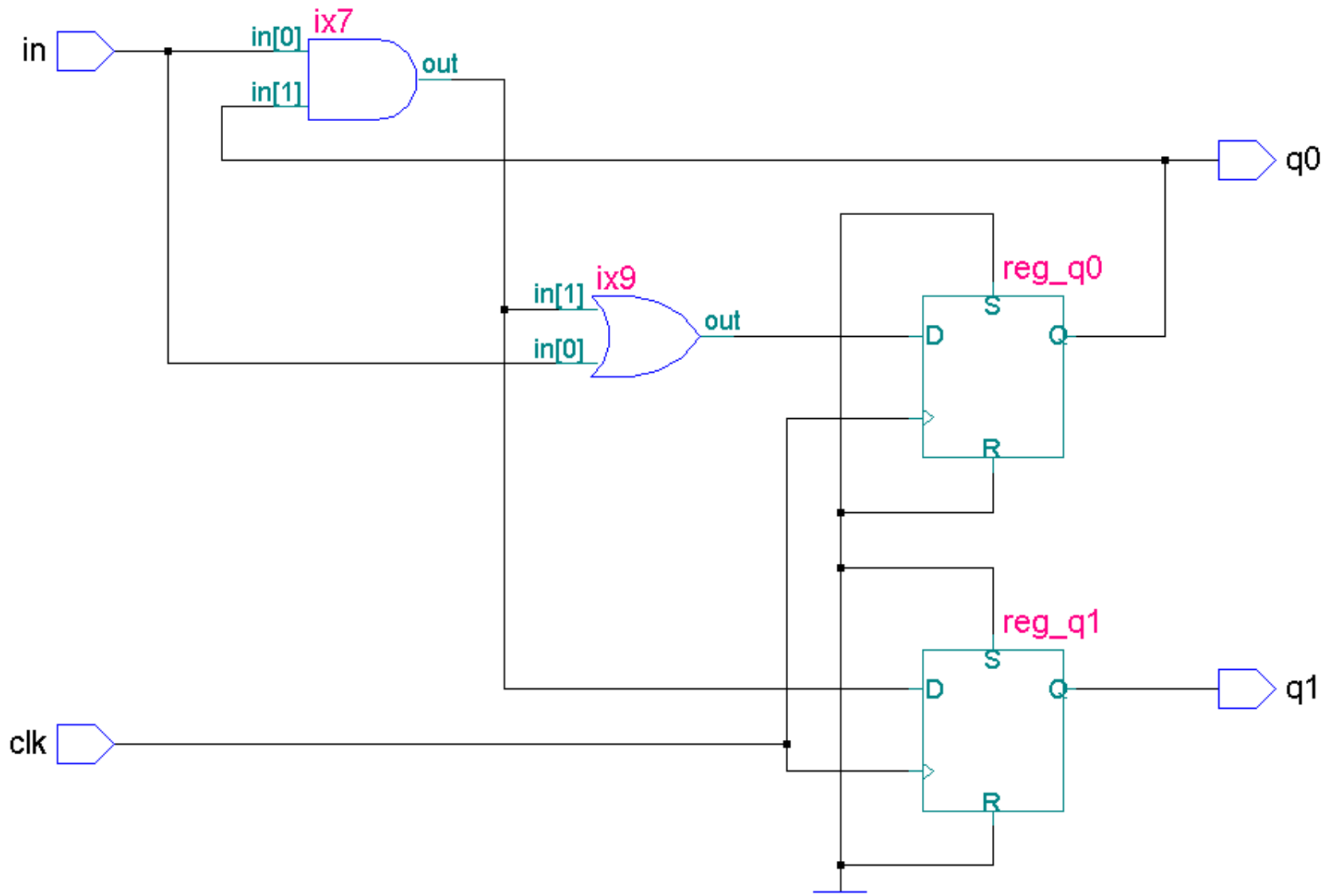
  always @(posedge clock) begin
    Q1 = in & Q0;
    Q0 = in | Q1;
  end
endmodule
```

```
module fsm (Q1, Q0, in, clock);
  output  Q1, Q0;
  input   clock, in;
  reg     Q1, Q0;

  always @(posedge clock) begin
    Q0 <= in | Q1;
    Q1 <= in & Q0;
  end
endmodule
```

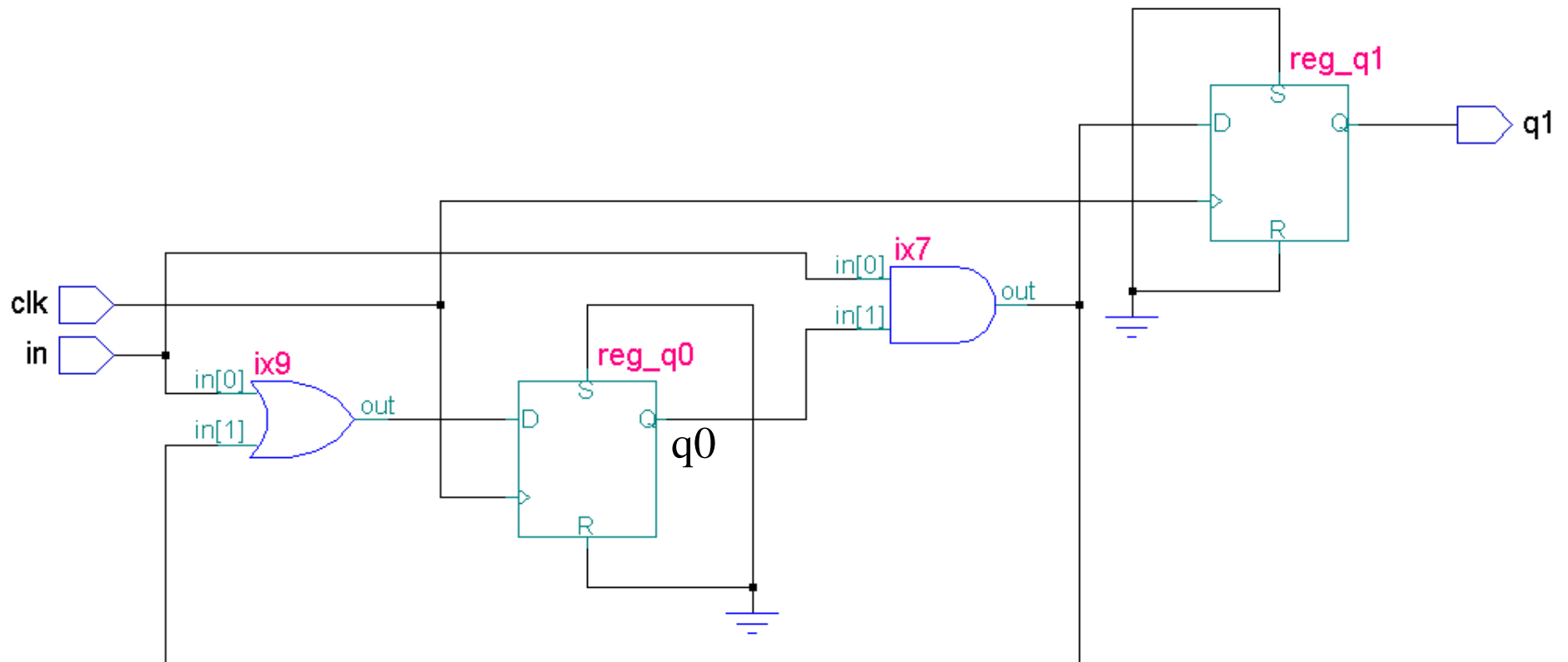
A very different animal?

The same?



If this is the code

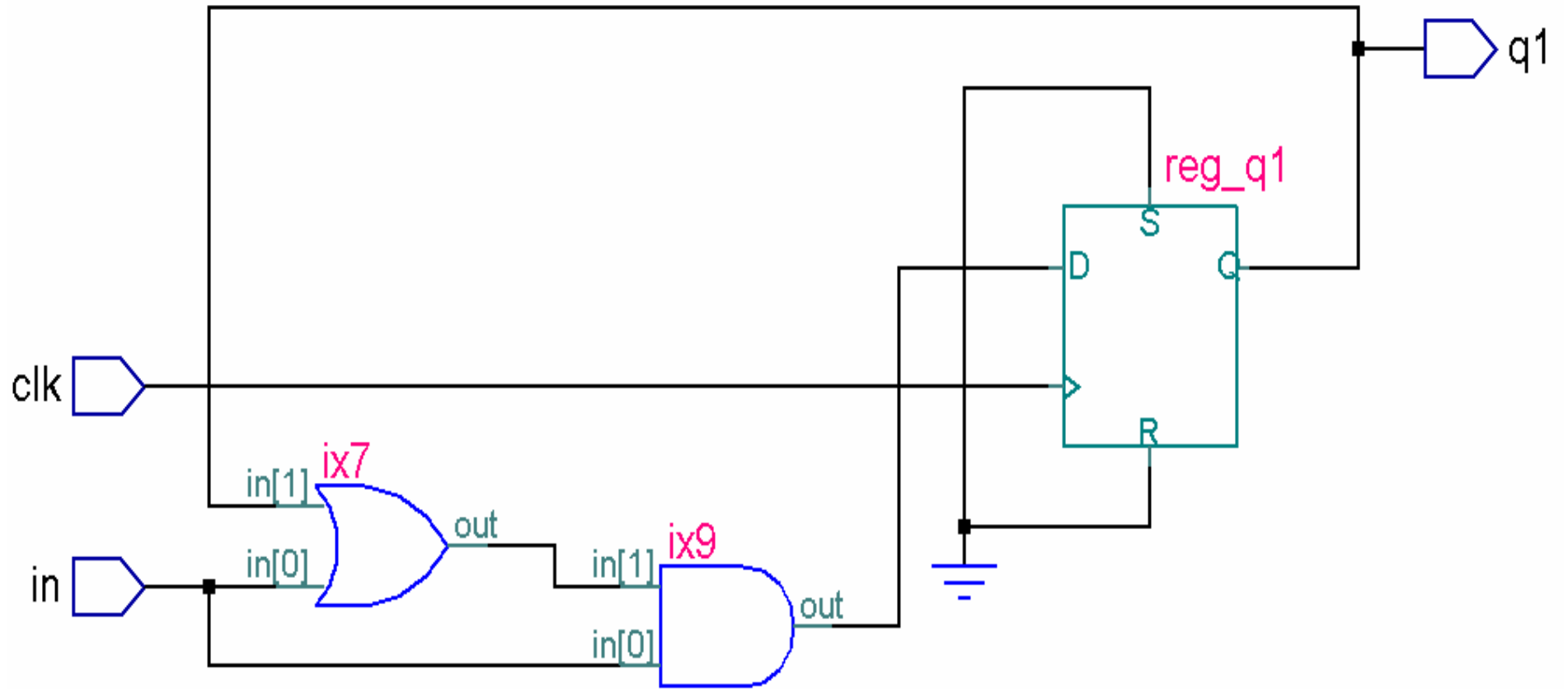
```
module ft (q1, in, clk);  
input clk, in;  
output q1;  
reg q1, q0; // Here q0 is an intermediate signal not an output  
always @ (posedge clk)  
begin  
q1 = in & q0; // output is computed with previous q0  
q0 = in | q1; // q0 is computed for next iteration  
end  
endmodule
```



No difference in hardware

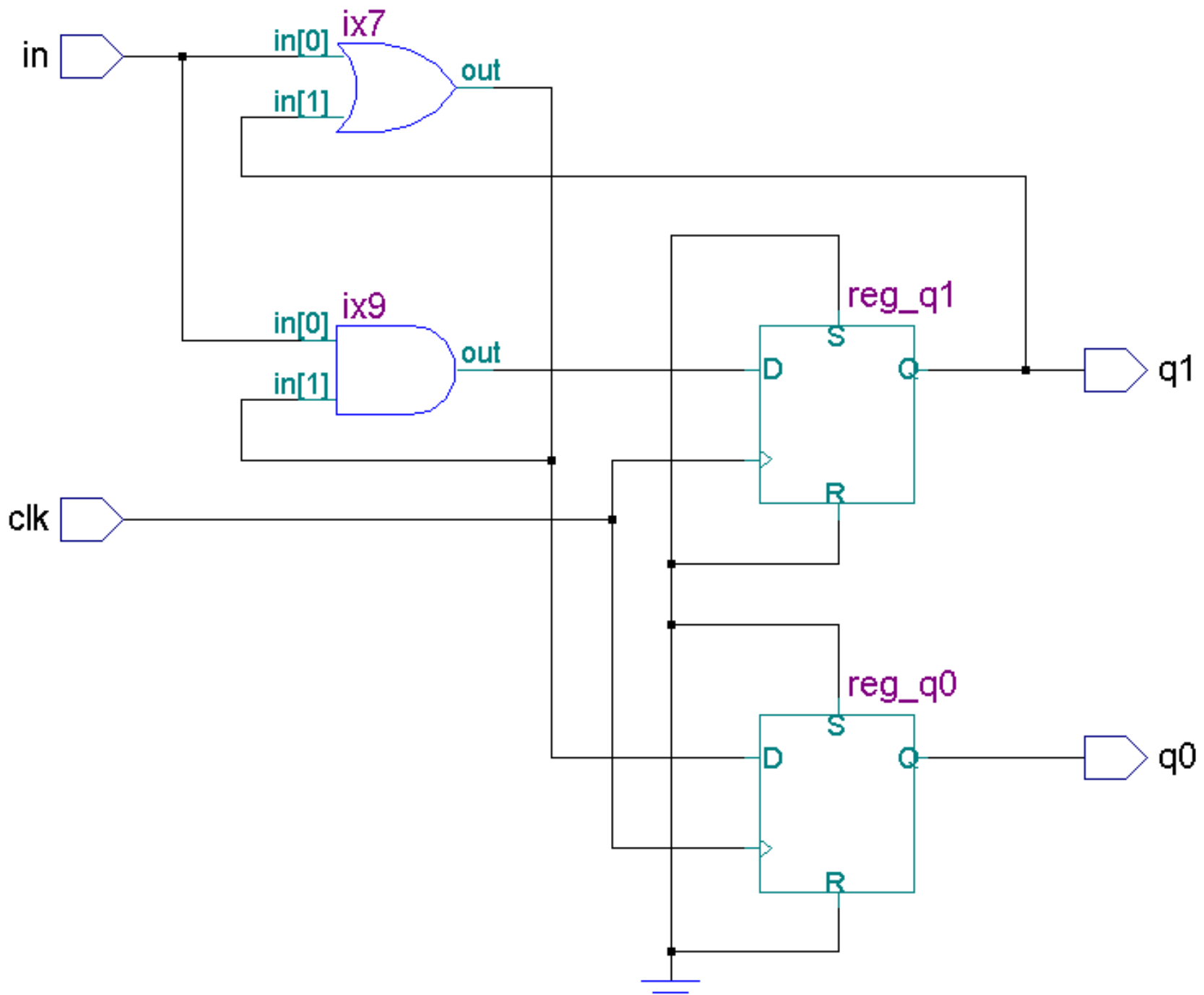
If this is the code

```
module ft (q1, in, clk);  
input clk, in;  
output q1;  
reg q1, q0;  
always @ (posedge clk)  
begin  
q0 = in | q1;  
q1 = in & q0; //output assignment is the second assign.  
end  
endmodule
```



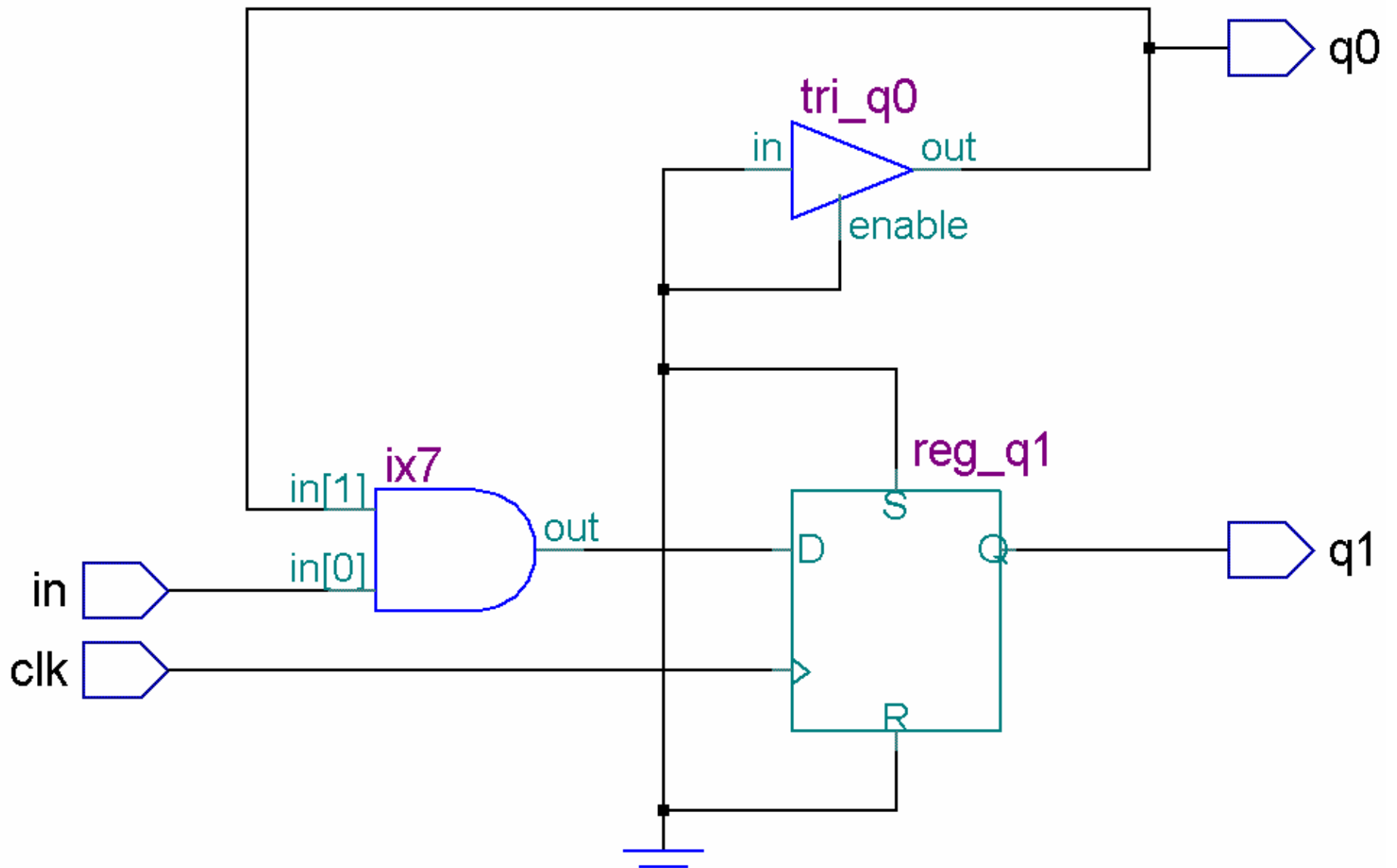
If this is the code

```
module ft (q0, q1, in, clk);  
input clk, in;  
output q1, q0;  
reg q1, q0;  
always @ (posedge clk)  
begin  
q0 = in | q1;  
q1 = in & q0;  
end  
endmodule
```



If this is the code

```
module ft (q0, q1, in, clk);  
input clk, in;  
output q1, q0;  
reg q1, q0;  
always @ (posedge clk)  
begin  
q1 = in & q0;  
end  
endmodule
```



How about these?

```
module fsm1 (Q1, Q0, in, clock);
    output    Q1;
    input     clock, in, Q0;
    reg       Q1;

    always @(posedge clock) begin
        Q1 <= in & Q0;
    end
endmodule
```

```
module fsm0 (Q1, Q0, in, clock);
    output    Q0;
    input     clock, in, Q1;
    reg       Q0;

    always @(posedge clock) begin
        Q0 <= in | Q1;
    end
endmodule
```

Will these work?

```
module fsm1 (Q1, Q0, in, clock);
    output    Q1;
    input     clock, in, Q0;
    reg       Q1;

    always @(posedge clock) begin
        Q1 = in & Q0;
    end
endmodule
```

```
module fsm0 (Q1, Q0, in, clock);
    output    Q0;
    input     clock, in, Q1;
    reg       Q0;

    always @(posedge clock) begin
        Q0 = in | Q1;
    end
endmodule
```

These?


Behavioral Blocks

 **initial** -- This block executes exactly once during the simulation.

Used in writing test benches and initializing the registers.

Multiple "initial" blocks are allowed.

This block is not synthesizable

 **always** -- This block is used to model continuously repeated activity in digital circuits.

Used to model the core logic

Multiple "always" blocks are allowed.

System Tasks

- *\$display* Displays at that simulation time

\$display ("cin = %b",cin);

- *\$monitor* Displays when there is a change in the value

\$monitor ("cin = %b",cin);

Simulation control system tasks

- *\$stop* Suspends the simulation

\$stop(1) – Prints the simulation time and instance

\$stop(2) – Prints the simulation time,instance and

CPU Utilization

- *\$finish* Terminates the simulation and control goes to the host operating system

Timing Controls

i.Delay Based Timing Control

ii.Event Based Timing Control

Delay Based Timing Control

- Regular delay control

#10 out = a & b;

- Intra Assignment Delay Control

out = #10 a & b;

- Zero Delay Control

#0 out = a & b;

Event Based Timing Control

Regular Event Control

always @ (a)

Named Event Control

event e1;

always @ (enable)

begin

if (flag)

-> e1;

end

always @ (e1)

begin

Wait Statement

Can be used to synchronize two blocks. This construct suspends the execution if the expression is False and start the execution when it becomes True.

Example: *always* @(a or b)

begin

.....

.....

wait(flag = 1'b1);

.....

end //Difference between 'if' and 'wait'

Conditional Statements

if (condition)

begin

Set of statements to be executed

end

else

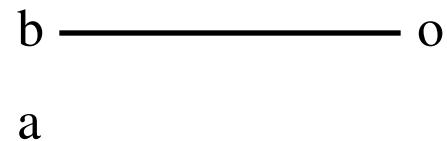
begin

Set of statements to be executed

end

what if you assign 'x' – don't care

```
module test (a,b,o);  
input a,b;  
output o;  
always @ ( a or b)  
if (a == 1'b1)  
o = b;  
else  
o = 1'bx;  
endmodule
```



Multiway Branching

If the condition has more than two possibilities
case statement can be used

case is used to implement ‘multiplexer’ or
‘decoder’

casez --Treats all 'z' values as don't cares

casex --Treats all 'x' and 'z' values as don't
cares

Priority encoder

```
case (a)
```

```
  4'b1xxx : o = 2'b00;
```

```
  4'b01xx : o = 2'b01;
```

```
  4'b001x : o = 2'b10;
```

```
  4'b0001 : o = 2'b11;
```

```
  default : o = 2'b00;
```

```
endcase
```

Loops

for loop

for (i = 0 ; i < 10 ; i = i + 1)

begin

set of statements to be executed

end

repeat (no.of iterations)

begin

set of statements to be executed

end

Loops

while (expression)

begin

set of statements to be executed

end//repeats until the expression is false

forever

begin

set of statements to be executed

end//unconditional repetitive execution of
//statements

Fork - Join Block

All statements in this block will execute concurrently

always @(in1 or in2)

begin

//Behavioral Statements to be executed

 //Sequentially

fork

begin

//Behavioral Statements to be executed

 //Concurrently

end

join

end

.....

fork

```
# 50 sig = 1'b1;
```

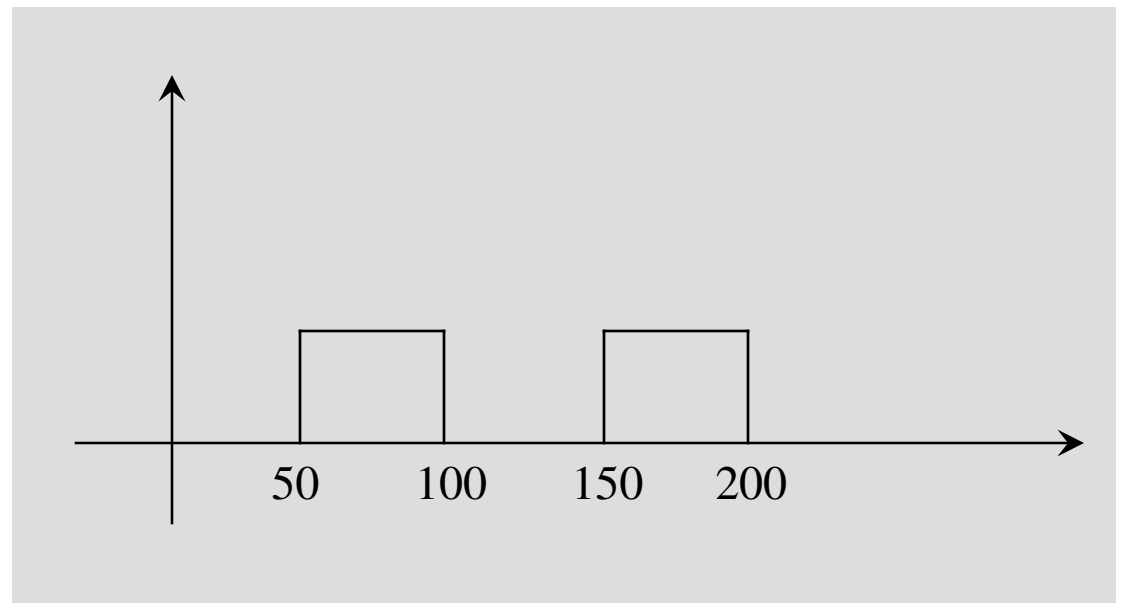
```
#100 sig = 1'b0;
```

```
#150 sig = 1'b1;
```

```
#200 sig = 1'b0;
```

join

....



Subprograms

Tasks and Functions

Frequently used functionalities are written as Tasks or Functions

Tasks: *task* <task name>

//input declaration

//output declaration

//parameter declaration

//register declaration

begin

//Behavioral statements

end

endtask

Subprograms

Tasks and Functions

Functions:*function* [length:1] <function name>
 //input declaration
 //parameter declaration
 //register declaration
 begin
 //Behavioral statements
 end
endfunction

Subprograms

Tasks and Functions

These should be called in behavioral blocks

Tasks:

- i. *always* @(in)
 bitwise(out,in);
- ii. Can call other tasks and functions
- iii. May have one or more than one outputs
- vi. May contain delays

Functions:

- i. *always* @(in)
 out = bitwise(in);
- ii. Can call other functions but not tasks
- iii. Returns a single value, outputs are not allowed
- vi. should not contain delays