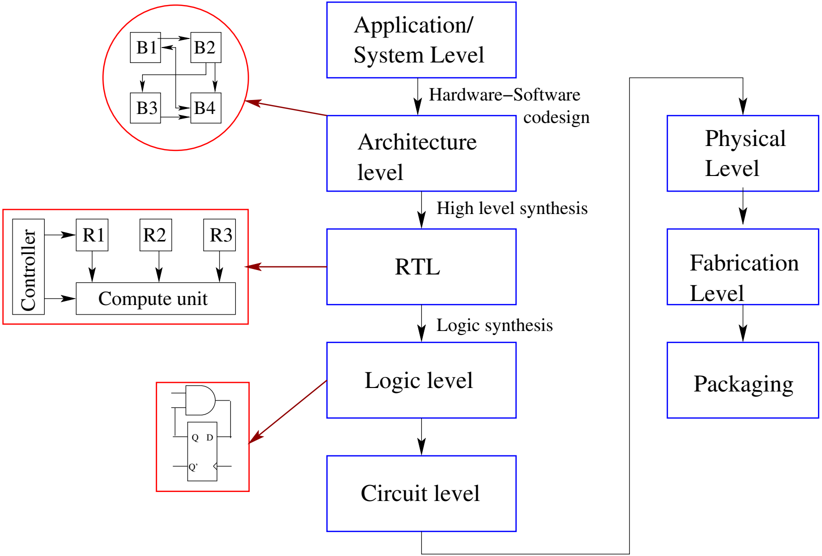
# Verilog: Hardware Description Language

Arijit Mondal
Dept. of CSE

# Digital CAD flow



Application/ System Level

Hardware–Software codesign

Architecture level

High level synthesis

RTL

Logic synthesis

Logic level

Circuit level

Physical Level

Fabrication Level

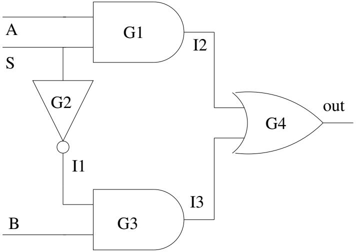
Packaging

# Introduction

- Verilog was designed primarily for digital hardware designers developing FPGAs and ASICs.
- It is different from C/C++/Java.
- Uses discrete event simulation techniques
- This is one of the most commonly used languages for describing hardware. Other popular language is VHDL
- Supports both structural as well as behavioral description

# Structural vs Behavioral

- Structural
  - ∗ Connectivities of gates



- Behavioral
  - ∗ Behavior of module
  - ∗ $out = SA + \bar{S}B$

# Concept of `module`

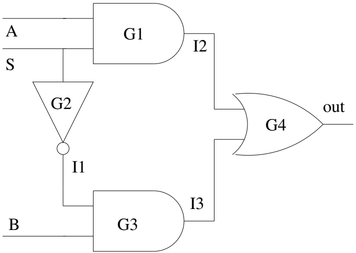
- Basic unit is module that describes a hardware component
  - ∗ Modules cannot contain definitions of other modules.
  - ∗ A module can, however, be instantiated within another module.
  - ∗ Allows the creation of a hierarchy in a Verilog description.

# Syntax of module definition

```
module module_name(list_of_ports);
  input /output declaration;
  local net declaration;
  statements;
endmodule
```

# MUX : Structural form

```
module MUX(out, A, B, S);
  output out;
  input A, B, S;
  wire I1, I2, I3;

  and G1(I2, A, S);
  and G3(I3, I1, B);
  not G2(I1, S);
  or G4(out, I2, I3);

endmodule
```

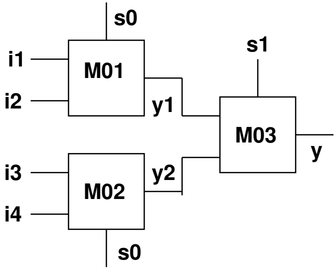


# Primitive gates

- `and, or, nand, not, nor, xor, xnor, buf`
- Syntax: `<gate_name> <inst_name> (<out>,<in1>,<in2>);`
- Syntax: `<gate_name> <inst_name> (<out>,<in1>);`

# Hierarchical design

```
module mux4x1(y,i1,i2,i3,i4,s0,s1);
input i1,i2,i3,i4,s0,s1;
output y;

MUX MUX_01(y01,i1,i2,s0);
MUX MUX_02(y02,i3,i4,s0);
MUX MUX_03(y,y01,y02,s1);

endmodule
```



# Specifying connectivity

- There are two alternate ways of specifying connectivity:
  - ∗ Positional association
    - — The connections are listed in the same order

      ```
      add A1 (c_out, sum, a, b, c_in);
      ```

  - ∗ Explicit association
    - — May be listed in any order

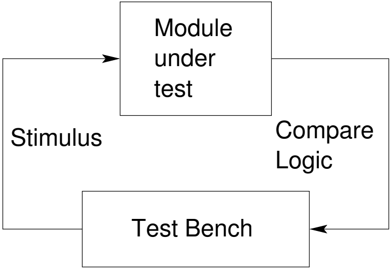
      ```
      add A1 (.in1(a), .in2(b), .cin(c_in), .sum(sum), .cout(c_out));
      ```

# Port connectivity

```
module subMod(out1,out2,in1,in2);
...
endmodule

module Mod;

//positional association
subMod M1(O1,O2,I1,I2);

//explicit association
subMod M2(.out1(O1),.out2(O2),.in1(I1),.in2(I2));

endmodule
```

# Testbench

- A procedural block which executes only once
- Used for simulation
- Generates clock, reset, and the required test vectors



# How to write testbench

- Create a dummy module
  - ∗ Declare inputs to the module-under-test (MUT) as 'reg' and the outputs as 'wire'
  - ∗ Instantiate the MUT
- Initialization
  - ∗ Assign some known values to the MUT inputs
- Generate the clock
- Use simulator directives to print the results

# Testbench

```
module top;
  reg a,b,c,d;
  reg s1, s2;
  wire y;
  mux4x1 mux_inst(y,a,b,c,d,s1,s2);

  initial begin
    a=b=c=d=s1=s2=1'b0;
    $monitor("y=%b a=%b b=%b c=%b d=%b s0=%b s1=%b"
            time=%2d,y,a,b,c,d,s0,s1,$time);
    #1 a=1'b1; #1 s1=1'b1; #1 a=1'b0;
    #1 c=1'b1; #1 s0=1'b1;
    #10 $finish;
  end
endmodule
```

# Logic values

- 0,1 – Binary value
- X – Unknown value
- Z – High impedance state

– All unconnected nets are set to 'z'

– All registers variables are set to 'x'

# Functional table for primitive gates

| AND | | | | |
|---|---|---|---|---|
| | 0 | 1 | x | z |
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | x | x |
| x | 0 | x | x | x |
| z | 0 | x | x | x |

| XOR | | | | |
|---|---|---|---|---|
| | 0 | 1 | x | z |
| 0 | 0 | 1 | x | x |
| 1 | 1 | 0 | x | x |
| x | x | x | x | x |
| z | x | x | x | x |

# Verilog operator

- Arithmetic operator: *,/,+,-
- Logical operator: !, &&, ||
- Relational operator: >, <, >=, <=, ==, ! =
- Bitwise operator: &,|, etc.
- Reduction operator: Operate on all the bits within a word (&, ˜&, etc.)
- Shift operator: <<, >>
- Concatenation operator: {}
- Replication operator: {{}}
- Conditional: expr? exp1 : exp2;

# Data types

- Net: Represents the continuous updating of outputs with respect to their changing inputs.
  - wire, supply0, supply1
  - `wire wire01,wire02;`
- reg: Has to be assigned values explicitly. Value is held until a new assignment is made
  - `reg out1, out2;`
- Integer: `integer intparam;`
- Real: `real realparam;`
- Vector: `reg [3:0] output;`
- Arrays: `reg data [7:0];`
- 1K memory of 16 bit elements –

  `reg [15:0] mem16_1024 [0:1023]`

# Data type: `wire`

- wire elements are used to connect input and output ports of a module instantiation together with some other element in your design.
- wire elements are used as inputs and outputs within an actual module declaration.
- wire elements must be driven by something, and cannot store a value without being driven.
- wire elements cannot be used as the left-hand side of an $=$ or $<=$ sign in an always@ block.
- wire elements are the only legal type on the left-hand side of an assign statement.
- wire elements are a stateless way of connecting two pieces in a Verilog-based design.
- wire elements can only be used to model combinational logic.
- `wire, wor, wand, tri, supply0, supply1`

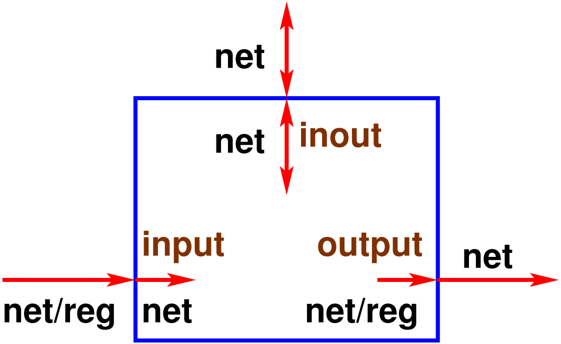
# Data type: `reg`

- It can be connected to the input port of a module instantiation.
- It cannot be connected to the output port of a module instantiation.
- It can be used as outputs within an actual module declaration.
- It cannot be used as inputs within an actual module declaration.
- It is the only legal type on the left-hand side of an always@ block $=$ or $<=$ sign.
- It is the only legal type on the left-hand side of an initial block $=$ sign (used in Test Benches).
- It cannot be used on the left-hand side of an assign statement.
- It can be used to create registers when used in conjunction with always@(posedge Clock) blocks.
- It can, therefore, be used to create both combinational and sequential logic.

# Data type: `reg`

- Different ‘register’ types supported for synthesis:
  - ∗ reg, integer
- The ‘reg’ declaration explicitly specifies the size.
  - ∗ `reg x, y; // single-bit register variables`
  - ∗ `reg [15:0] bus; // 16-bit bus, bus[15] MSB`
- For ‘integer’, it takes the default size, usually 32-bits.
  - ∗ Synthesizer tries to determine the size.
- Arithmetic expressions
  - ∗ An ‘integer’ is treated as a 2’s complement signed integer.
  - ∗ A ‘reg’ is treated as an unsigned quantity.
- General rule of thumb
  - ∗ reg used to model actual hardware registers such as counters, accumulator, etc.
  - ∗ integer used for situations like loop counting.

# Port connectivity



# Specifying const values

- Values can be specified in sized or unsized form
  - ∗ Syntax for sized form: `<size>'<base><number>`
- Examples
  - ∗ `4'b0011 //4-bit binary number`
  - ∗ `12'hA2D //hexadecimal number`
  - ∗ `12'hCx5 //1100 xxxx 0101 in binary`
  - ∗ `25 //signed number 32 bits`
  - ∗ `1'b0 //Logic 0`
  - ∗ `1'b1 //Logic 1`

# Parameters

- A parameter is a constant with name
- No size is allowed to be specified for a parameter
- The size gets decided from the constant itself. 32 bits if nothing is specified.
- Examples
  - `parameter HI = 25, LO = 5;`
  - `parameter up = 1'b0;`

# Note

- For all primitive gates:
  - ∗ The output port must be connected to a net.
  - ∗ The input port must be connected to a net or reg.
  - ∗ Can have only single output but any number of inputs

- Boolean true/false
  - ∗ true is equivalent to 1'b1;
  - ∗ false is equivalent to 1'b0;

# Simple AND gate

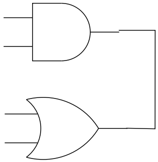
```
module simple_and(out, A, B);
  input A, B;
  output out;
  assign out = A & B;
endmodule
```

# Two level circuits

```
module two_level (a, b, c, d, f);
input a, b, c, d;
output f;
wire t1, t2;
  assign t1 = a & b;
  assign t2 = ~(c | d);
  assign f = t1 ^ t2;
endmodule
```

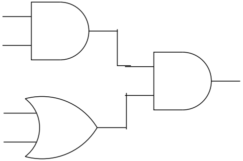
# Example: wire

```
module using_wire (A, B, C, D, f);
  input A, B, C, D;
  output f;
  // net f declared as ‘wire’
  wire f;
  assign f = A & B;
  assign f = C | D;
endmodule
```



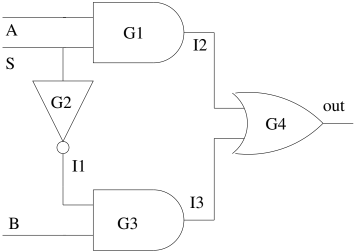
# Example: wire

```
module using_wire (A, B, C, D, f);
  input A, B, C, D;
  output f;
  // net f declared as ‘wand’
  wand f;
  assign f = A & B;
  assign f = C | D;
endmodule
```



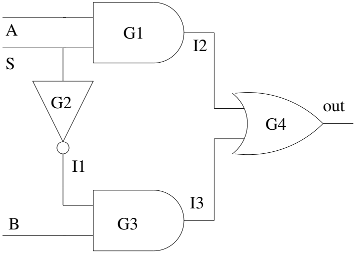
# MUX : Behavioral form

```
module MUX(out, A, B, SEL);
  output out;
  reg out;
  input A, B, SEL;

  always @(SEL, A, B)
    if(S)
       out=A;
    else
       out=B;

endmodule
```



# MUX : Continuous assignment

```
module MUX(out, A, B, SEL);
  output out;
  input A, B, SEL;

  assign out = SEL ? A : B;

endmodule
```



# 8-bit adder

```
module adder(sum,cout,in1,in2,cin);
  input [7:0] in1, in2;
  input cin;
  output [7:0] sum;
  output cout;
  assign #20 {cout,sum} = in1 + in2 + cin;
endmodule
```

# Description styles

- Data flow
  - Continuous assignment
- Behavioral
  - Procedural assignment
    - Blocking
    - Non-blocking

# Description styles: Continuous assignment

- Identified by the word 'assign'
- `assign a = b & c;`
- The 'net' is being assigned on the LHS, i.e. LHS must be of 'net' type
- Expression is on the RHS. RHS may contain both 'reg' or 'net'.
- Assignment is continuously active
- Exclusively used to model combinational logic
- A module can contain any number of continuous assignment statements

# Behavioral style: Procedural assignment

- A region of code containing sequential statements
- The statements execute in the order they are written
- Two types of procedural blocks
  - ∗ 'always' – Continuous loop that never terminate
  - ∗ 'initial' – Executed once at the beginning of simulation

# initial block

- An initial block consists of a statement or a group of statements enclosed in **begin ... end** which will be executed only once at simulation time 0.
- Multiple initial blocks in an design are executed in parallel
- Normally used for initialization, monitoring, generating wave forms (eg, clock pulses) and processes which are executed once in a simulation.

# Procedural assignment: always

- A module can contain any number of 'always' block
- Syntax for 'always' block

```
always @(event_expression)
begin
  statement
end
```

- @(event_expression) is required for both combinational and sequential logic

# only ’reg’ can be assigned within ’always’

- ’always’ block executes only when the event expression triggers
- At other time block is doing nothing
- An object being assigned to must therefore remember the last value assigned
- any kind of variable may appear in the event expression

# Sequential statements

-
  ```
  begin
    sequential_statements
  end
  ```

-
  ```
  if(expression)
    sequential_statement
  [else
    sequential_statement]
  ```

-
  ```
  case(expression)
    expr:  sequential_statement
    default:  sequential_statement
  endcase
  ```

# Sequential statements (contd)

-
  ```
  forever
    sequential_statement
  ```

-
  ```
  repeat(expression)
    sequential_statement
  ```

-
  ```
  while(expression)
    sequential_statement
  ```

-
  ```
  for(expr1;expr2;expr3)
    sequential_statement
  ```

# Blocking vs Non-blocking

- Sequential statements within procedural blocks can use two types of assignments
  - ∗ Blocking assignment : Use the '=' operator
  - ∗ Nonblocking assignment : Use the '<=' operator

# Blocking

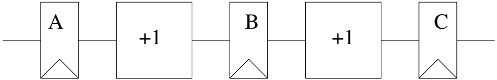
- Most commonly used type
- The target of assignment gets updated before the next sequential statement in the procedural block
- It blocks the execution of the statements following it
- Recommended style for modeling combinational logic

# Nonblocking

- The assignment to the target gets scheduled for the end of the simulation cycle
  - ∗ Normally occurs at the end of the sequential block
  - ∗ Statement subsequent to the instruction under consideration are not blocked by the assignment
- Recommended style for modeling sequential logic
  - ∗ Can be used to assign several 'reg' type variable synchronously, under the control of a common clock.
- A variable cannot appear as the target of both blocking and a nonblocking assignment

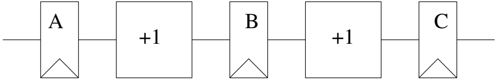
# Example

```
reg aout, bout, cout;
wire ain, bin, cin;
always @(negedge clk)

begin
    aout<=ain;
    bout<=aout+1;
    cout<=bout+1;
end
```



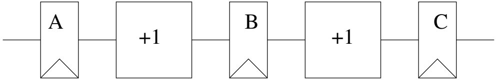
# Example

```
reg aout, bout, cout;
wire ain, bin, cin;

always @(negedge clk)
begin
    aout<=ain;
    bout<=bin;
    cout<=cin;
end

assign bin=aout+1;
assign cin=bout+1;
```



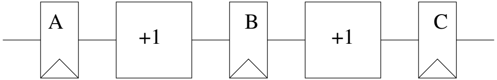
# Example

```
reg aout, bout, cout;
wire ain, bin, cin;

always @(negedge clk)
    aout<=ain;

assign bin=aout+1;

always @(negedge clk)
    bout<=bin;

assign cin=bout+1;

always @(negedge clk)
    cout<=cin;
```



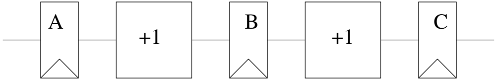
# Example

```
reg aout, bout, cout;
wire ain, bin, cin;

always @(negedge clk)
begin
    aout=ain;
    bout=bin;
    cout=cin;
end

assign bin=aout+1;
assign cin=bout+1;
```



# Example

```
reg aout, bout, cout;
wire ain, bin, cin;

always @(negedge clk)
begin
    aout=ain;
    bout=bin;
    cout=cin;
end

assign bin=aout+1;
assign cin=bout+1;
```



NOT CORRECT

# Counter design

```
module counter(out,en,reset,clk);
  output [3:0] out;
  reg     [3:0] out;
  input   en,reset,clk;
  wire    en,reset,clk;

  always @(posedge clk)
  begin :COUNTER
    if(reset == 1'b1) out = 4'b0000;
    else if(en == 1'b1) out = out + 1;
  end // end of COUNTER

endmodule
```

# Clock generator

```
module cklGen(out);
  output out;
  reg     out;

  initial begin
    out = 1'b0;
  end

  always
  begin :CLK
    out = #1 ~out;
  end // end of CLK

endmodule
```

# Testbench

```
module top;
  wire clk;
  clkGen M1(clk);
  counter M2(out,reset,en,clk);

  initial begin
    $monitor("out=%d time=%d\n",out,$time);
    reset=1’b0; en=1’b0;
    #5 en=1’b1;
    #100 $finish;
  end
endmodule
```

# Simulator directives

- $display – Similar to printf
- $monitor – Similar to $display, but prints whenever the value of some variables in the given list changes
- $finish – Stop simulation
- $dumpfile, $dumpvar, $readmemb, $readmemh

# Synchronous up-down counter

```
module counter (mode, clr, ld, d_in, clk, count);
  input mode, clr, ld, clk; input [0:7] d_in;
  output [0:7] count;
  reg [0:7] count;
  always @ (posedge clk)
    if(ld)
      count <= d_in;
    else if(clr)
      count <= 0;
    else if(mode)
      count <= count + 1;
    else
      count <= count – 1;
endmodule
```

# Multiple clocks

```
module multiple_clk (clk1, clk2, a, b, c, f1, f2);
  input clk1, clk2, a, b, c;
  output f1, f2;
  reg f1, f2;

  always @ (posedge clk1)
    f1 <= a & b;

  always @ (negedge clk2)
    f2 <= b ^ c;

endmodule
```

# Multiple edges of the clk

```
module multi_phase_clk (a, b, f, clk);
  input a, b, clk;
  output f;
  reg f, t;

  always @ (posedge clk)
    f <= t & b;

  always @ (negedge clk)
    t <= a | b;

endmodule
```

# Ring counter

```verilog
module ring_counter (clk, init, count);
  input clk, init;
  output [7:0] count;
  reg [7:0] count;
  always @ (posedge clk)
  begin
    if(init)
      count = 8’b10000000;
    else begin
      count = count << 1;
      count[0] = count[7];
    end
  end
endmodule
```

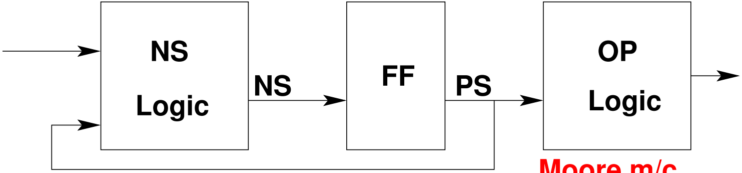
# Ring counter (contd.)

```
module ring_counter (clk, init, count);
  input clk, init;
  output [7:0] count;
  reg [7:0] count;
  always @ (posedge clk)
  begin
    if(init)
      count = 8’b10000000;
    else begin
      count <= count << 1;
      count[0] <= count[7];
    end
  end
endmodule
```

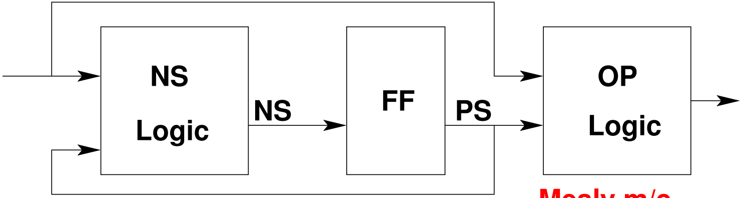
# Ring counter (contd.)

```
module ring_counter (clk, init, count);
  input clk, init;
  output [7:0] count;
  reg [7:0] count;
  always @ (posedge clk)
  begin
    if(init)
      count = 8'b10000000;
    else begin
      count = {count[6:0],count[7]};
    end
  end
endmodule
```

# Finite State Machine



Moore m/c



Mealy m/c

# FSM: Traffic Light Controller

- Simplifying assumptions made
- Three lights only (RED, GREEN, YELLOW)
- The lights glow cyclically at a fixed rate
  - ∗ Say, 10 seconds each
  - ∗ The circuit will be driven by a clock of appropriate frequency

# Traffic light controller

```
module traffic_light (clk, light);
input clk;
output [0:2] light;
reg [0:2] light;
parameter S0=0, S1=1, S2=2;
parameter RED=3'b100, GREEN=3'b010,
YELLOW=3'b001;
reg [0:1] state;
```

# Traffic light controller

```
module traffic_light (clk, light);
input clk;
output [0:2] light;
reg [0:2] light;
parameter S0=0, S1=1, S2=2;
parameter RED=3'b100, GREEN=3'b010,
YELLOW=3'b001;
reg [0:1] state;
always @ (posedge clk)
case(state)
  S0:  begin //RED
    light <= YELLOW;
    state <= S1;
  end
```

# Traffic light controller

```
module traffic_light (clk, light);
input clk;
output [0:2] light;
reg [0:2] light;
parameter S0=0, S1=1, S2=2;
parameter RED=3'b100, GREEN=3'b010,
YELLOW=3'b001;
reg [0:1] state;
always @ (posedge clk)
case(state)
  S0:  begin //RED
    light <= YELLOW;
    state <= S1;
  end
```

```
S1:  begin //YELLOW
  light <= GREEN;
  state <= S2;
end
```

# Traffic light controller

```
module traffic_light (clk, light);
input clk;
output [0:2] light;
reg [0:2] light;
parameter S0=0, S1=1, S2=2;
parameter RED=3’b100, GREEN=3’b010,
YELLOW=3’b001;
reg [0:1] state;
always @ (posedge clk)
case(state)
  S0:  begin //RED
    light <= YELLOW;
    state <= S1;
  end
S1:  begin //YELLOW
  light <= GREEN;
  state <= S2;
end
S2:  begin //GREEN
  light <= RED;
  state <= S0;
end
```

# Traffic light controller

```
module traffic_light (clk, light);
input clk;
output [0:2] light;
reg [0:2] light;
parameter S0=0, S1=1, S2=2;
parameter RED=3'b100, GREEN=3'b010,
YELLOW=3'b001;
reg [0:1] state;
always @ (posedge clk)
case(state)
  S0:  begin //RED
    light <= YELLOW;
    state <= S1;
  end
```

```
S1:  begin //YELLOW
  light <= GREEN;
  state <= S2;
end
S2:  begin //GREEN
  light <= RED;
  state <= S0;
end
default:  begin
  light <= RED;
  state <= S0;
end
endcase
endmodule
```

# Traffic light controller (contd.)

```
module traffic_light_nonlatched_op
(clk, light);
input clk; output [0:2] light;
reg [0:2] light;
parameter S0=0, S1=1, S2=2;
parameter RED=3'b100,
GREEN=3'b010, YELLOW=3'b001;
reg [0:1] state;
```

# Traffic light controller (contd.)

```
module traffic_light_nonlatched_op
(clk, light);
input clk; output [0:2] light;
reg [0:2] light;
parameter S0=0, S1=1, S2=2;
parameter RED=3'b100,
GREEN=3'b010, YELLOW=3'b001;
reg [0:1] state;
always @ (posedge clk)
case(state)
  S0:  state <= S1;
  S1:  state <= S2;
  S2:  state <= S0;
  default:  state <= S0;
endcase
```

# Traffic light controller (contd.)

```
module traffic_light_nonlatched_op
(clk, light);
input clk; output [0:2] light;
reg [0:2] light;
parameter S0=0, S1=1, S2=2;
parameter RED=3'b100,
GREEN=3'b010, YELLOW=3'b001;
reg [0:1] state;
always @ (posedge clk)
case(state)
  S0:  state <= S1;
  S1:  state <= S2;
  S2:  state <= S0;
  default:  state <= S0;
endcase
```

```
always @ (state)
case(state)
  S0:  light = RED;
  S1:  light = YELLOW;
  S2:  light = GREEN;
  default:  light = RED;
endcase
endmodule
```

# Verilog: `function` & `task`

- These allow the designers to abstract Verilog code that is used at many places in the design
- `function`
  - ∗ Can enable other functions but no task
  - ∗ Executes in 0 simulation time
  - ∗ Must have at least one input argument
  - ∗ Returns a single value
- `task`
  - ∗ Can enable other functions and tasks
  - ∗ May execute in non-zero simulation time
  - ∗ May have zero or more argument
  - ∗ Does not return value but can pass value through inout or output

# Example: Task

```
module op;
...
always @(...)  begin
  ...
  bitwiseOpr(AB_AND,A,B)
end

task bitwiseOpr
input A,B;
output AB_AND
begin
  AB_AND = A & B;
end
endtask
endmodule
```

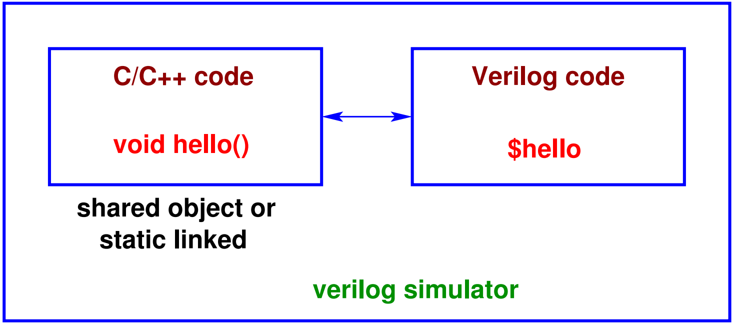
# Example: Function

```
module op;
reg X;
always @(...)  begin
  ...
  X=bitwiseOpr(A,B)
end

function bitwiseOpr;
input A,B;
begin
  bitwiseOpr = A & B;
end
endfunction
...
endmodule
```

# Verilog PLI

- The Verilog PLI is a simulation interface
  - ∗ Reads/modifies simulation data structures
  - ∗ Does not read Verilog source code
  - ∗ Does not work with synthesis compilers or other tools
- The PLI is a protecting layer between user programs and simulation data structure
  - ∗ Indirect access through PLI libraries
    - — C program cannot directly read or modify the simulation data
    - — Protects the simulator from bad C programs
    - — Protects C programs from bad Verilog code

# Verilog PLI


C/C++ code
void hello()
Verilog code
$hello
shared object or
static linked
verilog simulator

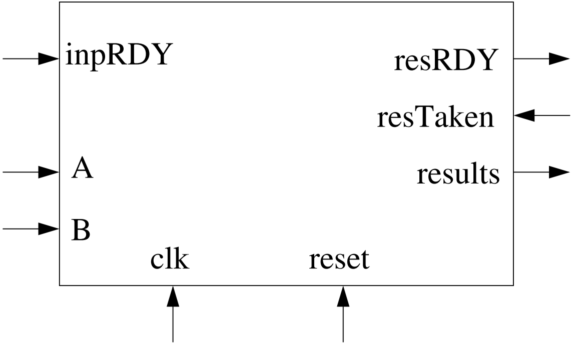
# Example: GCD in C

```
int GCD(int A, int B){
  int done=0; int tmpA = A, tmpB = B;
  while(done != 1){
    if(A < B){
      int tmp = B; B = A; A = tmp;
    }else if(B != 0){
      A = A - B;
    }else{
      done = 1;
    }
  }
  return A;
}
```

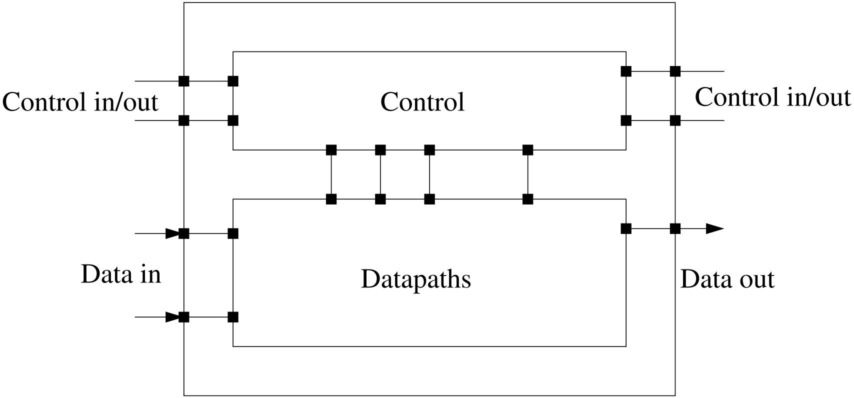
# Example: GCD in verilog

```
module GCD#(parameter W=8) (out, A, B);
  input [w-1:0] A, B; output [w-1:0] out;
  reg [w-1:0] tmpA, tmpB, out, swap; integer done;
  always @(*)   begin
    done=0; tmpA = A; tmpB = B;
    while(!done)   begin
      if(tmpA<tmpB)
        swap = tmpA; tmpA = tmpB; tmpB = swap;
      else if(tmpB!=0)
        tmpA = tmpA - tmpB;
      else
        done = 1;
    end
    out = tmpA
  end
endmodule
```

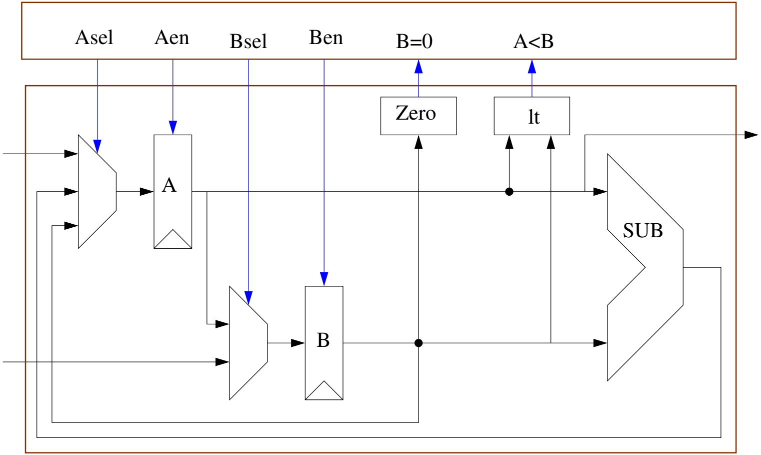
# GCD: Top level view



# GCD: Datapath & Control module



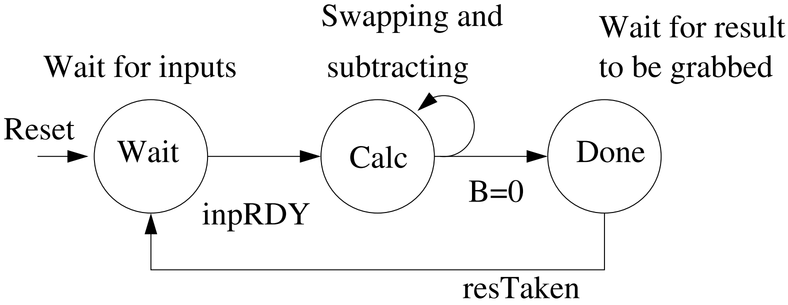
# GCD: Datapath & Control module



# GCD: Datapath module

```
module GCDDP#(parameter W=8) (.....){
  input [w-1:0] A, B; output [w-1:0] results;
  input Aen, Ben, Asel, Bsel;
  output bZero, A_lt_B;
  input clk;

  AMUX();
  AFF();
  BMUX();
  BFF();
  B_EQ_0();
  LessThan();
  Subtractor();

endmodule
```

# GCD: FSM


Swapping and
subtracting
Wait for result
to be grabbed
Wait for inputs
Reset
Wait
Calc
Done
inpRDY
B=0
resTaken

# GCD: Control module

```
module GCDControl (.....)
  input inpRDY, resTaken; output resRDY;
  output Aen, Ben, Asel, Bsel;
  input bZero, A_lt_B;
  input clk, reset;

  reg nextState; wire state
  DFF(nextState -> state);
  always @(*)
    WAIT: set various signal lines
    CALC: set various signal lines
    DONE: set various signal lines
  endcase
```

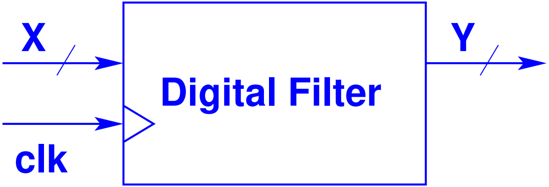
# GCD: Control module

```
  always @(*)
  case(state)
    WAIT: if(inpRDY) nextState = CALC
    CALC: if(bZero) nextState = DONE
    DONE: .....
    default:  nextState = state
  endcase
endmodule
```

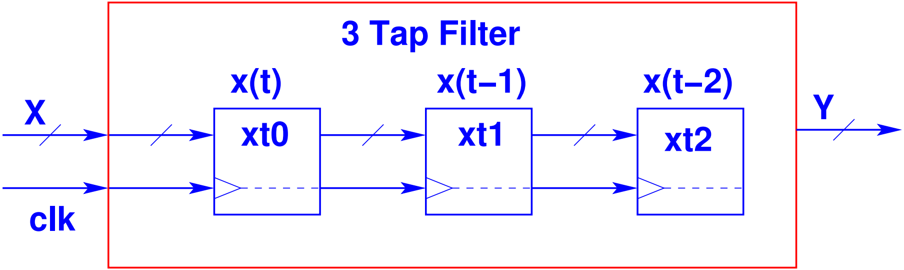
# FIR Filter

- Let us consider car engine temperature ($X$) in every second: 180, 181, 180, 240, 180, 181
- 240 is spurious value, needs to be ignored
- $Y(t) = c_0 \times x(t) + c_1 \times x(t-1) + c_2 \times x(t-2)$

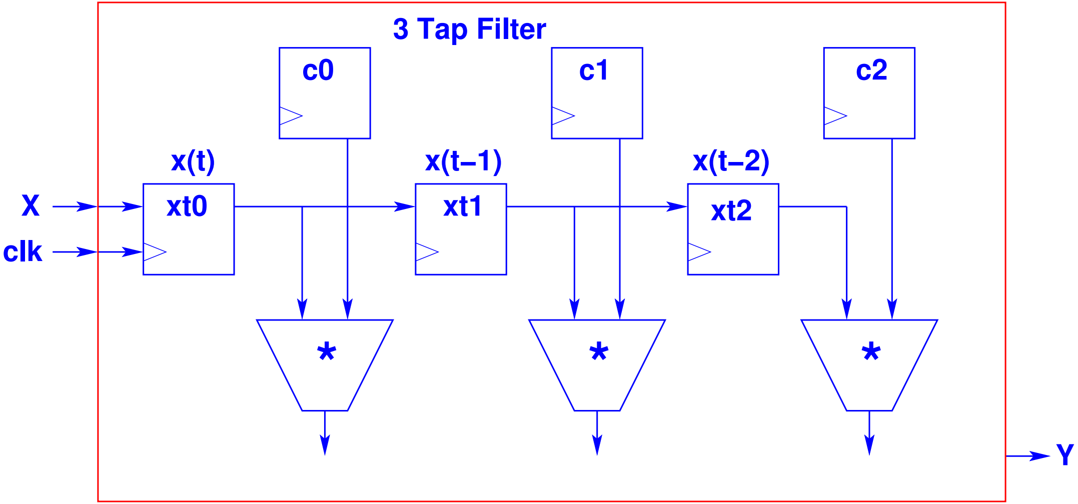
# FIR Filter



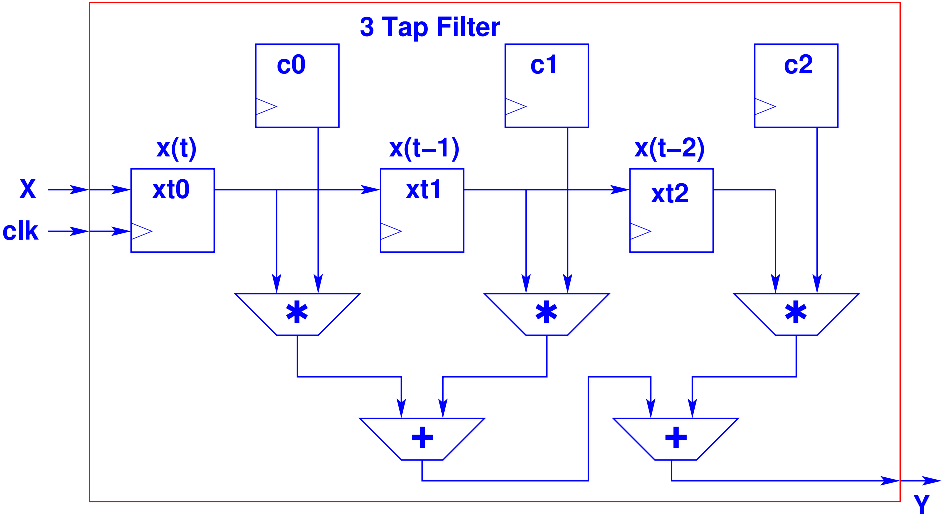
# FIR Filter



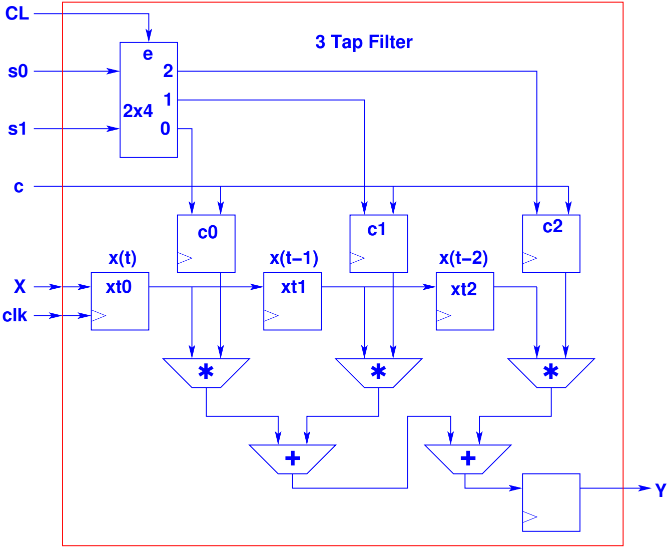
# FIR Filter



# FIR Filter



# FIR Filter


CL
s0
s1
c
X
clk
3 Tap Filter
e
2
1
0
2x4
c0
c1
c2
x(t)
x(t–1)
x(t–2)
xt0
xt1
xt2
*
*
*
+
+
Y

Thank you!