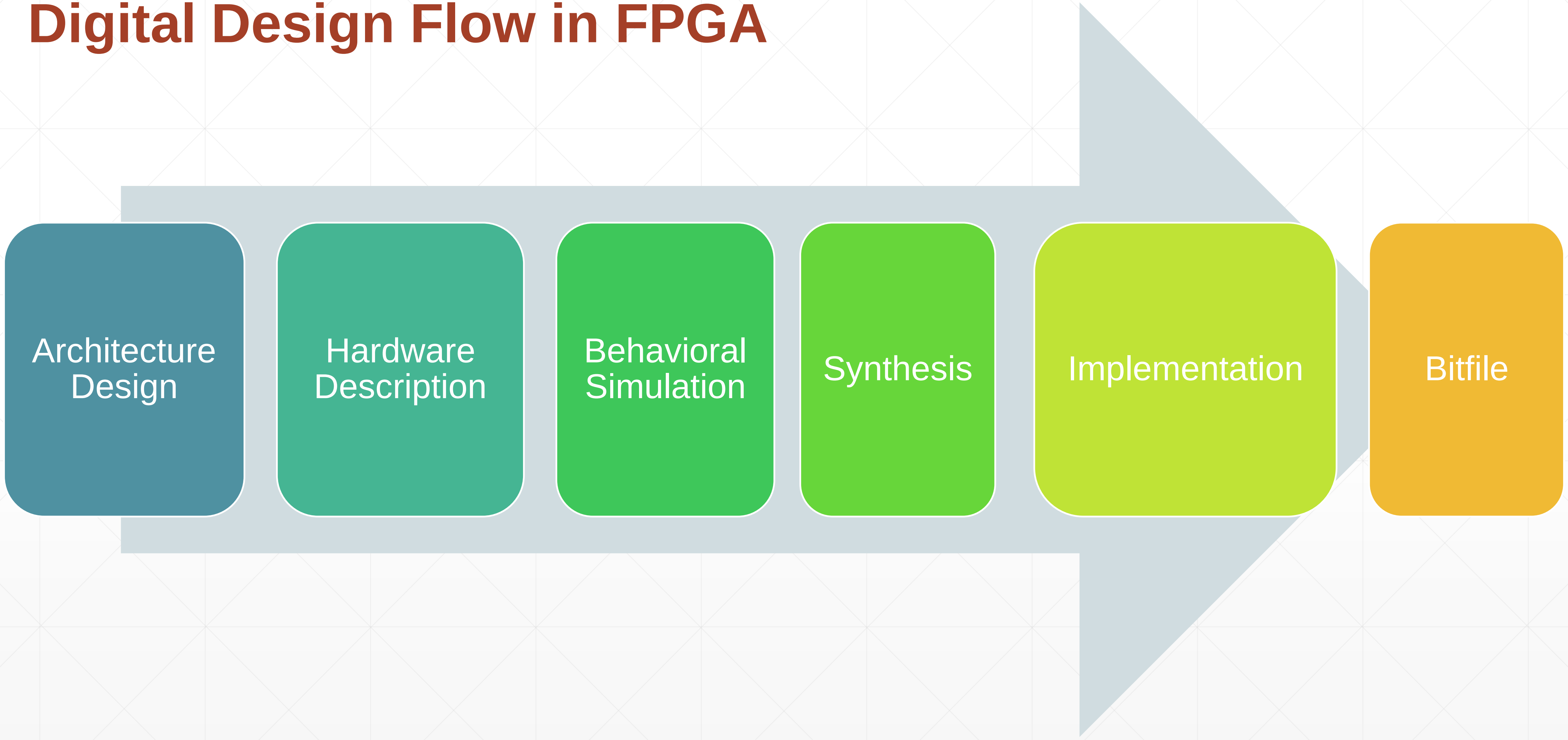
Introduction to Verilog

Siddhartha Chowdhury

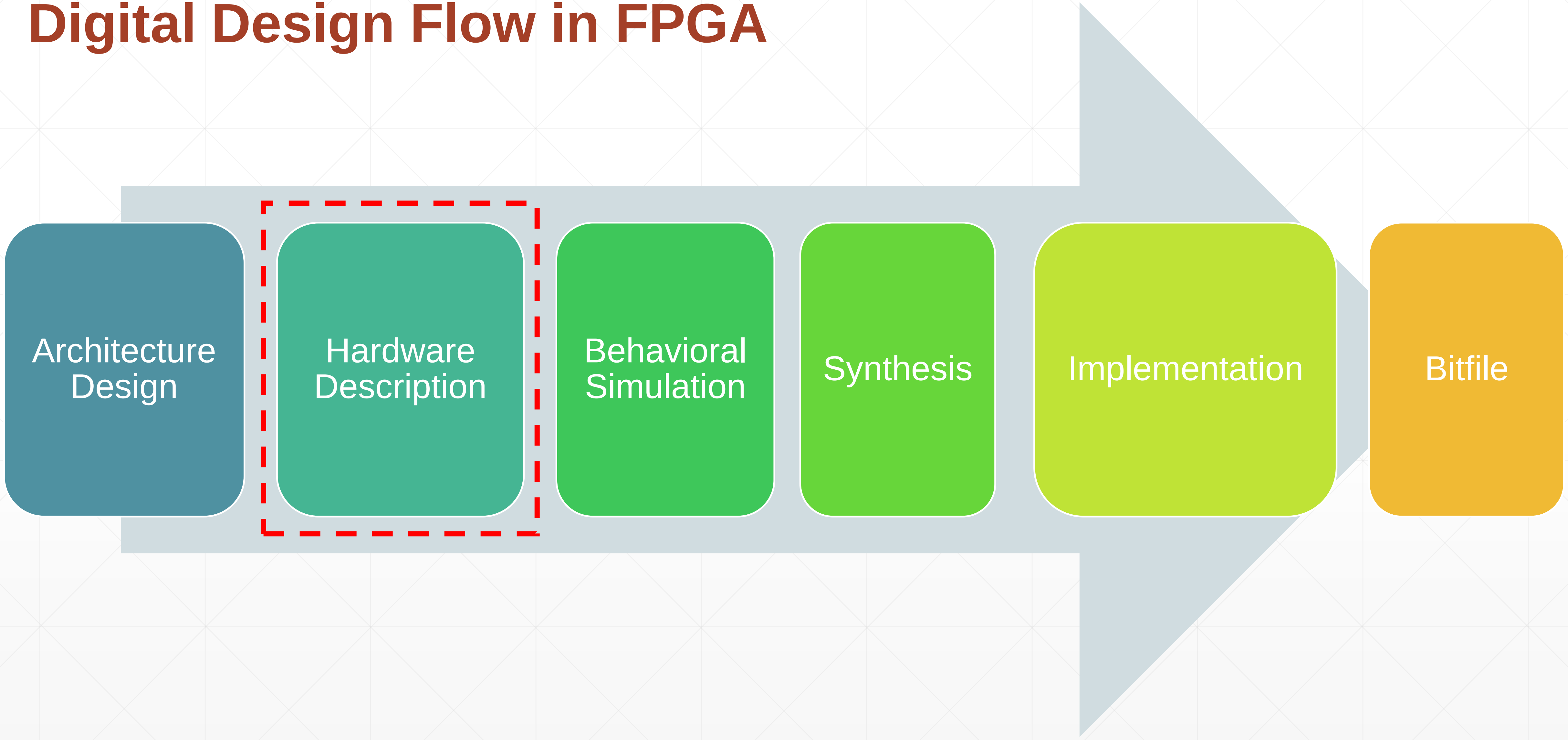
Secured Embedded Architecture Laboratory (SEAL)
Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur

Bootcamp on Embedded Security and Trust

Digital Design Flow in FPGA



Digital Design Flow in FPGA



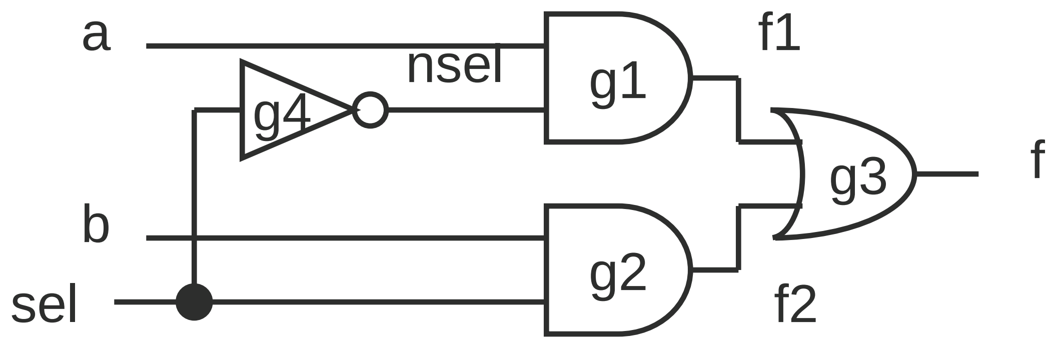
Multiplexer Built From Primitives

```
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

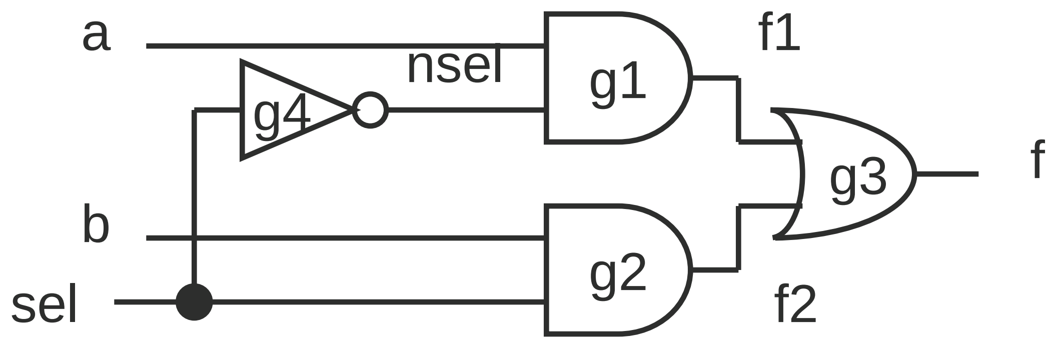
Module may contain structure: instances of primitives and other modules



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

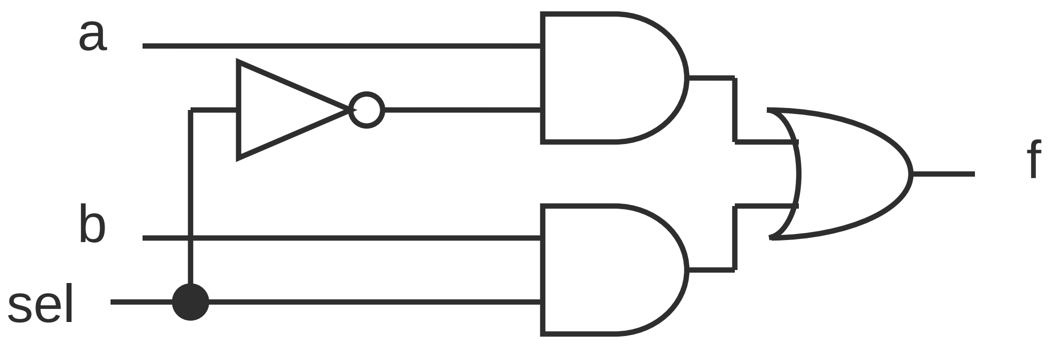


Multiplexer Built With Always

```
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

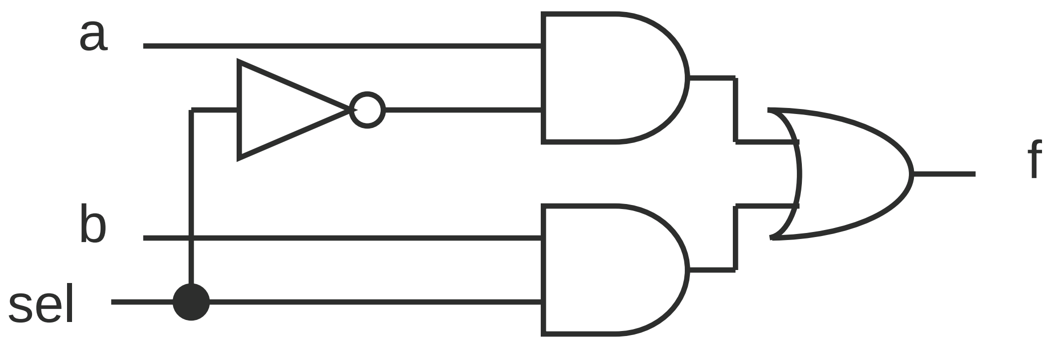


Multiplexer Built With Always

```
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

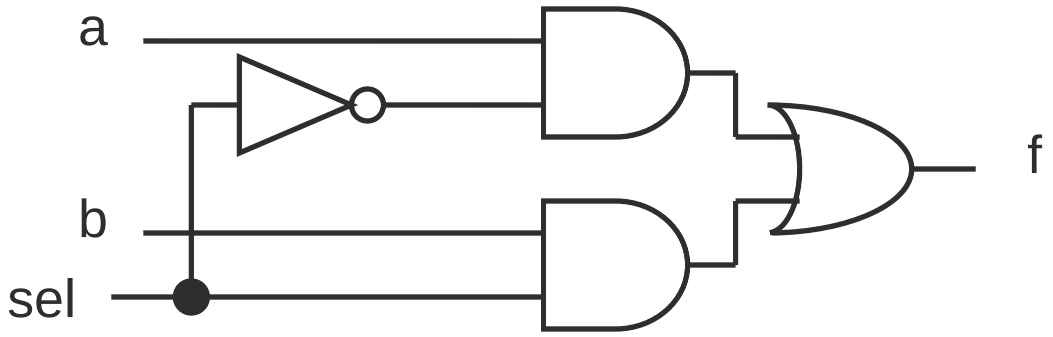


Mux with Continuous Assignment

```
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
```

Architecture Design

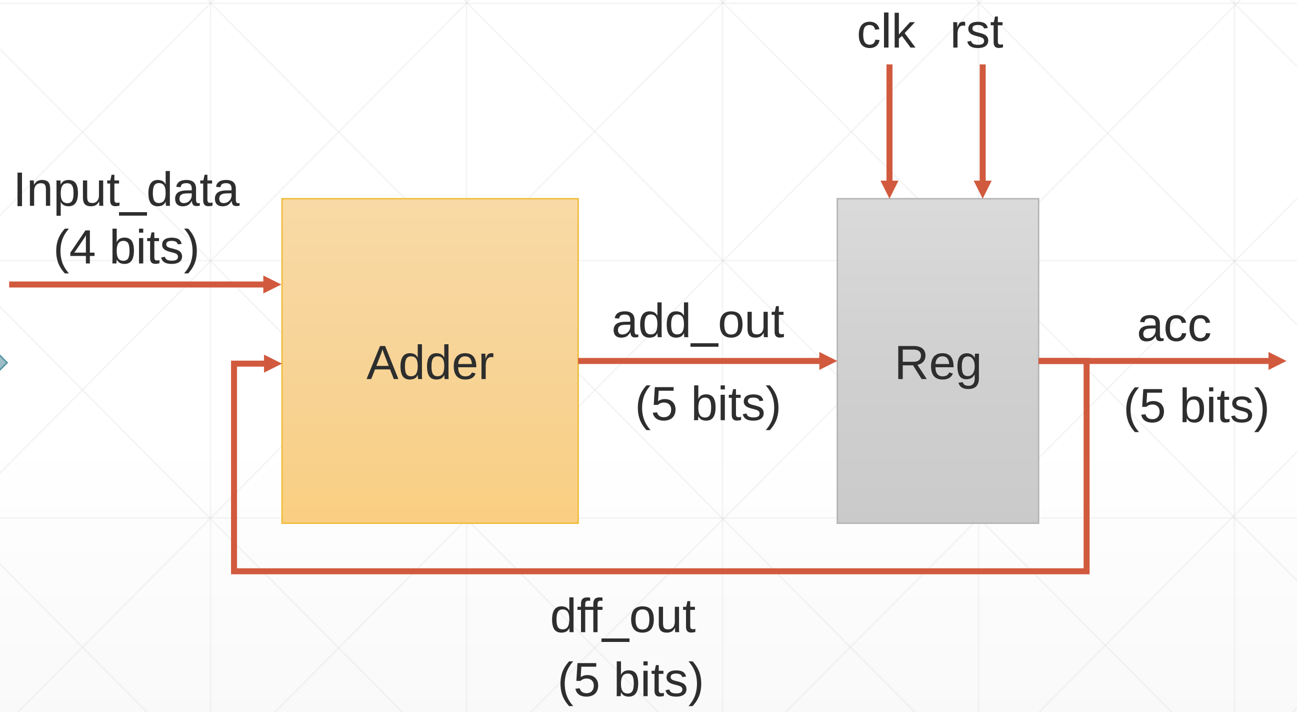
# Accumulator Algorithm
accumulator = 0

# Loop for N data inputs
input_data = get_input() # Fetch new data

accumulator += input_data # Accumulate (accumulator = accumulator + input_data)

# Output the final accumulated value **print(accumulator)**

Algorithm of the Accumulator



Hardware Design Architecture of the Accumulator

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
```

Code

```
module tristate_bus (
    input wire enable1, enable2,
    input wire [7:0] data1, data2,
    output wire [7:0] bus
);

    assign bus = enable1 ? data1 : 8'bz; // Tristate driver 1

    assign bus = enable2 ? data2 : 8'bz; // Tristate driver 2

endmodule
```

- If enable1 is 1 and enable2 is 0, bus takes data1.
- If enable2 is 1 and enable1 is 0, bus takes data2.
- If both enable signals are 1 and data1 ≠ data2, the bus enters an undefined (x) state due to contention.

Code

```
module tristate_bus (
    input [7:0] data1, data2,
    output [7:0] bus
);

    assign bus = data1

    assign bus = data2
endmodule
```

Error: "Net out_signal has multiple drivers."

Code

```
module multi_driver_error (
    input wire clk,
    input wire rst,
    output reg out_signal
);

    always @(posedge clk) begin
        if (rst)
            out_signal <= 0; // First driver
    end

    always @(posedge clk) begin
        out_signal <= 1; // Second driver (Conflicting)
    end

endmodule
```

Error: "Signal out_signal has multiple drivers. "

Code

```
module multi_driver_fix (
    input wire clk,
    input wire rst,
    output reg out_signal
);

    always @(posedge clk) begin
        if (rst)
            out_signal <= 0;
        else
            out_signal <= 1;
    end

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
```

Instantiating a Module

```
module adder4 ( input
[3:0] a, input [3:0] b,
output [3:0] sum );

assign sum = a + b;

endmodule
```

```
module top_module;

reg [3:0] x, y;

wire [3:0] result;

adder4 uut (x, y, result);

initial

begin

x = 4'b0011;

y = 4'b0101;

#10;

end

endmodule
```

```
module top_module;

reg [3:0] x, y;

wire [3:0] result;

adder4 uut (.a(x), .b(y), .sum(result));

initial

begin

x = 4'b0011;

y = 4'b0101;

#10;

end

endmodule
```

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

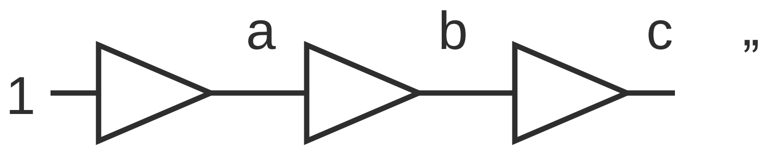
b = old value of a

c = old value of b

Nonblocking Looks Like Latches

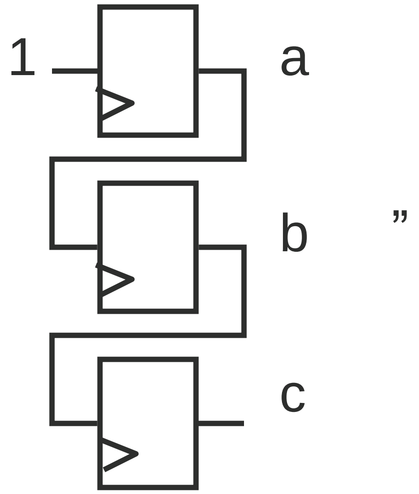
- RHS of nonblocking taken from latches
- RHS of blocking taken from wires

```
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;

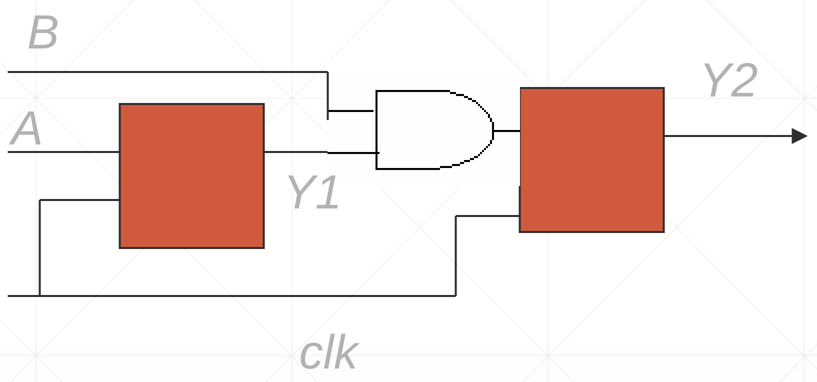
  end

- 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:

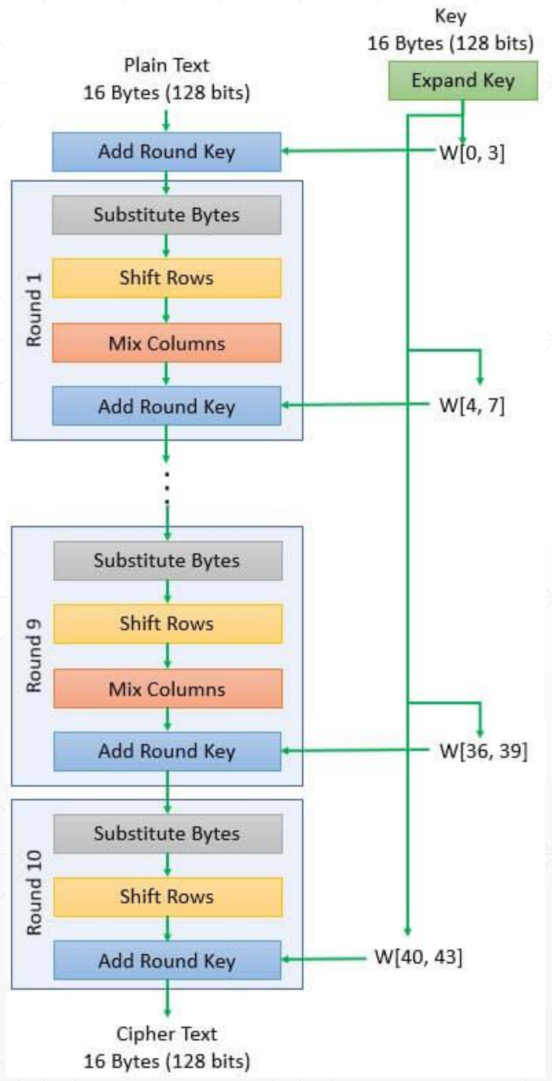
```
function <range_or_type> <func_name>;
 <input declaration(s)>
 <variable_declaration(s)>
 begin // if more than one statement needed
   <statements>
 end     // if begin used
endfunction
```

Function Examples Controllable Shifter

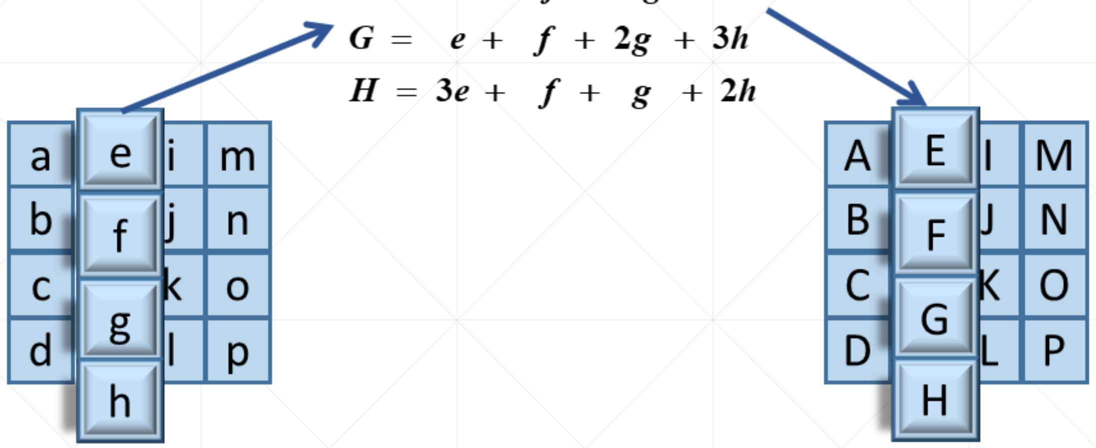
```
module shifter;
`define LEFT_SHIFT      1'b0
`define RIGHT_SHIFT     1'b1
reg [31:0] addr, left_addr,
   right_addr;
reg control;

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
```

Mix Column Operation



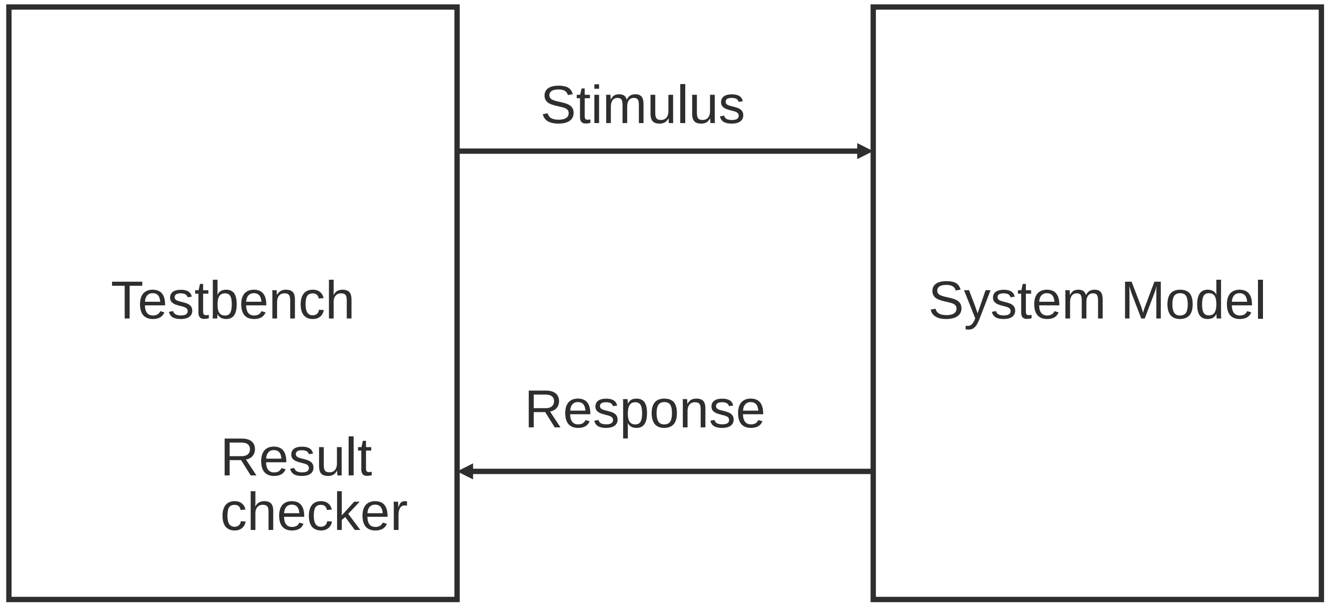
$$
\begin{aligned}
E &= 2e + 3f + g + h \\
F &= e + 2f + 3g + h \\
G &= e + f + 2g + 3h \\
H &= 3e + f + g + 2h
\end{aligned}
$$



```
function [7:0] mixcolumn32;
input [7:0] i1,i2,i3,i4;
begin
mixcolumn32[7]=i1[6] ^ i2[6] ^ i2[7] ^ i3[7] ^ i4[7];
mixcolumn32[6]=i1[5] ^ i2[5] ^ i2[6] ^ i3[6] ^ i4[6];
mixcolumn32[5]=i1[4] ^ i2[4] ^ i2[5] ^ i3[5] ^ i4[5];
mixcolumn32[4]=i1[3] ^ i1[7] ^ i2[3] ^ i2[4] ^ i2[7] ^ i3[4] ^ i4[4];
mixcolumn32[3]=i1[2] ^ i1[7] ^ i2[2] ^ i2[3] ^ i2[7] ^ i3[3] ^ i4[3];
mixcolumn32[2]=i1[1] ^ i2[1] ^ i2[2] ^ i3[2] ^ i4[2];
mixcolumn32[1]=i1[0] ^ i1[7] ^ i2[0] ^ i2[1] ^ i2[7] ^ i3[1] ^ i4[1];
mixcolumn32[0]=i1[7] ^ i2[7] ^ i2[0] ^ i3[0] ^ i4[0];
end
endfunction
```

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:

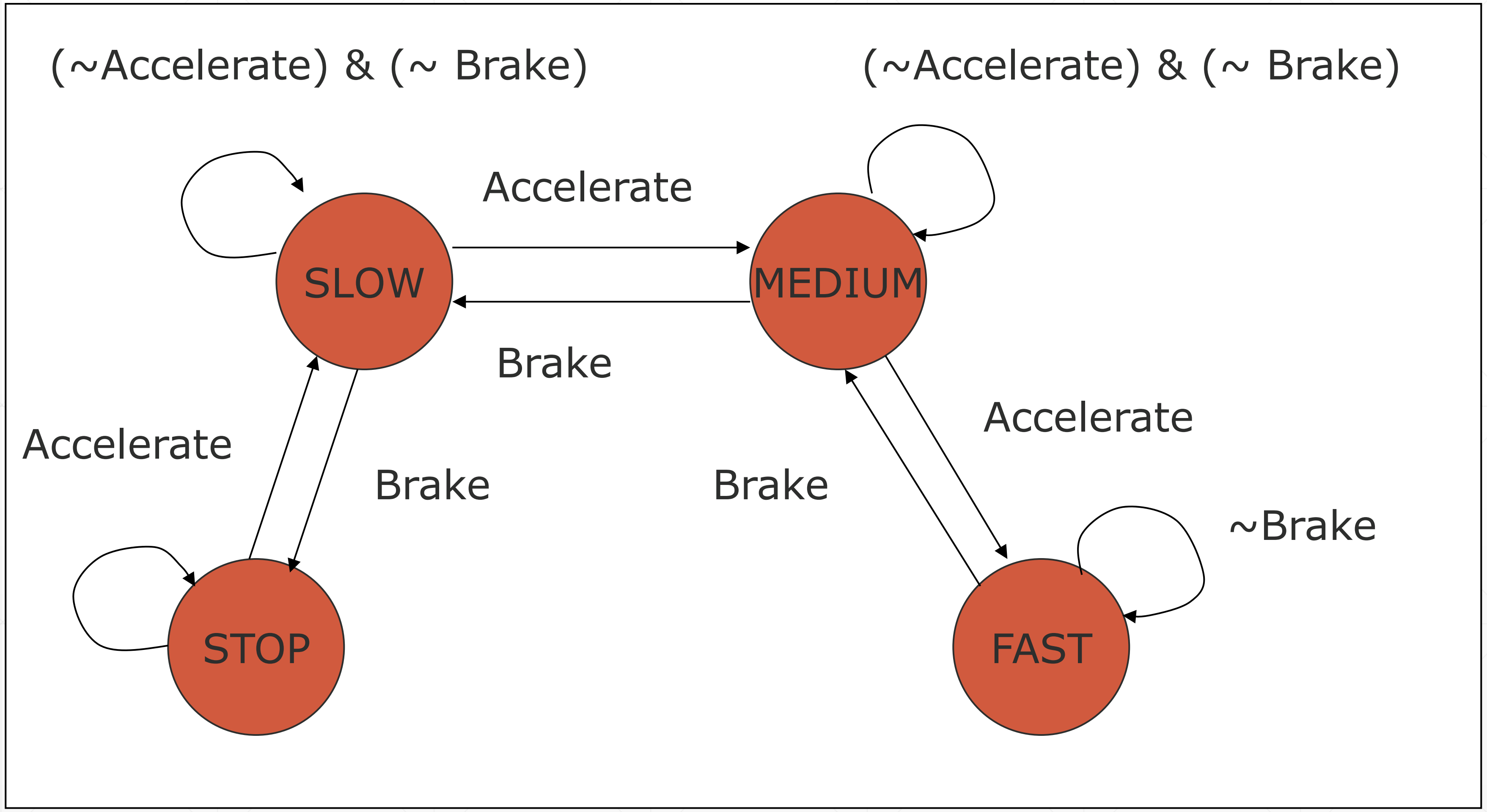
```
  assign out = select ? in1 : in0;
```

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 = medium;
      medium: speed = fast;
        fast: speed = fast;
      endcase
```

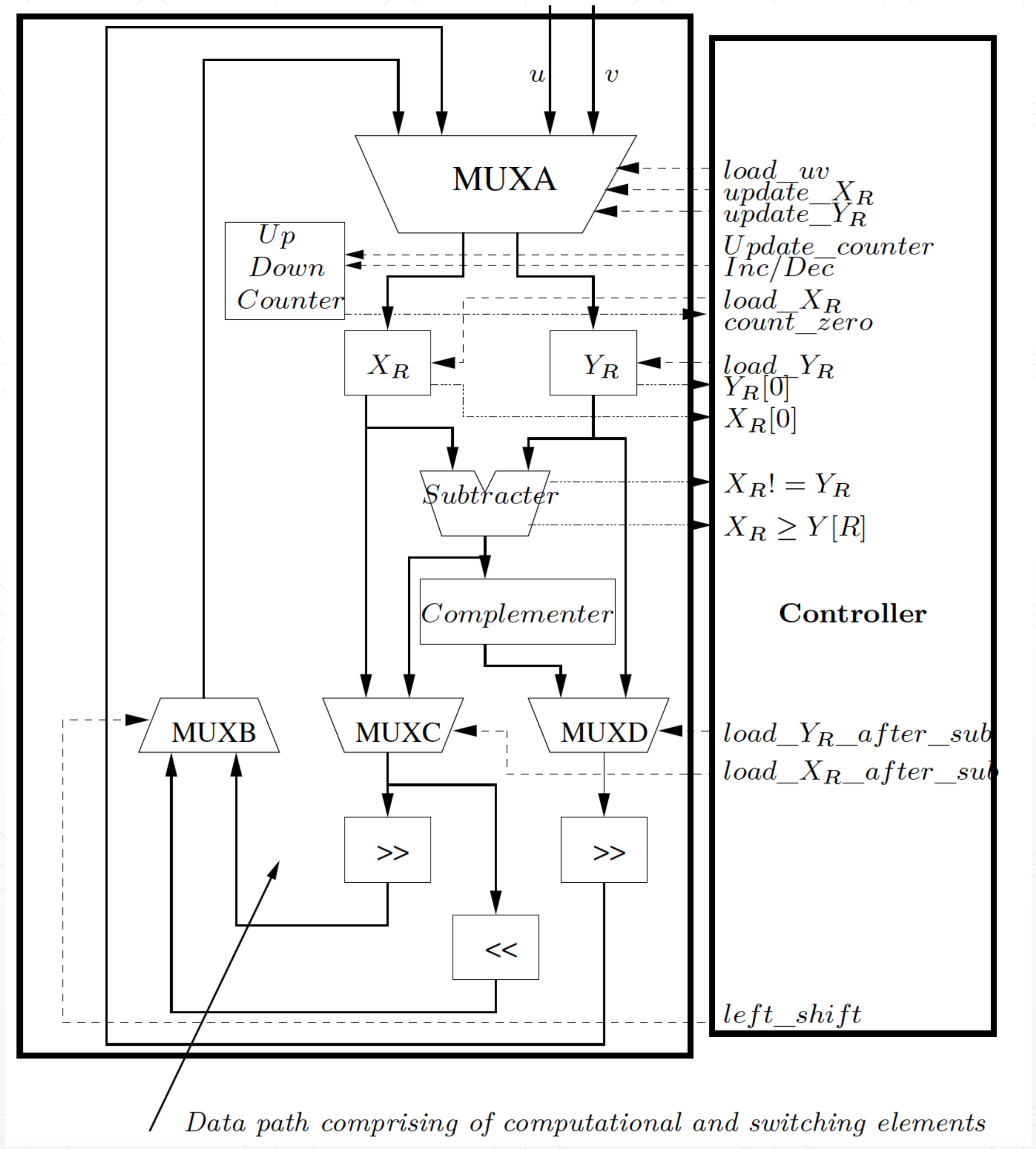
```
else if(brake)
     case(speed)
        stop: speed = stop;
     slow: speed = stop;
      medium: speed = slow;
        fast: speed = medium;
      endcase
    else
      speed = speed;
   end

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.

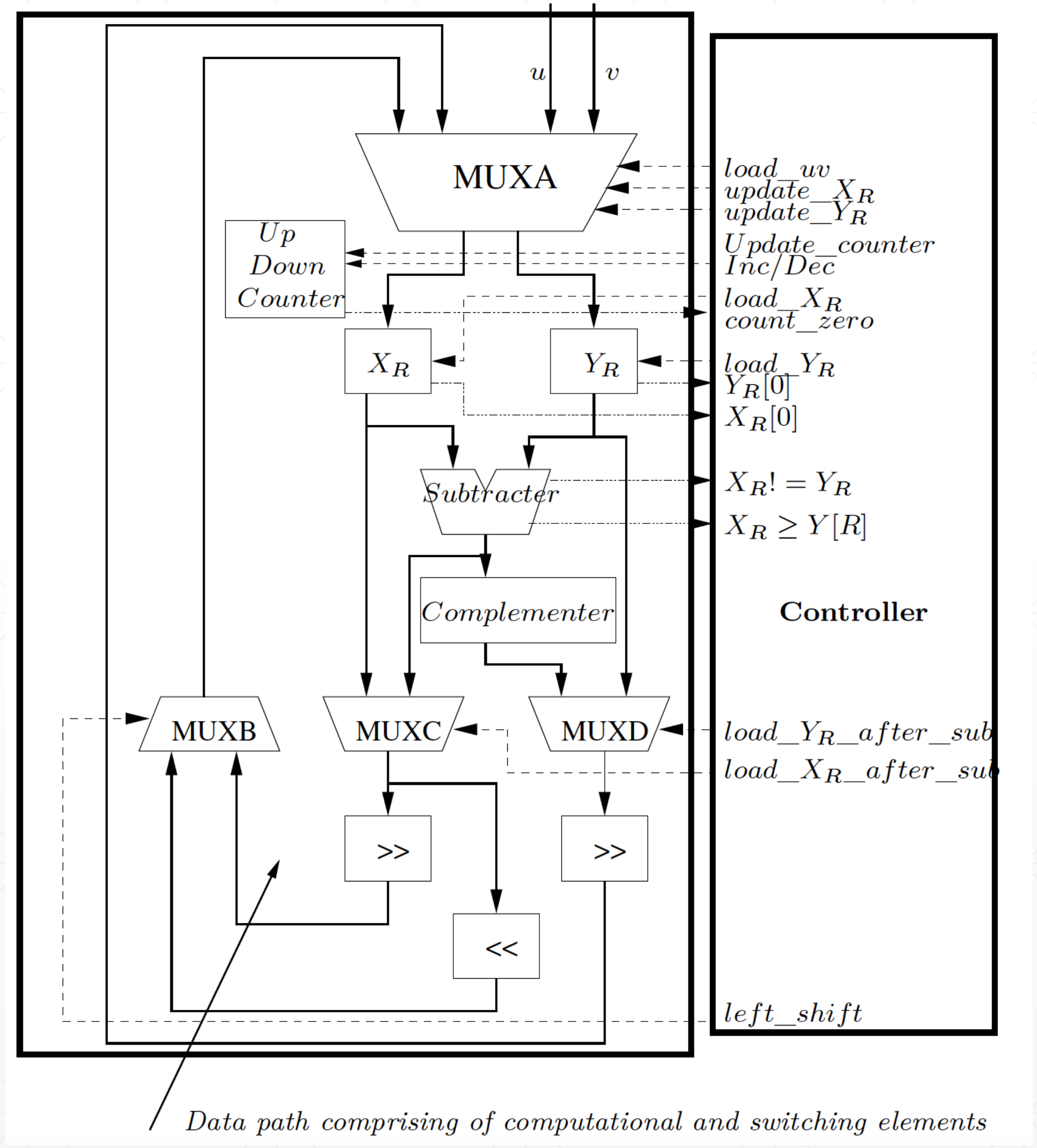
```
module fsm_car_speed_2(clk, keys, brake, accelerate, speed);
  input clk, keys, brake, accelerate;
  output [1:0] speed;
  reg [1:0] speed;
  reg [1:0] newspeed;

  parameter stop   = 2'b00,
            slow = 2'b01,
            medium = 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 = medium;
    else
      newspeed = slow;
  mdium:
    if(brake)
      newspeed = slow;
    else if(accelerate)
      newspeed = fast;
    else
      newspeed = medium;
  fast:
    if(brake)
      newspeed = medium;
    else
      newspeed = fast;
  default:
      newspeed = stop;
 endcase
end

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

endmodule
```

State Machine of the Controller



```
always @(posedge clk)
    begin
        if(reset)
            state<=S0;
        else
            state<=next_state;
    end
endmodule
```

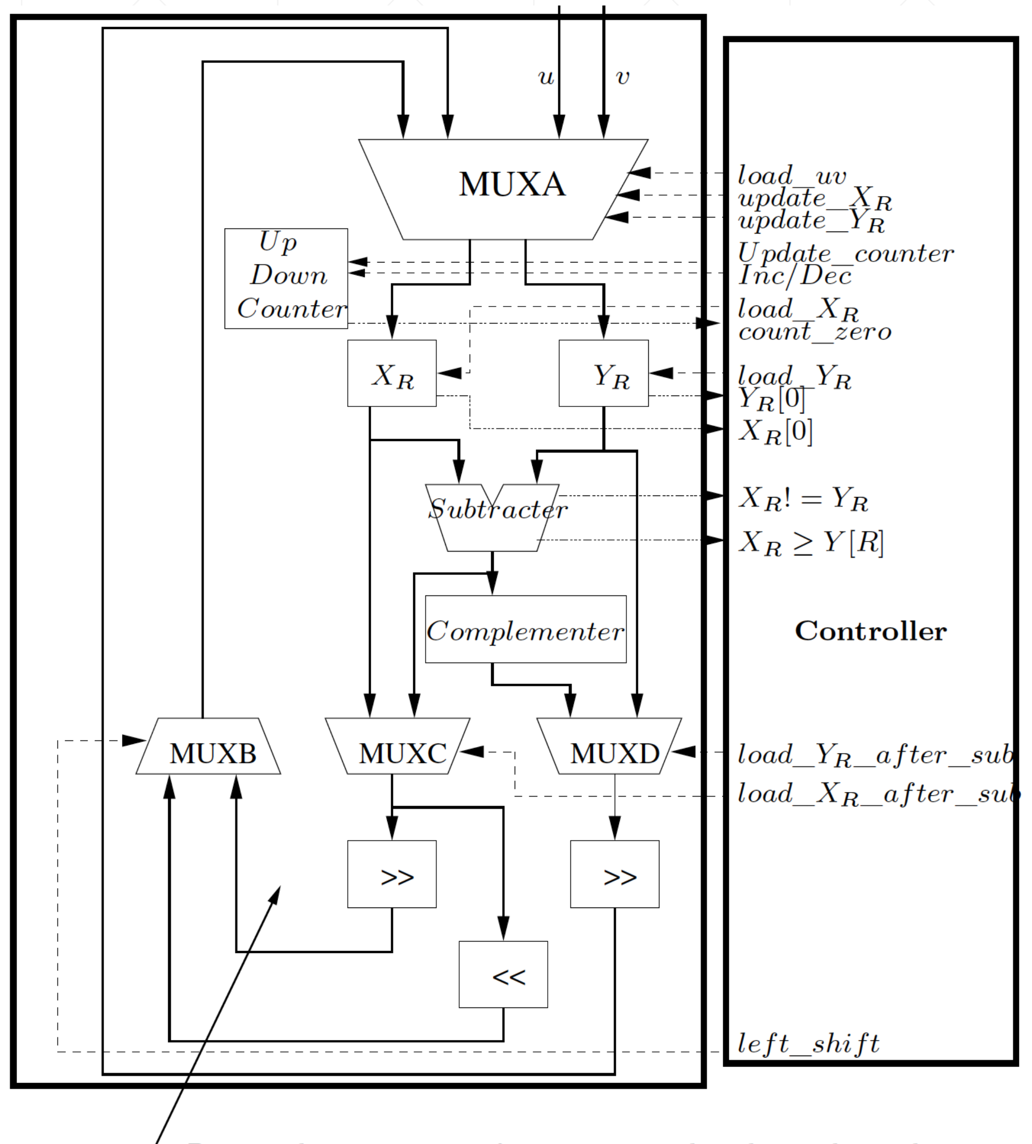
State Machine of the Controller



Data path comprising of computational and switching elements

```
localparam S0 = 3'd0;
localparam S1 = 3'd1;
localparam S2 = 3'd2;
localparam S3 = 3'd3;
localparam S4 = 3'd4;
localparam S5 = 3'd5;
always @(state or in)
  begin
    if(!in[0])
      next_state <=S5;
    else if(in[0] && in[3]==0 && in[2]==0)
      next_state <= S1;
    else if(in[0] && in[3]==1 && in[2]==0)
      next_state <= S2;
    else if(in[0] && in[3]==0 && in[2]==1)
      next_state <= S3;
    else if(in[0] && in[3]==1 && in[2]==1)
      next_state <= S4;
    else if(state == S5)
      next_state <= S5;
    else
      next_state <= S0;
    if(state == S5)
      next_state <= S5;
  end
```

State Machine of the Controller



Data path comprising of computational and switching elements

```
always @(state or in or done or cnt_zero)
   begin
      case(state)
         S0:
            begin
               ld_in1_in2<=1;
               update_Xr<=0;
               update_Yr<=0;
               ld_Xr<=1;
               ld_Yr<=1;
               ld_Xr_aftr_subs<=0;
               ld_Yr_aftr_subs<=0;
               lShift<=0;
            end
         S1:
            begin
               ld_in1_in2<=0;
               update_Xr<=1;
               update_Yr<=1;
               update_Counter<=1;
               incr_dec<=1;
               ld_Xr<=1;
               ld_Yr<=1;
               ld_Xr_aftr_subs<=0;
               ld_Yr_aftr_subs<=0;
               lShift<=0;
            end
S2:
            begin
               ld_in1_in2<=0;
               update_Xr<=0;
               update_Yr<=1;
               update_Counter<=0;
               incr_dec<=0;
               ld_Xr<=0;
               ld_Yr<=1;
               ld_Xr_aftr_subs<=0;
               ld_Yr_aftr_subs<=0;
               lShift<=0;
            end
         S3:
            begin
               ld_in1_in2<=0;
               update_Xr<=1;
               update_Yr<=0;
               update_Counter<=0;
               incr_dec<=0;
               ld_Xr<=1;
               ld_Yr<=0;
               ld_Xr_aftr_subs<=0;
               ld_Yr_aftr_subs<=0;
               lShift<=0;
            end
```

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.

Thank You