VERILOG TUTORIAL

VLSI II

E. Özgür ATES

Outline

- Introduction
- Language elements
- Gate-level modeling
- Data-flow modeling
- Behavioral modeling
- Modeling examples
- Simulation and test bench

Hardware Description Language

- Have high-level language constructs to describe the functionality and connectivity of the circuit
- Can describe a design at some levels of abstraction
  - Behavioral (Algoritmic, RTL), Structural (Gate-level), Switch

(For example, an HDL might describe the layout of the wires, resistors and transistors on an Integrated Circuit (IC) chip, i. e. the switch level. Or, it might describe the logical gates and flip flops in a digital system, i. e., the gate level. An even highe level describes the registers and the transfers of vectors of information between registers. This is called the Register Transfer Level (RTL). Verilog supports all of these levels.)

Hardware Description Language

- A design’s abstraction levels

– **Behavioral**

Algorithmic: A model that implements a design algorithm in high-level language constructs

RTL: A model that describes the flow of data between registers and how a design processes that data

– **Gate-level:** A model that describes the logic gates and the connections between logic gates in a design

– **Switch-level:** A model that describes the transistors and storage nodes in a device and the connections between them

Verilog Hardware Description Language

- Verilog was started initially as a proprietary hardware modeling language by Gateway Design Automation Inc. around 1984. It is rumored that the original language was designed by taking features from the most popular HDL language of the time, called HiLo, as well as from traditional computer languages such as C.

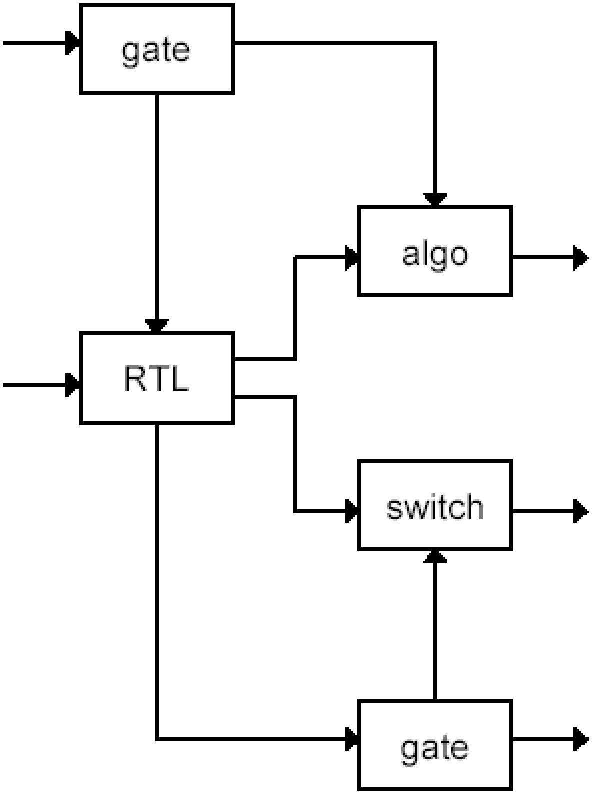
- Verilog simulator was first used beginning in 1985 and was extended substantially through 1987. The implementation was the Verilog simulator sold by Gateway. The first major extension was Verilog-XL, which added a few features and implemented the infamous "XL algorithm" which was a very efficient method for doing gate-level simulation.

Verilog Hardware Description Language

- The time was late 1990. Cadence Design System, whose primary product at that time included Thin film process simulator, decided to acquire Gateway Automation System. Along with other Gateway products, Cadence now became the owner of the Verilog language.

What is Verilog HDL ?

- Hardware description language
- Mixed level modeling
  - Behavioral
    - Algorithmic
    - Register transfer
  - Structural
    - Gate
    - Switch
- Single language for design and simulation
- Built-in primitives and logic functions
- User-defined primitives
- Built-in data types
- High-level programming constructs



Hardware Description Language

Hardware Description Languages describe the architecture and behavior of discrete and integrated electronic systems. Modern HDLs and their associated simulators are very powerful tools for integrated circuit designers.

Main reasons of important role of HDL in modern design methodology:

- Design functionality can be verified early in the design process. Design simulation at this higher level, before implementation at the gate level, allows you to evaluate architectural and design decisions.

Hardware Description Language

- Coupling HDL Compiler with logic synthesis tools, you can automatically convert an HDL description to a gate-level implementation in a target technology.

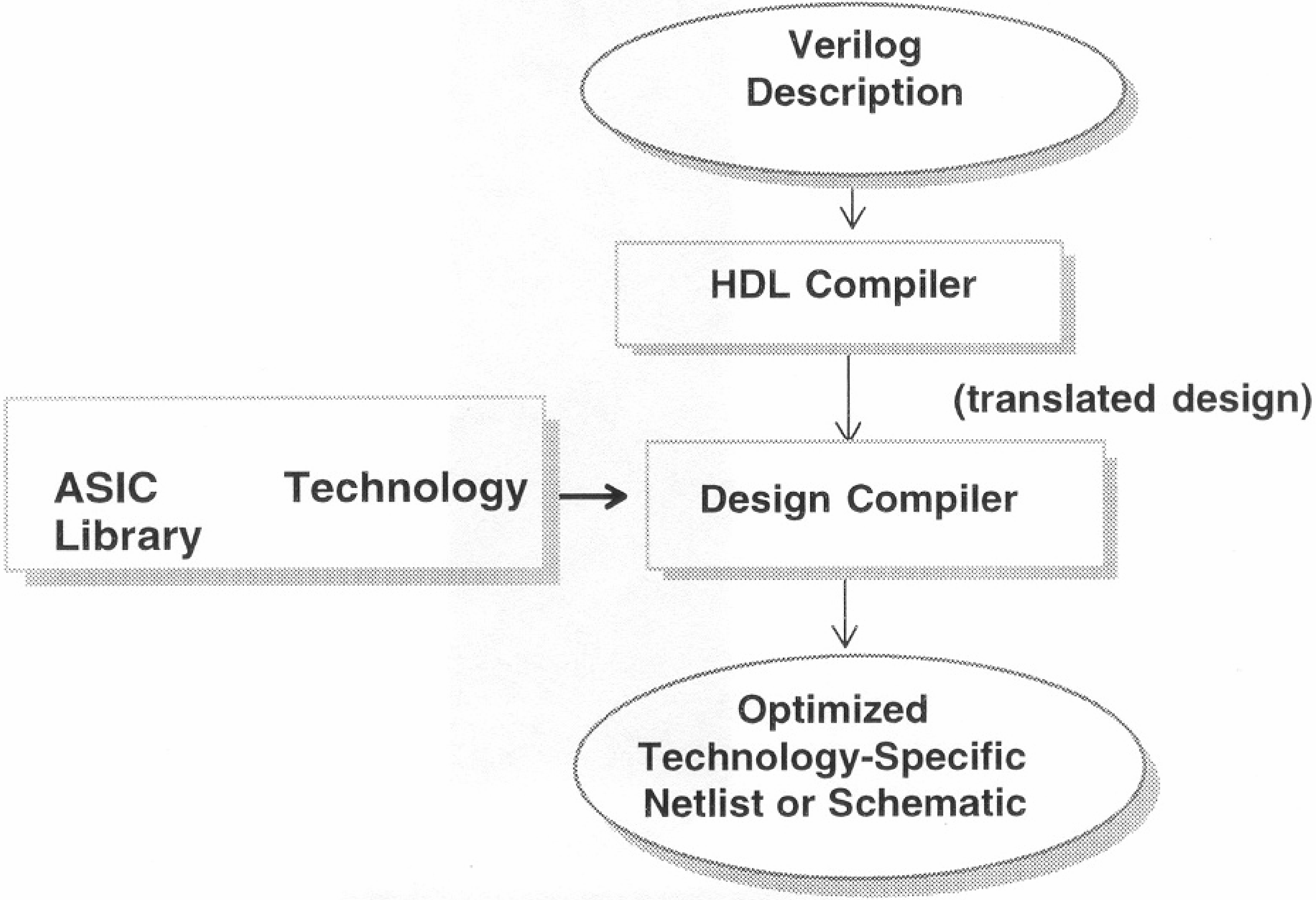
- HDL descriptions provide technology-independent documentation of a design and its functionality. Since the initial HDL design description is technology-independent, you can use it again to generate the design in a different technology, without having to translate from the original technology.

Hardware Description Language

Verilog digital logic simulator tools allow you to perform the following tasks in the design process without building a hardware prototype:

- **Determine the feasibility of new design ideas**
- **Try more than one approach to a design problem**
- **Verify functionality**
- **Identify design problems**

HDL Compiler and Design Compiler



Verilog Description

HDL Compiler

(translated design)

ASIC Technology Library

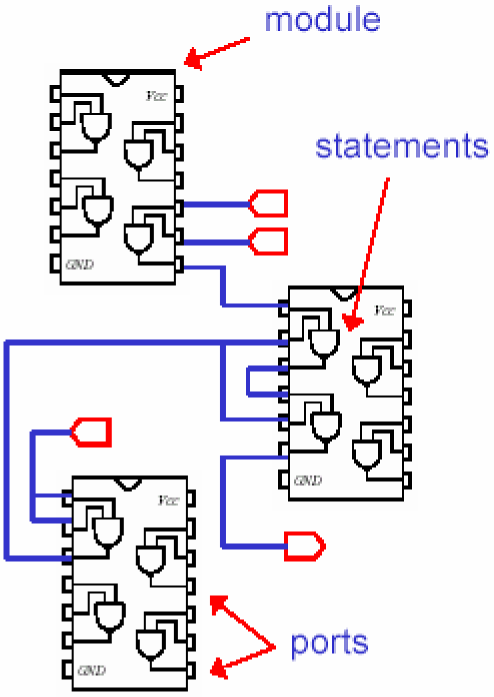
Design Compiler

Optimized Technology-Specific Netlist or Schematic

Basic Unit -- Module

- Modules communicate externally with input, output and bi-directional ports
- A module can be instantiated in another module

module module_name (port_list);
declarations:
    port declaration (input, output, inout, …)
    data type declaration (reg, wire, parameter, …)
    task and function declaration
statements:
    initial block
    always block
} Behavioral
    module instantiation
    gate instantiation
    UDP instantiation
} Structural
    continuous assignment
} Data-flow
endmodule



An Example

```
module FA_MIX (A, B, CIN, SUM, COUT);
   input A,B,CIN;
   output SUM, COUT;
   reg COUT;
   reg T1, T2, T3;
   wire S1;

   xor X1 (S1, A, B); // Gate instantiation.

   always @ (A or B or CIN)   // Always Block
   begin
      T1 = A & CIN;
      T2 = B & CIN;
      T3 = A & B;
      COUT = (T1 | T2 | T3);
   END
   assign SUM = S1 ^ CIN;   // Continuous assignment
endmodule
```

Structural Hierarchy Description Style

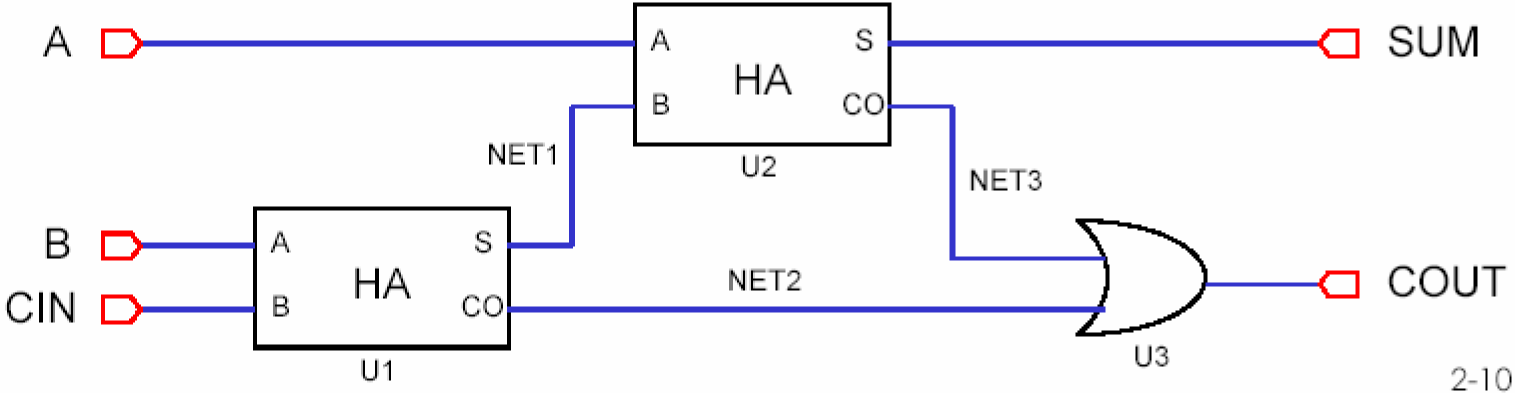
- Direct instantiation and connection of models from a separate calling model
  - Form the structural hierarchy of a design
- A module may be declared anywhere in a design relative to where it is called
- Signals in the higher “calling” model are connected to signals in the lower “called” model by either:
  - Named association
  - Positional association

Structural Hierarchy Description Style

- Example: (Full Adder)

```
module FULL_ADD (A, B, CIN, SUM, COUT);
    input A, B, CIN;
    output SUM, COUT;
    wire NET1, NET2, NET3;
    HA  U1(NET1, NET2, B, CIN);    /* positional association */
    HA  U2(.S(SUM), .CO(NET3), .A(A), .B(NET1));    /* named */
    OR2  U3(COUT, NET2, NET3);
endmodule
```

instance name



Lexical Conventions

- Verilog is a free-format language
  - Like C language
- White space (blank, tab, newline) can be used freely
- Verilog is a **case-sensitive** language
- Identifiers
  - User-provided names for Verilog objects in the descriptions
  - Legal characters are “a-z”, “A-Z”, “0-9”, “_”, and “$”
    - First character has to be a letter or an “_”
  - Example: Count, _R2D2, FIVE$
- Keywords
  - Predefined identifiers to define the language constructs
  - All keywords are defined in lower case
  - Cannot be used as idenfiers
  - Example: **initial, assign, module**

Lexical Conventions

- Comments: two forms

  /* First form: can
      extend over many
          lines */

  // Second form: ends at the end of this line

- Strings
  - Enclosed in double quotes and must be specified in one line
    - “Sequence of characters”
  - Accept C-liked escape character
    - \n = newline
    - \t = tab
    - \\ = backslash
    - \” = quote mark (“)
    - %% = % sign

Value Set

- 0: logic-0 / FALSE
- 1: logic-1 / TRUE
- x: unknown / don’t care, can be 0, 1 or z.
- z: high-impedance

Number Representation

<size><base format><number>

```
549 // decimal number
'h 8FF // hex number
'o765 // octal number
4'b11 // 4-bit binary number 0011
3'b10x // 3-bit binary number with least significant bit unknow
5'd3 // 5-bit decimal number
-4'b11 // 4-bit two's complement of 0011 or 1101
```

Data Types

- Nets
  - Connects between structural elements
  - Values come from its drivers
    - Continuous assignment
    - Module or gate instantiation
  - If no drivers are connected to net, default value is Z
- Registers
  - Represent abstract data storage elements
  - Manipulated within procedural blocks
  - The value in a register is saved until it is overridden
  - Default value is X

Net Types

- **wire, tri**: standard net
- **wor, trior**: wired-or net
- **wand, triand**: wired-and net
- **trireg**: capacitive
  - If all drivers at z, previous value is retained
- **tri1**: pull up (if no driver, 1)
- **tri0**: pull down (if no driver, 0)
- **supply0**: ground
- **supply1**: power
- A net that is not declared defaults to a 1-bit wire

```
wire reset;
wor [7:0] DBUS;
supply0 GND;
```

Register Types

- **reg**: any size, unsigned
- **integer**: 32-bit signed (2’s complement)
- **time**: 64-bit unsigned
- **real, realtime**: 64-bit real number
  - Defaults to an initial value of 0
- Examples:

```
reg CNT;
reg [31:0] SAT;
integer A, B, C;   // 32-bit
real SWING;
realtime CURR_TIME;
time EVENT;
```

Outline

- Introduction
- Language elements
- Gate-level modeling
- Data-flow modeling
- Behavioral modeling
- Modeling examples
- Simulation and test bench

Primitive Gates

- The following gates are built-in types in the simulator
- **and, nand, nor, or, xor, xnor**
  - First terminal is output, followed by inputs

    and a1 (out1, in1, in2);
    nand a2 (out2, in21, in22, in23, in24);
- **buf, not**
  - One or more outputs first, followed by one input

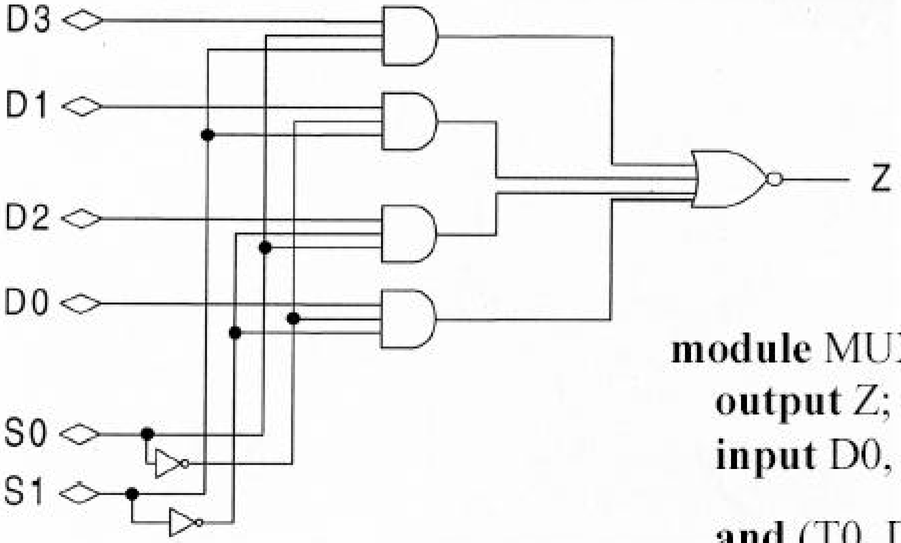
    not N1 (OUT1, OUT2, OUT3, OUT4, INA);
    buf B1 (BO1, BIN);
- **bufif0, bufif1, notif0, notif1**: three-state drivers
  - Output terminal first, then input, then control

    bufif1 BF1 (OUTA, INA, CTRLA);
- **pullup, pulldown**
  - Put 1 or 0 on all terminals

    pullup PUP (PWRA, PWRB, PWRC);
- Instance names are optional

  ex: **not** (QBAR, Q)

Example



4 X 1 multiplexer circuit

```
module MUX4x1 (Z, D0, D1, D2, D3, S0, S1);
   output Z;
   input D0, D1, D2, D3, S0, S1;

   and (T0, D0, S0BAR, S1BAR),
        (T1, D1, S0BAR, S1),
        (T2, D2, S0, S1BAR),
        (T3, D3, S0, S1);

   not (S0BAR, S0),
       (S1BAR, S1);

   nor (Z, T0, T1, T2, T3);
endmodule
```

Outline

- Introduction
- Language elements
- Gate-level modeling
- Data-flow modeling
- Behavioral modeling
- Modeling examples
- Simulation and test bench

Data-Flow Description Style

- Models behavior of combinational logic
- Assign a value to a net using ***continuous assignment***
- Examples:

```
wire [3:0] Z, PRESET, CLEAR;
assign Z = PRESET & CLEAR;

wire COUT, CIN;
wire [3:0] SUM, A, B;
assign {COUT, SUM} = A + B + CIN;
```

- Left-hand side (target) expression can be a:
  - Single net (ex: Z)
  - Part-select (ex: SUM[2:0])
  - Bit-select (ex: Z[1])
  - Concatenation of both (ex: {COUT, SUM[3:0]})
- Expression on right-hand side is evaluated whenever any operand value changes

Delays

- Delay between assignment of right-hand side to left-hand side

```
assign #6 ASK = QUIET || LATE;   //Continuous delay
```

- Net delay

```
wire #5 ARB;
// Any change to ARB is delayed 5 time units before it
  takes effect
```

- If value changes before it has a chance to propagate, latest value change will be applied
  - Inertial delay

Operators

Arithmetic Operators	+, -, *, /, %
Relational Operators	<, <=, >, >=
Logical Equality Operators	==, !=
Case Equality Operators	===, !==
Logical Operators	!, &&, ||
Bit-Wise Operators	~, &, |, ^(xor), ~^(xnor)
Unary Reduction Operators	&, ~&, |, ~|, ^, ~^
Shift Operators	>>, <<
Conditional Operators	? :
Concatenation Operator	{ }
Replication Operator	{ { } }

Outline

- Introduction
- Language elements
- Gate-level modeling
- Data-flow modeling
- Behavioral modeling
- Modeling examples
- Simulation and test bench

Behavioral Modeling

- Procedural blocks:
  - **initial block**: executes only once
  - **always block**: executes in a loop
- Block execution is triggered based on user-specified conditions
  - **always** @ (**posedge** clk) ………
- All procedural blocks are automatically activated at time 0
- All procedural blocks are executed **concurrently**
- **reg** is the main data type that is manipulated within a procedural block
  - It holds its value until assigned a new value

Parameter Statement

The parameter statement allows the designer to give a constant a name. Typical uses are to specify width of registers and delays. For example, the following allows the designer to parameterized the declarations of a model.

```
parameter byte_size = 8;

reg [byte_size - 1:0] A, B;
```

Initial Statement

- Executes only once at the beginning of simulation

```
initial
    statements
```

- Used for initialization and waveform generation

```
//Initialization:
reg [7:0] RAM[0:1023];
reg RIB_REG;

initial
begin
    integer INX;

    RIB_REG =0;
    for (INX = 0; INX < 1024; INX = INX + 1)
          RAM[INX] = 0;
end
```

group
multiple
statements

Always Statement

- Executes continuously; must be used with some form of timing control

```
always (timing_control)
   statements
```

```
always
   CLK = ~CLK
   // Will loop indefinitely
```

- Four forms of event expressions are often used
  - An **OR** of several identifiers (comb/seq logic)
  - The rising edge of a identifier (for clock signal of a register)
  - The falling edge of a identifier (for clock signal of a register)
  - Delay control (for waveform generator)
- Any number of *initial* and *always* statements may appear within a module
- Initial and always statements are all executed in parallel

Truth Table to Verilog

```
module COMB(A, B, Y1, Y2);
    input  A, B;
    output  Y1, Y2;
    reg  Y1, Y2;

  always @(A or B)
   begin
     case ({A, B})
      2'b 00 : begin Y1=1; Y2=0; end
      2'b 01 : begin Y1=1; Y2=0; end
      2'b 10 : begin Y1=1; Y2=0; end
      2'b 11 : begin Y1=0; Y2=1; end
    endcase
  end

endmodule
```

Any value changes of A or B will trigger this block

A	B	Y1	Y2
0	0	1	0
0	1	1	0
1	0	1	0
1	1	0	1

Other Examples

```
module example (D, CURRENT_STATE, Q, NEXT_STATE);

   input D, CURRENT_STATE;
   output Q, NEXT_STATE;
   reg CLK, Q, NEXT_STATE;

   always #5 CLK = ~CLK;

   always @(posedge CLK)
   begin
      Q = D;
   end

   always @(negedge CLK)
   begin
      NEXT_STATE = CURRENT_STATE;
   end

endmodule
```

delay-controlled always block
clock period = 10

activated when CLK has
a 0 -> 1 transition

activated when CLK has
a 1 -> 0 transition

Procedural Assignments

- The assignment statements that can be used inside an *always* or *initial* block
- The target must be a register or integer type
- The following forms are allowed as a target
  - Register variables
  - Bit-select of register variables (ex: A[3])
  - Part-select of register variables (ex: A[4:2])
  - Concatenations of above (ex: {A, B[3:0]})
  - Integers

```
always @(posedge CLK) begin
        B = A;
        C = B;
end
```

Conditional Statements

- *if* and *else if* statements

if (*expression*)
statements
{ **else if** (*expression*)
statements }
[**else**
statements]

```
if (total < 60) begin
    grade = C;
    total_C = total_C + 1;
end
else if (sum < 75) begin
    grade = B;
    total_B = total_B + 1;
end
else grade = A;
```

- *case* statement

case (*case_expression*)
case_item_expression
{, *case_item_expression* } :
statements
……
[**default**: *statements*]
endcase

```
case (OP_CODE)
   2`b10:      Z = A + B;
   2`b11:      Z = A – B;
   2`b01:      Z = A * B;
   2`b00:      Z = A / B;
   default:    Z = 2`bx;
endcase
```

Loop Statements

- Four loop statements are supported
  - The *for* loop
  - The *while* loop
  - The *repeat* loop
  - The *forever* loop
- The syntax of loop statements is very similar to that in C language
- Most of the loop statements are not synthesizable in current commercial synthesizers

```
for(i = 0; i < 10; i = i + 1)
begin
        $display("i= %0d", i);
end
```

```
i = 0;
while(i < 10)
begin
        $display("i= %0d", i);
        i = i + 1;
end
```

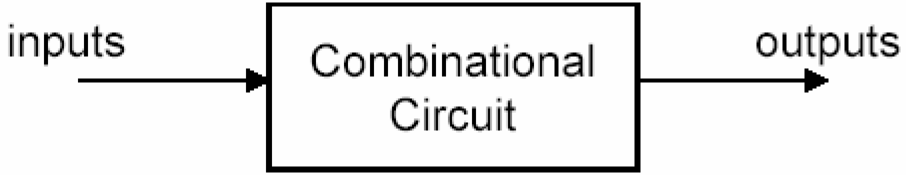
```
repeat (5)
begin
        $display("i= %0d", i);
         i = i + 1;
end
```

Outline

- Introduction
- Language elements
- Gate-level modeling
- Data-flow modeling
- Behavioral modeling
- Modeling examples
- Simulation and test benches

Combinational Circuit Design

- Outputs are functions of inputs



inputs

Combinational Circuit

outputs

- The sensitivity list must include all inputs

```
always @ (a or b or c)
  f = a & ~c | b & c ;
```

- Wrong example:

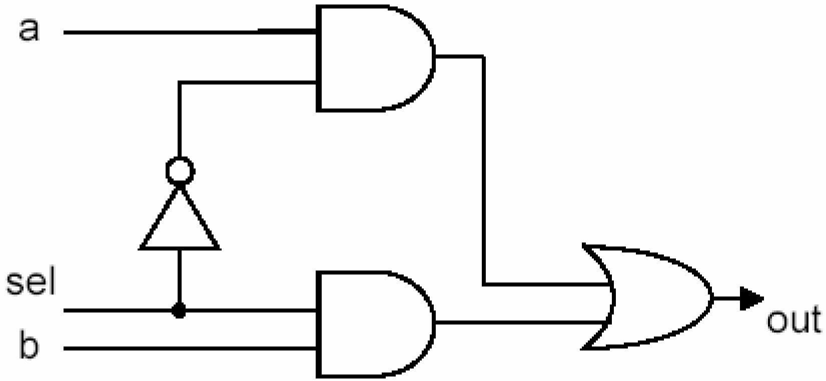
```
always @ (a or b)
  f = a & ~c | b & c ;
```

  - The changes of c will not change the output immediately
  - May cause functional mismatch in the synthesized circuits
    - Unlike simulation, synthesizers will skip the sensitivity list and deal with the following statements directly

An Example: Multiplexer

- Continuous assignment

```
module mux2_1(out,a,b,sel);
    output out;
    input a,b,sel;
    assign out = (a&~sel) |
                    (b&sel);
endmodule
```



- RTL modeling

```
module mux2_1(out,a,b,sel);
    output out;
    input a,b,sel;
    reg out;

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

endmodule
```

Multiplexer Example

Where is the register?

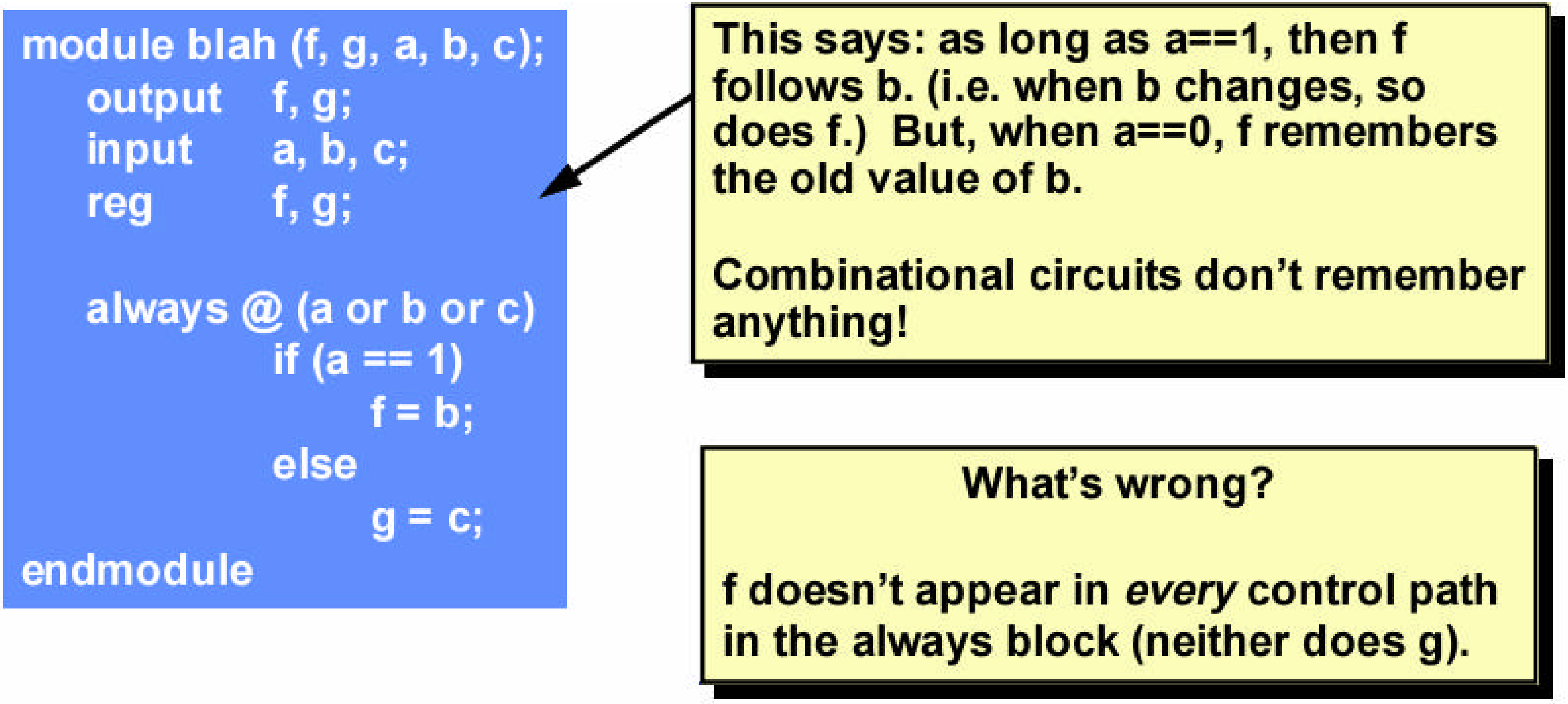
- The synthesis tool figures out that this is a combinational circuit. Therefore, it does not need a register.

How does it figure out that this is combinational?

- The output is only a function of the inputs (and not of previous values)

- Anytime an input changes, the output is re-evaluated.

Combinational Design Error



```
module blah (f, g, a, b, c);
    output    f, g;
    input     a, b, c;
    reg       f, g;

    always @ (a or b or c)
              if (a == 1)
                   f = b;
              else
                   g = c;
endmodule
```

This says: as long as a==1, then f follows b. (i.e. when b changes, so does f.) But, when a==0, f remembers the old value of b.

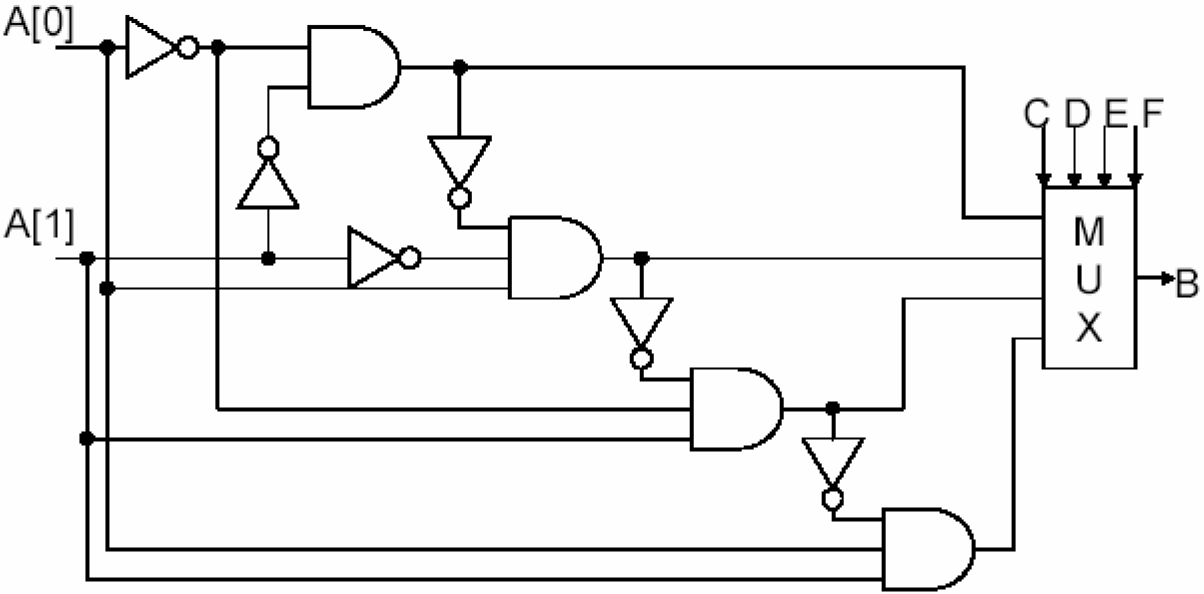
Combinational circuits don’t remember anything!

What’s wrong?

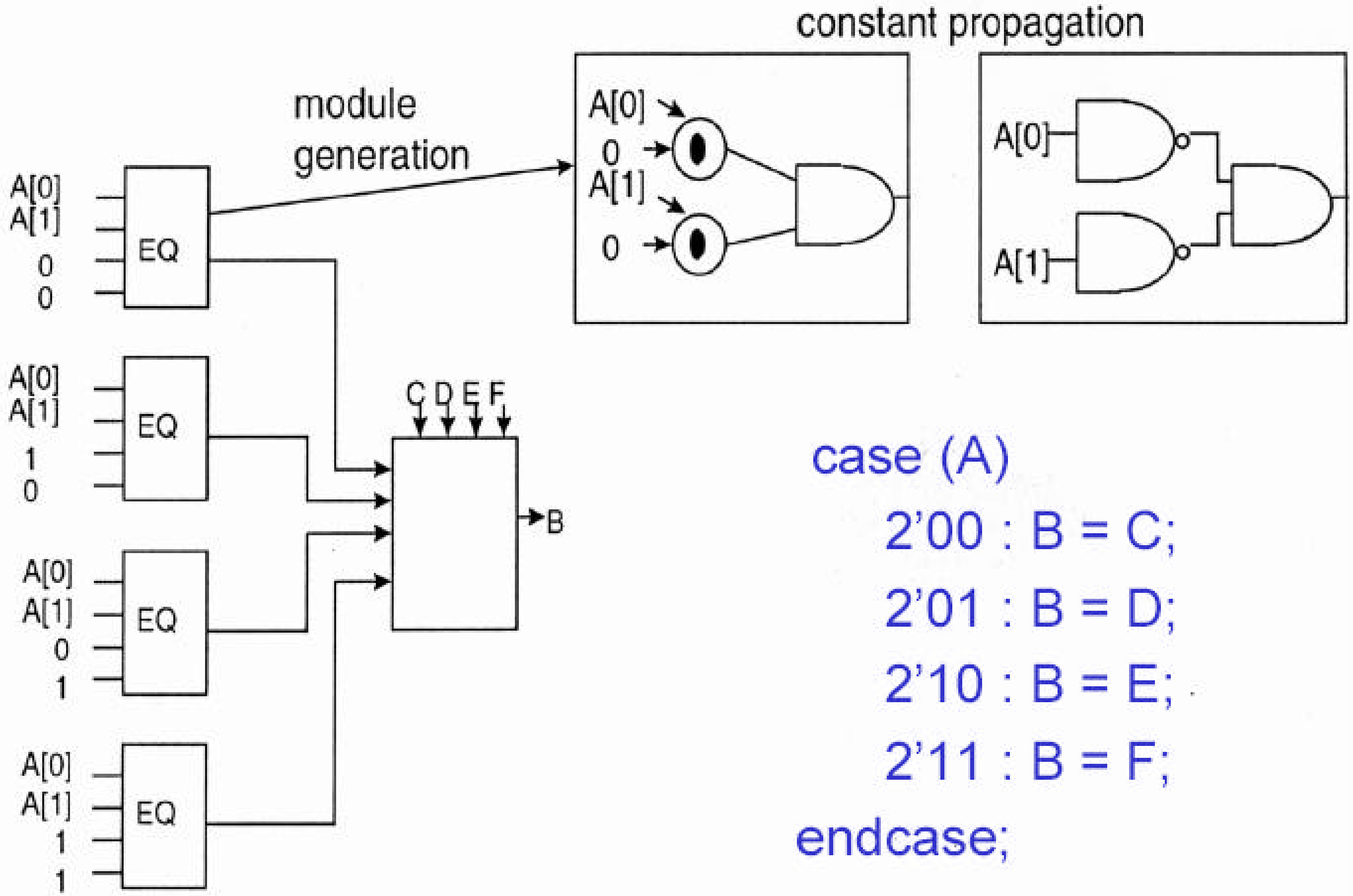
f doesn’t appear in *every* control path in the always block (neither does g).

IF-ELSE Statement

```
if (A(0) == 0 and A(1) == 0) then
    B = C;
else if (A(0) == 1 and A(1) == 0) then
    B = D;
else if (A(0) == 0 and A(1) == 1) then
    B = E;
else
    B = F;
end if;
```



Case Statement



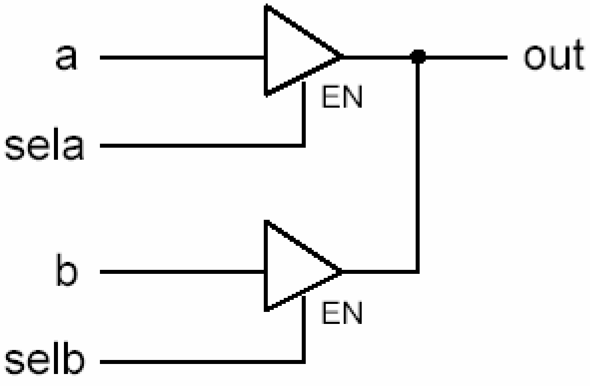
```
case (A)
    2'00 : B = C;
    2'01 : B = D;
    2'10 : B = E;
    2'11 : B = F;
endcase;
```

Tri-State Buffers

- Two popular ways to describe a tri-state buffer

```
// continuous assignment
assign out = (sela) ? a : 1'bz ;

// RTL modeling
always @(selb or b)
   if (selb)
      out = b ;
   else
      out = 1'bz;
```



Tasks and Functions

Tasks are like procedures in other programming languages, e. g
may have zero or more arguments and do not return a value. Fu
act like function subprograms in other languages. Except:

1. A Verilog function must execute during one simulation time ur
That is, no time controlling statements, i. e., no delay control (#),
event control (@) or **wait** statements, allowed. A task may conta
controlled statements.

2. A Verilog function can *not* invoke (call, enable) a task; wherea
task may call other tasks and functions.

Tasks and Functions

The definition of a task is the following:

```
task <task name>; // Notice: no parameter list or ()s
  <argument ports>
  <declarations>
  <statements>
endtask
```

An invocation of a task is of the following form:

`<name of task> (<port list>);`

Tasks and Functions

where **<port list>** is a list of expressions which correspond by p to the **<argument ports>** of the definition. Port arguments in th definition may be **input**, **inout** or **output**. Since the **<argument** in the task definition look like declarations, the programmer mu careful in adding declares at the beginning of a task.

Tasks and Functions

```
// Testing tasks and functions
module tasks;

task add;      // task definition
 input a, b;   // two input argument ports
 output c;     // one output argument port
 reg R;        // register declaration
 begin
   R = 1;
   if (a == b)
     c = 1 & R;
   else
     c = 0;
 end
endtask
```

Tasks and Functions

Task Continue…

```
initial begin: init1
   reg p;
   add(1, 0, p);  // invocation of task with 3 arguments
   $display("p= %b", p);
end

endmodule
```

Tasks and Functions

input and **inout** parameters are passed by value to the task and

and **inout** parameters are passed back to invocation by value o

Call by reference is not available.

- Allocation of all variables is static
- A task may call itself but each invocation of the task uses the s storage
- Since concurrent threads may invoke the same task, the progr must be aware of the static nature of storage and avoid unwant overwriting of shared storage space.

Tasks and Functions

- The purpose of a function is to return a value that is to be used expression
- A function definition must contain at least one **input** argument
- The definition of the function is as below:

```
function <range or type> <function name>;// Notice: no paramet
 or ()s
<argument ports>
<declarations>
<statements>
endfunction
```

Tasks and Functions

```
// Testing functions
module functions;
function [1:1] add2; // function definition
input a, b; // two input argument ports
reg R; // register declaration
begin R = 1;
if (a == b) add2 = 1 & R;
else add2 = 0;
end
endfunction
```

```
initial
begin: init1 reg p; p = add2(1, 0); // invocation of function with 2 arguments
$display("p= %b", p);
end
endmodule
```

Register Inference

- Allow sequential logic design
- Keep technology independent
- Latch — a level-sensitive memory device
- Flip-Flop — an edge-triggered memory device

Flip-Flop Inference

- Wire (port) assigned in the synchronous section

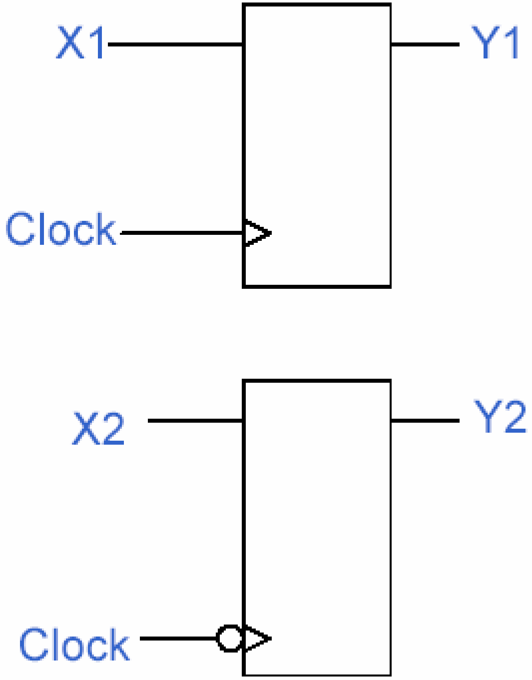
```
module FF_PN (Clock, X1, X2, Y1, Y2);
    input  Clock;
    input  X1, X2;
    output  Y1, Y2;

    reg  Y1, Y2;

   always @(posedge Clock)
    Y1 = X1;

   always @(negedge Clock)
    Y2 = X2;

 endmodule
```



Flip-Flop Inference

- Asynchronous / synchronous reset / enable

```
module FFS (Clock, SReset, ASReset, En, Data1, Data2, Y1, Y2
  Y3, Y4, Y5, Y6);

    input  Clock, SReset, ASReset, En, Data1, Data2;
    output  Y1, Y2, Y3, Y4, Y5, Y6;
    reg  Y1, Y2, Y3, Y4, Y5, Y6;

  always @(posedge Clock)
    begin
     // Synchronous reset
     if (! SReset)
       Y1 = 0;
     else
       Y1 = Data1 | Data2;
    end
```

Flip-Flop Inference

```
// Negative active asynchronous reset
always @(posedge Clock or negedge ASReset)
  if (! ASReset)
    Y2 = 0;
  else
    Y2 = Data1 & Data2;

// One synchronous & one asynchronous reset
always @(posedge Clock or negedge ASReset)
  if(! ASReset)
    Y3 = 0;
  else if (SReset)
    Y3 = 0;
  else
    Y3 = Data1 | Data2;
```

Flip-Flop Inference

```
// Single enable
 always @(posedge Clock)
   if (En)
     Y4 = Data1 & Data2;

 // Synchronous reset and enable
 always @(posedge Clock)
   if (SReset)
     Y5 = 0;
   else if (En)
     Y5 = Data1 | Data2;

endmodule
```

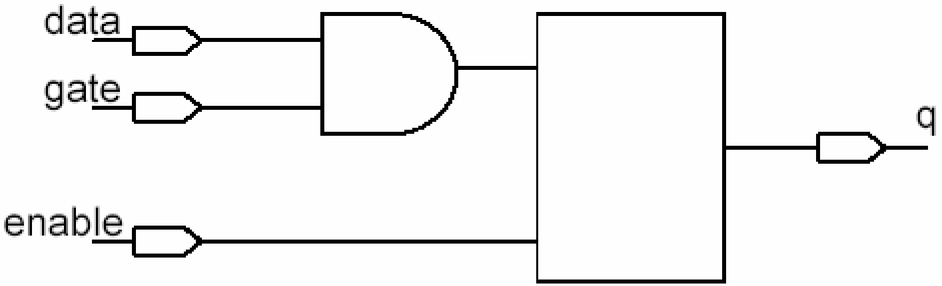
Latch Inference

- Incompletely specified wire (port) in the synchronous section
- D latch

```
always @(enable or data)
   if (enable)
        q = data;
```

- D latch with gated asynchronous data

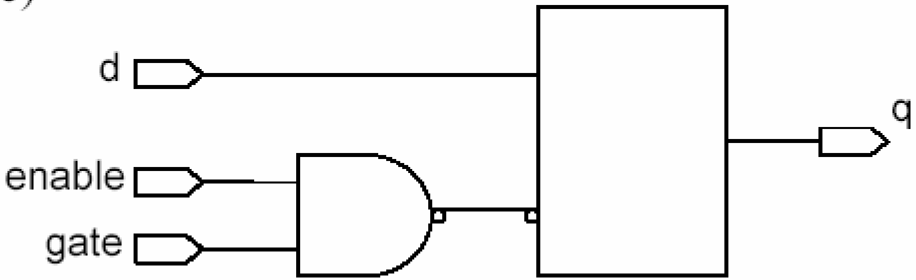
```
always @(enable or data or gate)
    if (enable)
        q = data & gate;
```



More Latches

- D latch with gated "enable"

```
always @(enable or d or gate)
  if (enable & gate)
        q = d;
```



- D latch with asynchronous reset

```
always @(reset or data or gate)
  if (reset)
     q = 1'b0;
  else if (enable)
     q = data;
```

Avoid Latch Inference

- Avoiding latch inference

```
always @(PHI_1 or A) begin
       Y = 0;
        if (PHI_1)
            Y = A;
end
```

- Another way

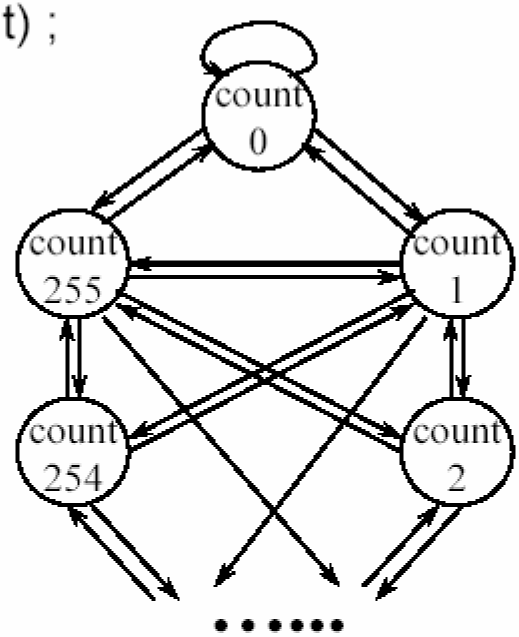
```
always @(PHI_1 or A) begin
       if (PHI_1)
            Y = A;
       else
             Y = 0;
end
```

1-Process FSM

- Lump all descriptions into a single process

```
module counter (clk, rst, load, in, count) ;
input         clk, rst, load ;
input   [7:0] in ;
output [7:0] count ;
reg      [7:0] count ;

always @(posedge clk) begin
    if (rst) count = 0 ;
    else if (load) count = in ;
    else if (count == 255) count = 0 ;
    else count = count + 1 ;
end
endmodule
```



256 states   66047 transitions

Verilog Template [1]

module *<module_name>* (*<ports>*)

input *<input_port_names>*;

output *<output_port_names>*;

reg *<outputs_and_values_to_be_used_in_always_blocks>*;

wire *<values_to_be_used_in_continuous_assignments>*;

//wire outputs do not need to be declared again

<called_module_name> **U1**(*<module_ports>*);

<called_module_name> **U2**(*<module_ports>*);

….

Verilog Template [2]

```
//continuous assignment

assign <wire_name> = <operation_of_wire_and_reg>;

//combinational always -> use blocking assignment “=“

always@ (<wire_or_reg1> or <wire_or_reg2> ..) begin

    <combinational_reg_name> = <operation_of_wire_and_reg>;

end
```

Verilog Template [3]

```
//sequentional always -> use non-blocking assignment “<=“

always@ (posedge clk or negedge reset) begin

if(!reset) begin

    <sequentional_reg_names> <= 0; //reset reg values

end

else begin

     <sequentional_reg_names> <= <operation_of_wire_and_reg>;

end

endmodule
```

Outline

- Introduction
- Language elements
- Gate-level modeling
- Data-flow modeling
- Behavioral modeling
- Modeling examples
- Simulation and test bench

Simulation

- Design, stimulus, control, saving responses, and verification can be completed in a single language
  - Stimulus and control
    - Use initial procedural block
  - Saving responses
    - Save on change
    - Display data
  - Verification
    - Automatic compares with expected responses
- The behavior of a design can be simulated by HDL simulators
  - Test benches are given by users as the inputs of the design
  - Some popular simulators
    - Verilog-XL (Cadence™, direct-translate simulator)
    - NC-Verilog (Cadence™, compiled-code simulator)
    - VCS (ViewLogic™, compiled-code simulator)

Supports for Verification

- Text output (show results at standard output)
  - **$display**: print out the current values of selected signals
    - Similar to the printf() function in C language
  - **$write**: similar to **$display** but it does not print a “\n”
  - **$monitor**: display the values of the signals in the argument list whenever any signal changes its value
  - Examples:

```
$display (“A=%d at time %t”, A, $time);
     A=5 at time 10

$monitor (“A=%d CLK=%b at time %t”, A, CLK, $time);
     A=2 CLK=0 at time 0
     A=3 CLK=1 at time 5
     ……
```

Test Bench

```
module test_bench;
    data type declaration
    module instantiation
    applying stimulus
    display results
endmodule
```

- A test bench is a top level module without inputs and outputs
- Data type declaration
  - Declare storage elements to store the test patterns
- Module instantiation
  - Instantiate pre-defined modules in current scope
  - Connect their I/O ports to other devices
- Applying stimulus
  - Describe stimulus by behavior modeling
- Display results
  - By text output, graphic output, or waveform display tools

Example Testfixture

```
`timescale 1ns / 1ps

reg [7:0] id1[0:63999];
reg [7:0] id2[0:63999];
reg [7:0] a[0:8];
integer c;

initial
begin


$readmemh ("e:\\matlabr12\\work\\clowngray.txt",id1);
$readmemh ("e:\\matlabr12\\work\\cartmangray.txt",id2);
a[0]=100;
a[1]=125;
a[2]=100;
a[3]=125;
a[4]=200;
a[5]=125;
```

```
a[6]=100;
a[7]=125;
a[8]=100;

#5 RST=1'b0;
#1 RST=1'b1;
#2 ModSel=1'b0;
#1431 StartFrame=1'b1;
InSelect=1'b0;
#1000 StartFrame=0;

$writememh("e:\\matlabr12\\work\\filtered1.txt",id3,0,63999);
$writememh("e:\\matlabr12\\work\\filtered2.txt",id4,0,63999);

#100 $finish;

end
```

```
always@ #50 CLK=~CLK;

always@(posedge RAMclk)
begin
        if(OutWrite==0)
        begin
                if (OutSelect==1'b0)
                        id3[AddOut16[15:0]]=DOut8[7:0];
                else
                        id4[AddOut16[15:0]]=DOut8[7:0];
        end

end
```

Some useful Verilog links,

http://www.cs.du.edu/~cag/courses/ENGR/ence3830/VHDL/
http://www.see.ed.ac.uk/~gerard/Teach/Verilog/manual/
http://oldeee.see.ed.ac.uk/~gerard/Teach/Verilog/manual/
http://www.ece.utexas.edu/~patt/02s.382N/tutorial/verilog_manu
http://www.eg.bucknell.edu/~cs320/1995-fall/verilog-manual.htm
http://mufasa.informatik.uni-mannheim.de/lsra/persons/lars/verilo
http://www.sutherland-hdl.com/on-line_ref_guide/vlog_ref_top.ht
http://www-cad.eecs.berkeley.edu/~chinnery/synthesizableVerilo
http://ee.ucd.ie/~finbarr/verilog/
http://athena.ee.nctu.edu.tw/courses/CAD/