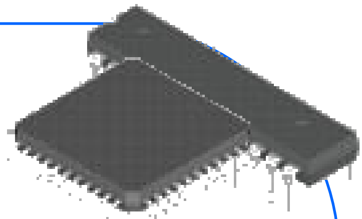


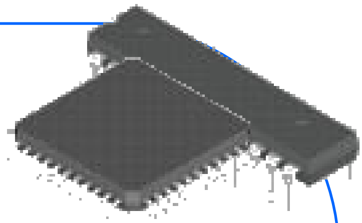
Hardware Description Language (HDL)

The first step of the Design Flow

D. Mukhopadhyay

Dept of Computer Sc and Engg,

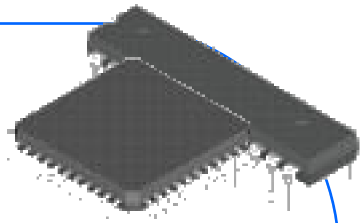
debdeep@cse.iitkgp.ernet.in



How it started!

- Gateway Design Automation
- Cadence purchased Gateway in 1989.
- Verilog was placed in the public domain.
- Open Verilog International (OVI) was created to develop the Verilog Language as IEEE standard.

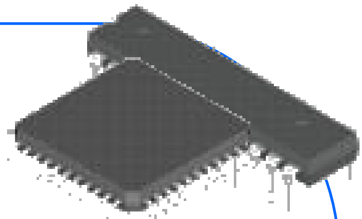
The Verilog Language



- Originally a modeling language for a very efficient event-driven digital logic simulator
- Later pushed into use as a specification language for logic synthesis
- Now, one of the two most commonly-used languages in digital hardware design (VHDL is the other)
- Virtually every chip (FPGA, ASIC, etc.) is designed in part using one of these two languages

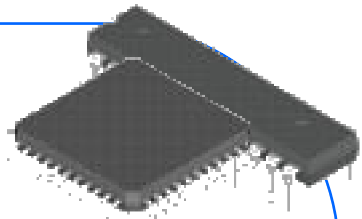
- Combines structural and behavioral modeling styles

Concurrency



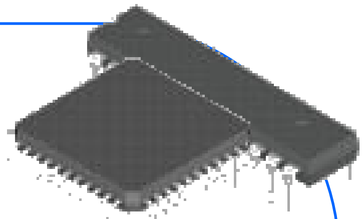
- Verilog or any HDL has to have the power to model concurrency which is natural to a piece of hardware.
- There may be pieces of two hardware which are even independent of each other.
- Verilog gives the following constructs for concurrency:
  - always
  - assign
  - module instantiation
  - non-blocking assignments inside a sequential block

However…



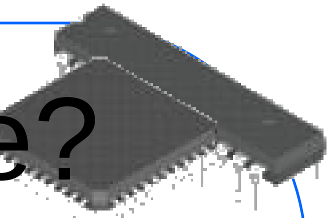
- The simulating machine is sequential.
- no-matter how many processors I have, I can write one more process which also have to be simulated concurrently.
- So, the processes are scheduled sequentially so that we have a feeling of parallelism:
  - like the desktop of our PC where we may open an editor, a netscape and so on. We have a feeling that they are executed parallely, but in reality there is a serialization involved.

Race…



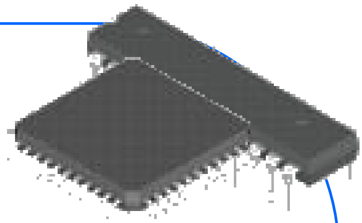
- So, always blocks (for example) are parallel…
- Even though they are not actually.
- So, which always block gets executed first? My simulation results will depend upon that!
- Standards do not say anything.
- It may be that the last always block is executed first…
- and so we have race!

How do we overcome race?



- We shall see in the class…
- Try to think of a clean hardware and you will not have race

Multiplexer Built From Primitives

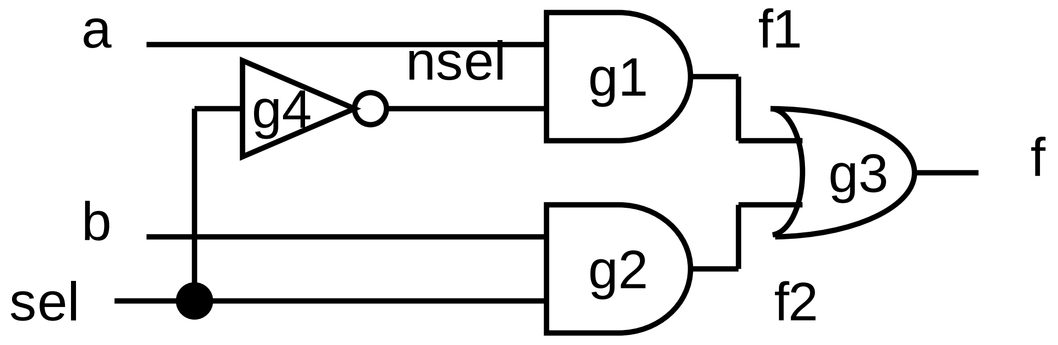


```
module mux(f, a, b, sel);
output f;
input a, b, sel;

and       g1(f1, a, nsel),
          g2(f2, b, sel);
or        g3(f, f1, f2);
not g4(nsel, sel);

endmodule
```

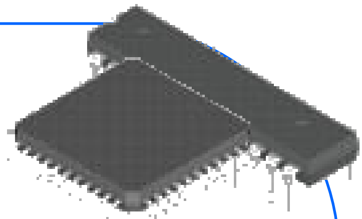
Verilog programs built from modules

Each module has an interface

Module may contain structure: instances of primitives and other modules

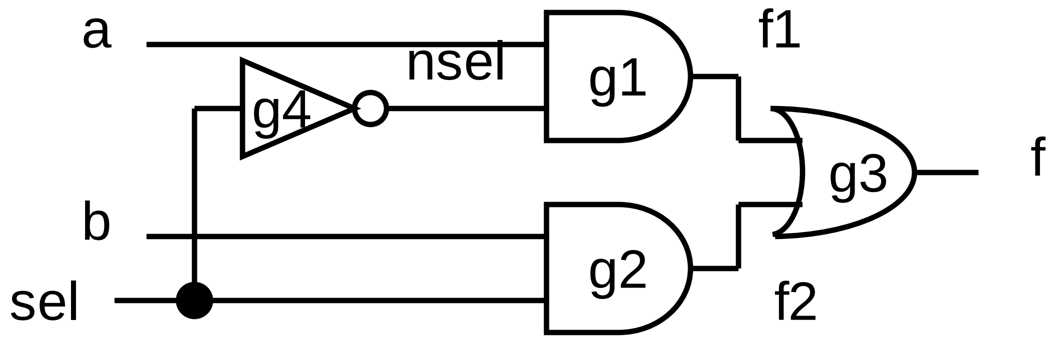


Multiplexer Built From Primitives



```
module mux(f, a, b, sel);
output f;
input a, b, sel;

and     g1(f1, a, nsel),
        g2(f2, b, sel);
or      g3(f, f1, f2);
not g4(nsel, sel);

endmodule
```

Identifiers not explicitly defined default to wires

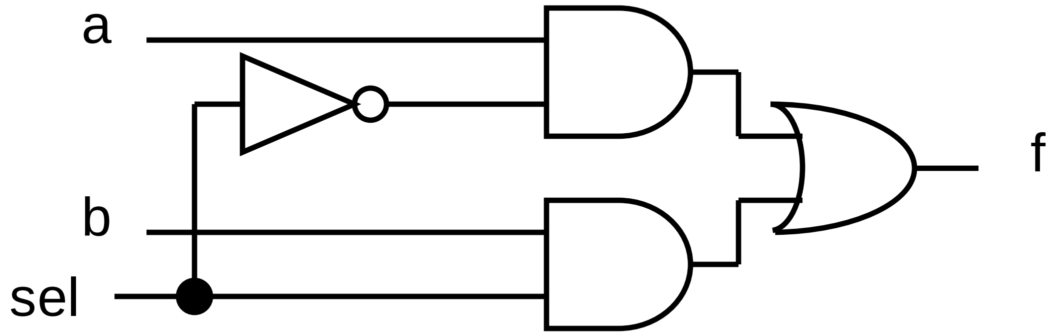


Multiplexer Built With Always

```
module mux(f, a, b, sel);
output f;
input a, b, sel;
reg f;

always @(a or b or sel)
  if (sel) f = b;
  else f = a;

endmodule
```

Modules may contain one or more *always* blocks

Sensitivity list contains signals whose change triggers the execution of the block

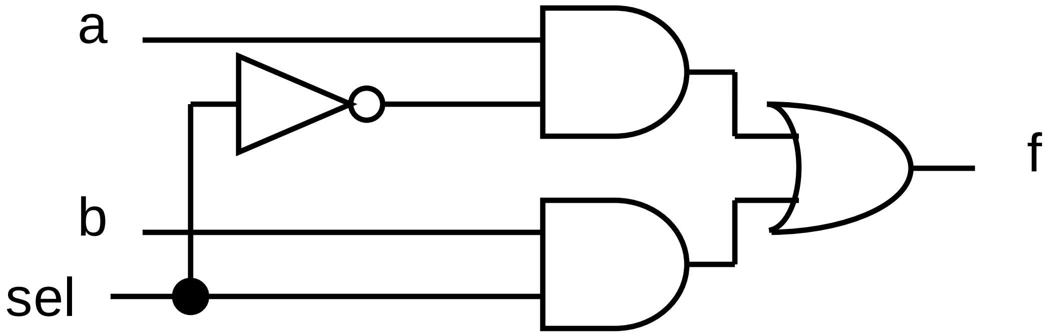


Multiplexer Built With Always

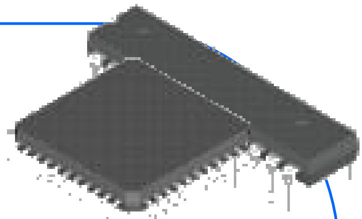
```
module mux(f, a, b, sel);
output f;
input a, b, sel;
reg f;

always @(a or b or sel)
  if (sel) f = b;
  else f = a;

endmodule
```

A *reg* behaves like memory: holds its value until imperatively assigned otherwise

Body of an *always* block contains traditional imperative code



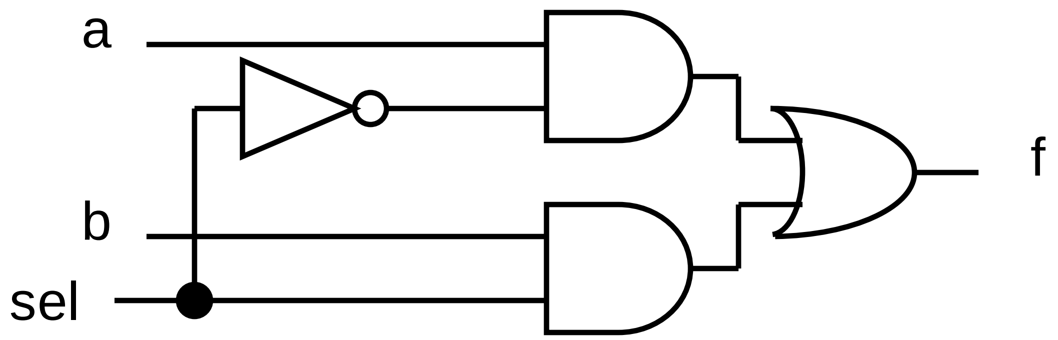
Mux with Continuous Assignment



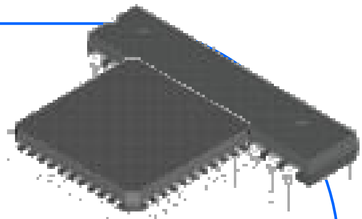
```
module mux(f, a, b, sel);
output f;
input a, b, sel;

assign f = sel ? b : a;

endmodule
```

LHS is always set to the value on the RHS

Any change on the right causes re-evaluation

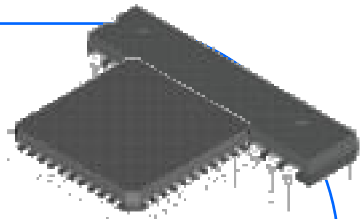


Identifiers in Verilog



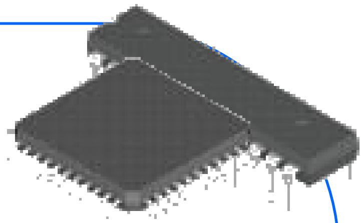
- Any Sequence of letter, digits, dollar sign, underscore.
- First character must be a letter or underscore.
- It cannot be a dollar sign.
- Cannot use characters such as hyphen, brackets, or # in verilog names

Verilog Logic Values



- Predefined logic value system or value set : ‘0’, ‘1’ ,’x’ and ‘z’;
- ‘x’ means uninitialized or unknown logic value
- ‘z’ means high impedance value.

Verilog Data Types



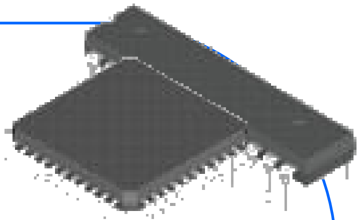
- Nets: wire, supply1, supply0
  - wire:

    i) Analogous to a wire in an ASIC.

    ii) Cannot store or hold a value.

- Integer: used for the index variables of say for loops. No hardware implication.

The reg Data Type



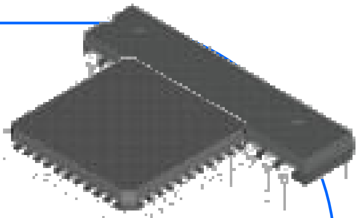
- Register Data Type: Comparable to a variable in a programming language.
- Default initial value: ‘x’
- module reg_ex1;

```
reg Q; wire D;
always @(posedge clk) Q=D;
```

- A reg is not always equivalent to a hardware register, flipflop or latch.
- module reg_ex2; // purely combinational

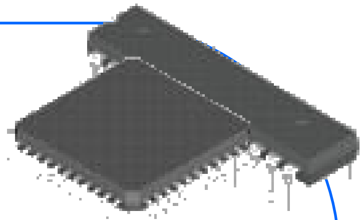
```
reg c;
always @(a or b) c=a|b;
endmodule
```

Difference between driving and assigning



- Programming languages provide variables that can contain arbitrary values of a particular type.
- They are implemented as simple memory locations.
- Assigning to these variables is the simple process of storing a value into the memory location.
- Verilog reg operates in the same way. Previous assignments have no effect on the final result.

Example



-
  ```
  module assignments;
  reg R;
  initial R<=#20 3;
  initial begin
      R=5; R=#35 2;
  end
  initial begin
      R<=#100 1;
      #15 R=4;
      #220;
      R=0;
   end
  endmodule
  ```

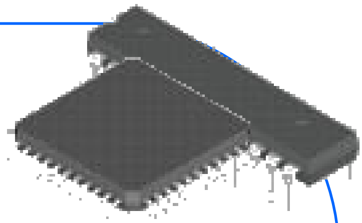
The variable R is shared by all the concurrent blocks.

R takes the value that was last assigned.

This is like a hardware register which also stores the value that was last loaded into them.

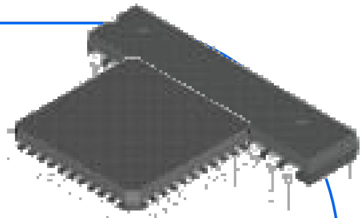
But a reg is not necessarily a hardware register.

Wire: helps to connect



- Consider a set of tristate drivers connected to a common bus.
- The output of the wire depends on *all* the outputs and not on the last one.
- To model connectivity, any value driven by a device must be driven continuously onto that wire, in parallel with the other driving values.

Code

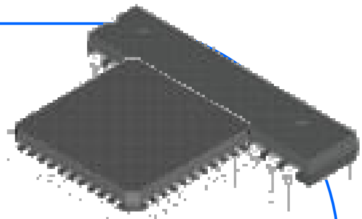


```
module simple(A,B,C,sel,Z);
input A, B, C;
input [1:0] sel;
output Z;
reg Z;
always @(A or B or C or SEL)
 begin
     2'b00: Z=1'bz;
     2'b01: Z=A;
     2'b10: Z=B;
     2'b11: Z=C;
  endcase
 end
endmodule
```

Inferred as a multiplexer.
But we wanted drivers!

```
module simple(A,B,C,sel,Z);
input A, B, C;
input [1:0] sel;
output Z;

assign Z=(SEL==2'b01)?A: 1'bz;
assign Z=(SEL==2'b10)?B:1'bz;
assign Z=(SEL==2'b11)?C:1'bz;

endmodule
```

Numbers

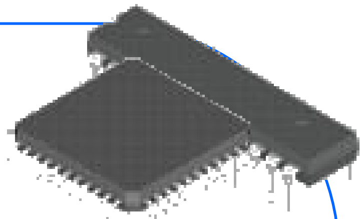


- Format of integer constants:

  Width’ radix value;
- Verilog keeps track of the sign if it is assigned to an integer or assigned to

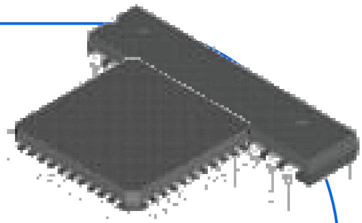
  a parameter.
- Once verilog looses sign the designer has to be careful.

Hierarchy



- Module interface provides the means to interconnect two verilog modules.
- Note that a reg cannot be an input/ inout port.
- A module may instantiate other modules.

Instantiating a Module



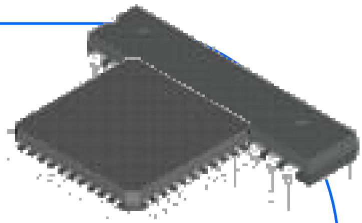
- Instances of

```
module mymod(y, a, b);
```

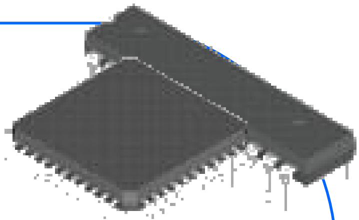
- Lets **instantiate** the module,

```
mymod mm1(y1, a1, b1);                    // Connect-by-position

mymod mm2(.a(a2), .b(b2), .y(c2));  // Connect-by-name
```

Sequential Blocks



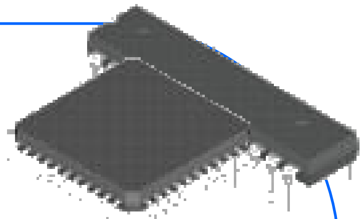
- Sequential block is a group of statements between a begin and an end.
- A sequential block, in an always statement executes repeatedly.
- Inside an initial statement, it operates only once.



Procedures

- A Procedure is an always or initial statement or a function.
- Procedural statements within a sequential block executes concurrently with other procedures.

Assignments



- module assignments

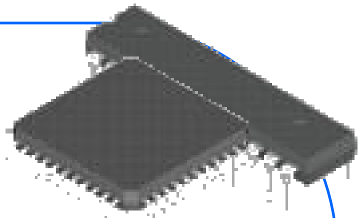
```
// continuous assignments
always // beginning of a procedure
    begin //beginning of a sequential block
    //….Procedural assignments
    end
 endmodule
```

- A Continuous assignment assigns a value to a wire like a real gate driving a wire.

```
module holiday_1(sat, sun, weekend);
 input sat, sun; output weekend;
// Continuous assignment
 assign weekend = sat | sun;
 endmodule
```

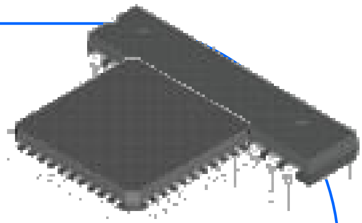
```
module holiday_2(sat, sun, weekend);
 input sat, sun; output weekend;
 reg weekend;
 always @(sat or sun)
    weekend = sat | sun; // Procedural
 endmodule                // assignment
```

Blocking and Nonblocking Assignments



- Blocking procedural assignments must be executed before the procedural flow can pass to the subsequent statement.
- A Non-blocking procedural assignment is scheduled to occur without blocking the procedural flow to subsequent statements.

Nonblocking Statements are odd!



```
a = 1;
b = a;
c = b;
```

Blocking assignment:

a = b = c = 1

```
a <= 1;
b <= a;
c <= b;
```

Nonblocking assignment:

a = 1

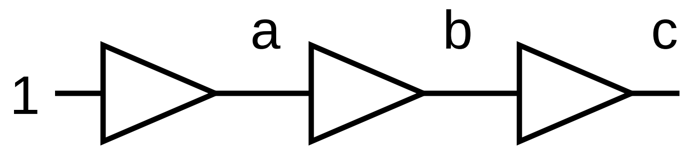
b = old value of a

c = old value of b

Nonblocking Looks Like Latches

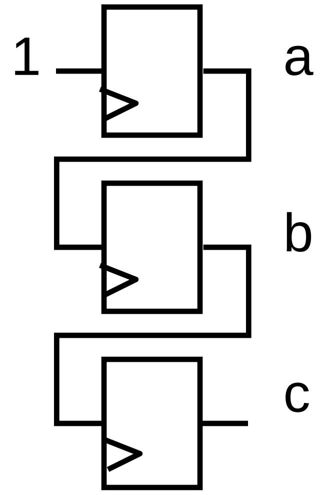
- RHS of nonblocking taken from latches
- RHS of blocking taken from wires

```
a = 1;
b = a;
c = b;
```

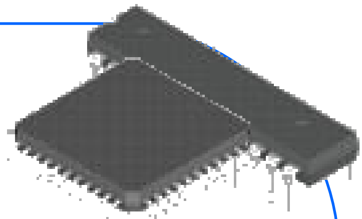
“


„

```
a <= 1;
b <= a;
c <= b;
```

“


”



Examples

- Blocking:

```
always @(A1 or B1 or C1 or M1)
 begin
        M1=#3(A1 & B1);
        Y1= #1(M1|C1);
 end
```

- Non-Blocking:

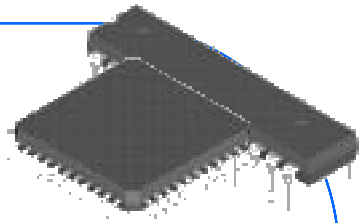
```
always @(A2 or B2 or C2 or M2)
begin
        M2<=#3(A2 & B2);
        Y2<=#1(M1 | C1);
 end
```

Statement executed at time t causing M1 to be assigned at t+3

Statement executed at time t+3 causing Y1 to be assigned at time t+4

Statement executed at time t causing M2 to be assigned at t+3

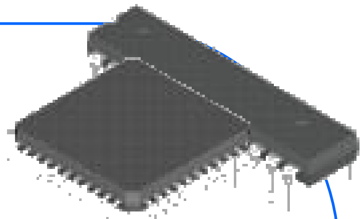
Statement executed at time t causing Y2 to be assigned at time t+1. Uses old values.



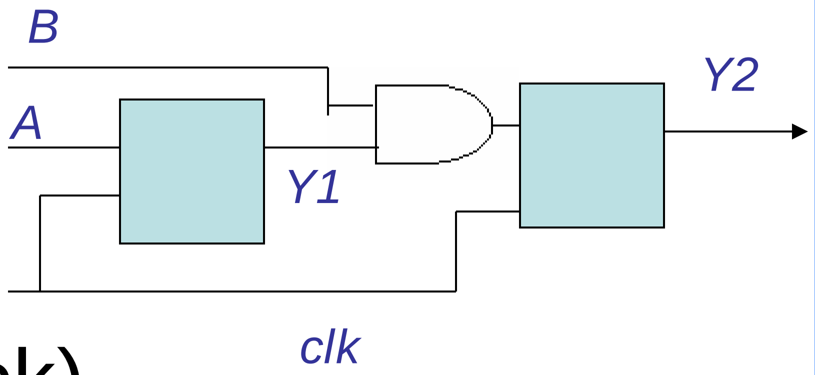
Order dependency of Concurrent Statements

- Order of concurrent statements does not affect how a synthesizer synthesizes a circuit.
- It can affect simulation results.

Order dependency of Concurrent Statements

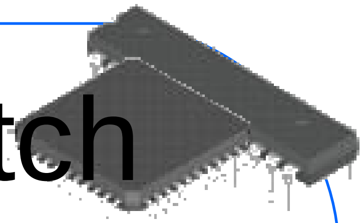


```
• always @(posedge clock)
      begin: CONCURR_1
          Y1<=A;
      end
• always @(posedge clock)
      begin: CONCURR_2
          if(Y1) Y2=B; else Y2=0;
      end
```



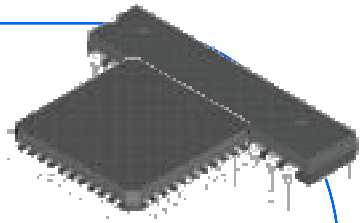
Can you figure out the possible mismatch of simulation with synthesis results?

Explanation of the mismatch



- The actual circuit is a *concurrent* process.
- The first and second flip flop are operating parallel.
- However if the simulator simulates CONCURR_1 block before CONCURR_2, we have an error. Why?
- So, how do we solve the problem?

Solution

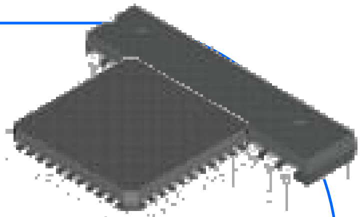


- always @(posedge clock)

```
begin
  Y1<=A;
  if(Y1==1)
     Y2<=B;
  else
      Y2<=0;
end
```

With non-blocking assignments the order of the assignments is immaterial…

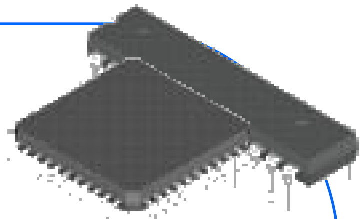
Parameterized Design



-
  ```
  module vector_and(z, a, b);
   parameter cardinality = 1;
   input [cardinality-1:0] a, b;
   output [cardinality-1:0] z;
   wire [cardinality-1:0] z = a & b;
  endmodule
  ```

- We override these parameters when we instantiate the module as:
  ```
  module Four_and_gates(OutBus, InBusA, InBusB);
    input [3:0] InBusA, InBusB; output[3:0] OutBus;
    Vector_And #(4) My_And(OutBus, InBusA, InBusB);
  endmodule
  ```

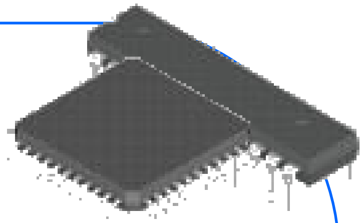
Functions (cont'd)



- Function Declaration and Invocation
  - Declaration syntax:

```
function <range_or_type> <func_name>;
  <input declaration(s)>
  <variable_declaration(s)>
  begin // if more than one statement needed
    <statements>
  end  // if begin used
endfunction
```

Function Examples Controllable Shifter

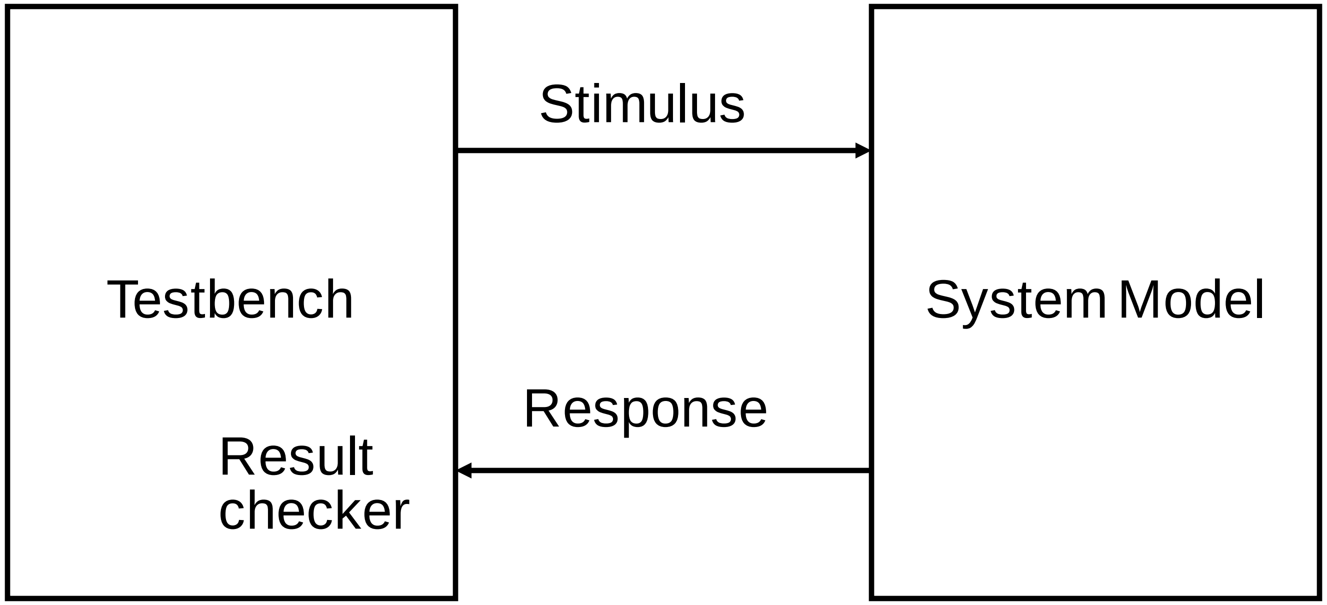


```
module shifter;
`define LEFT_SHIFT      1'b0
`define RIGHT_SHIFT     1'b1
reg [31:0] addr, left_addr,
   right_addr;
reg control;

initial
begin
   …
end
always @(addr)begin
  left_addr  =shift(addr,
   `LEFT_SHIFT);
  right_addr
   =shift(addr,`RIGHT_SHIFT);
end
```

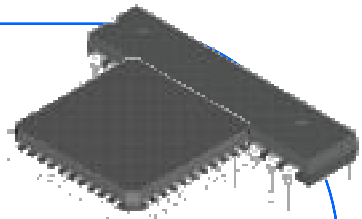
```
function [31:0]shift;
input [31:0] address;
input control;
begin
  shift = (control==`LEFT_SHIFT)
   ?(address<<1) : (address>>1);
end
endfunction

endmodule
```

How Are Simulators Used?

- Testbench generates stimulus and checks response
- Coupled to model of the system
- Pair is run simultaneously



Looking back at our multiplexer



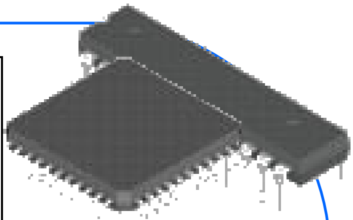
- “Dataflow” Descriptions of Logic

```
//Dataflow description of mux
module mux2 (in0, in1, select, out);
   input in0,in1,select;
   output out;
   assign out = (~select & in0)
                 | (select & in1);
endmodule // mux2
```

Alternative:

```
   assign out = select ? in1 : in0;
```

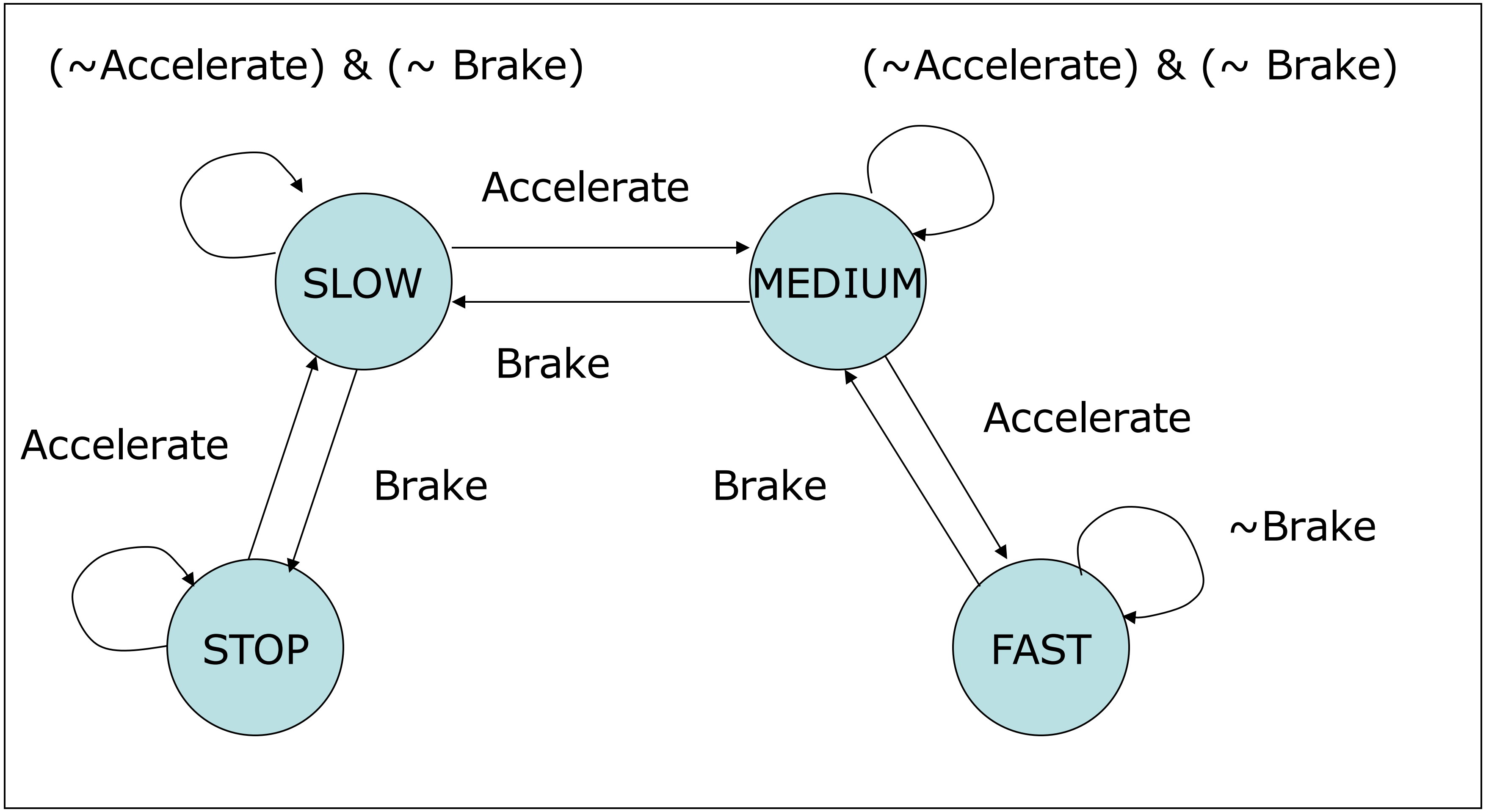
TestBench of the Multiplexer



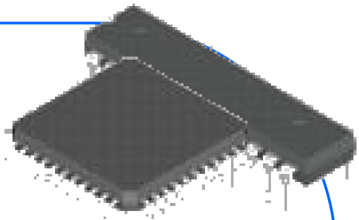
- Testbench

```
module  testmux;
 reg a, b, s;
 wire f;
 reg expected;

 mux2  myMux  (.select(s), .in0(a), .in1(b),  .out(f));

 initial
           begin
                  s=0; a=0; b=1; expected=0;
     #10 a=1; b=0; expected=1;
     #10 s=1; a=0; b=1; expected=1;
   end
initial
   $monitor(
       "select=%b in0=%b in1=%b out=%b, expected out=%b time=%d",
        s, a, b, f, expected, $time);
endmodule  // testmux
```

A Car Speed Controller

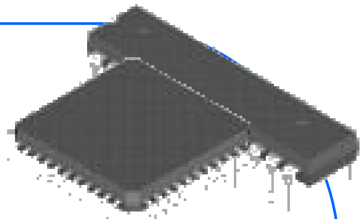


Car Controller Coding



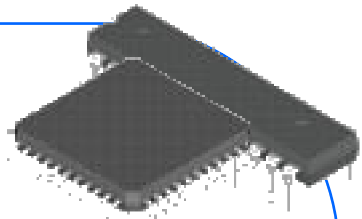
```
module fsm_car_speed_1(clk, keys, brake, accelerate,
    speed);
  input clk, keys, brake, accelerate;
  output [1:0] speed;
  reg [1:0] speed;

  parameter stop   = 2'b00,
            slow = 2'b01,
            mdium = 2'b10,
            fast = 2'b11;
```

Car Controller (contd.)



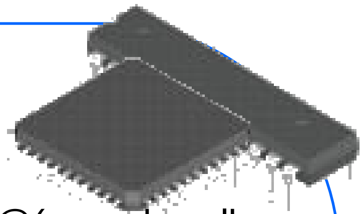
```
always @(posedge clk or
     negedge keys)
   begin
    if(!keys)
      speed = stop;
    else if(accelerate)
      case(speed)
        stop: speed = slow;
     slow: speed = mdium;
      mdium: speed = fast;
        fast: speed = fast;
      endcase
    else if(brake)
      case(speed)
        stop: speed = stop;
      slow: speed = stop;
       mdium: speed = slow;
         fast: speed = mdium;
       endcase
     else
       speed = speed;
   end

endmodule
```

A Better Way!

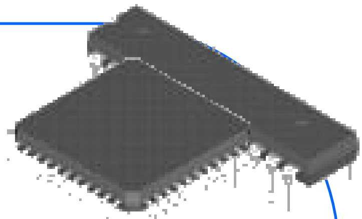


- We keep a separate control part where the next state is calculated.
- The other part generates the output from the next state.
- We follow this architecture in the coding of any finite state machines, like ALU, etc. (to be discussed later)

```
module  fsm_car_speed_2(clk, keys,
      brake, accelerate, speed);
  input clk, keys, brake, accelerate;
  output  [1:0] speed;
  reg [1:0] speed;
  reg [1:0] newspeed;

  parameter stop   = 2'b00,
            slow = 2'b01,
            mdium  = 2'b10,
            fast = 2'b11;

always @(keys or brake or
      accelerate or speed)
 begin
  case(speed)
   stop:
     if(accelerate)
       newspeed = slow;

     else
       newspeed = stop;
slow:
     if(brake)
       newspeed = stop;
     else if(accelerate)
       newspeed = mdium;
     else
       newspeed = slow;
   mdium:
     if(brake)
       newspeed = slow;
     else if(accelerate)
       newspeed = fast;
     else
       newspeed = mdium;
   fast:
     if(brake)
       newspeed = mdium;
     else
       newspeed = fast;
   default:
       newspeed = stop;
  endcase
 end

always @(posedge clk or
      negedge keys)
  begin
   if(!keys)
    speed = stop;
   else
    speed = newspeed;
  end

endmodule
```



Conclusion : Write codes which can be translated into hardware !



The following cannot be translated into hardware(non - synthesizable):

- Initial blocks
  - Used to set up initial state or describe finite testbench stimuli
  - Don’t have obvious hardware component
- Delays
  - May be in the Verilog source, but are simply ignored
- In short, write codes with a hardware in your mind. In other words do not depend too much upon the tool to decide upon the resultant hardware.
- Finally, remember that you are a better designer than the tool.