Modeling Combinational Logic: 1

Structural Combinational Logic

- Low-level gate primitives are built in SV:
  - **and**, **or**, **nand**, **nor**, **xor**, **xnor**, **not**, **buf**.
  - These are keywords in bold font.
- SV is case sensitive;
  - keywords are always lower case.

Instantiation of Gate Primitives

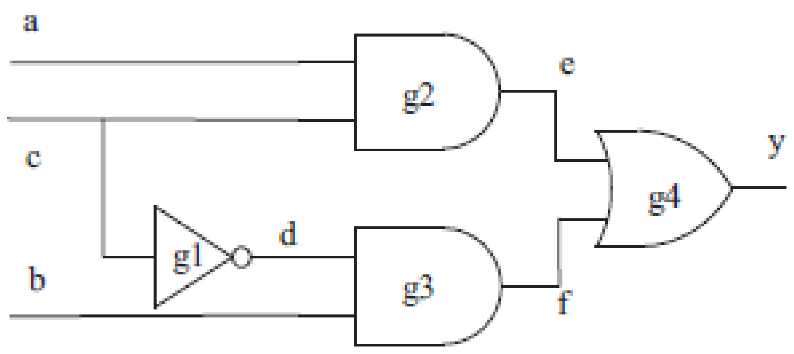
- A gate is connected to one or more **nets**.
  - These nets are listed in parentheses.
- A NAND gate with inputs a and b and output y:
  - **nand** g1 (y, a, b);
  - g1 is the label for the gate.
  - The semicolon (;) at the end of the instance.
- White space is not important, so this description could be split over two or more lines, or formatted to line up with other statements.

- Convention: The output comes first and is followed by the input(s).

Instantiation of Gate Primitives

- More than one gate instance declared at the same time:
  - nand g1 (y, a, b), g2 (w, c, d);
  - This describes two gates (g1 and g2).
- This can be split over two or more lines.

Netlist: Structural Modeling



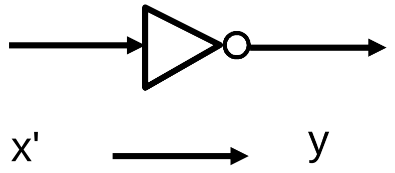
```
module ex1 (output wire y, input wire a, b, c);
    wire d, e, f;
    not g1 (d, c);
    and g2 (e, a, c);
    and g3 (f, d, b);
    or g4 (y, e, f);
endmodule
```

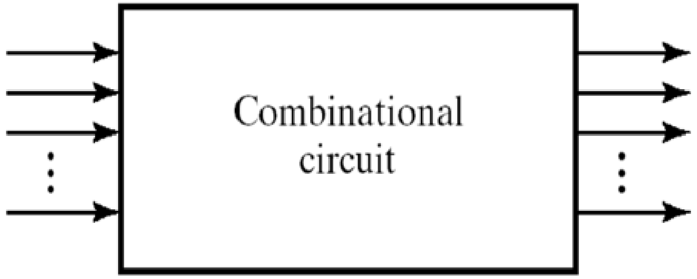
- The description begins with the keyword **module**, followed by a name.
- All the inputs and outputs are declared to be *nets* with the keyword **wire**.
- In fact, this keyword is not needed, but it is *strongly recommended*, however, that you declare all nets using the **wire** keyword (or the **logic** keyword).

Structural to Behavioral Modeling

- Although SV can describe a combinational function using interconnected primitive logic gates (e.g., AND, OR, NOR, ...),
  - this approach will be mostly ignored
  - since the design of large complex system requires synthesis tools that begin at a higher level of abstraction.

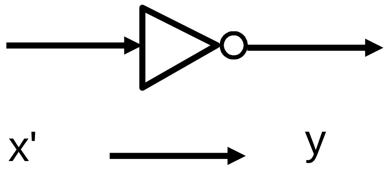
Behavioral Combinational Logic





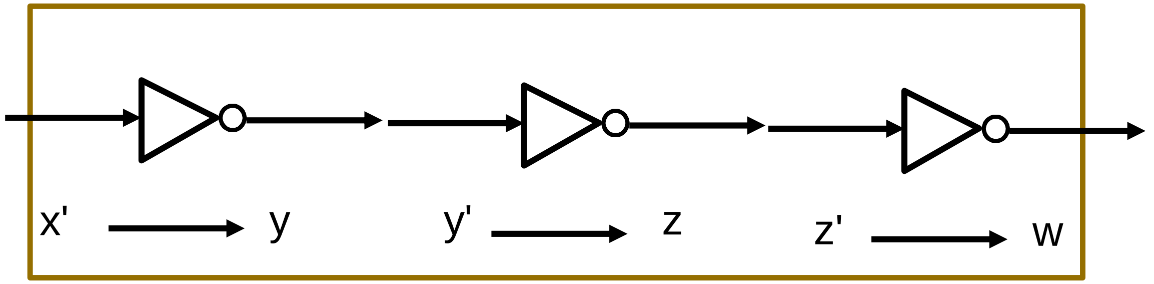
- Combinational logic is stateless:
  - changes in inputs are "immediately" reflected by changes (if any) in outputs.
- A direct flow from inputs to outputs without any memory elements.
- Behavioral Constructs:
  - assign statements
  - always_comb blocks
- The more abstract procedural constructs (always_comb) enable a synthesis tool to trade off implementation constraints such as area, propagation delay, and power.

Behavioral Combinational Logic: Continuum



```
assign y=~x;
```

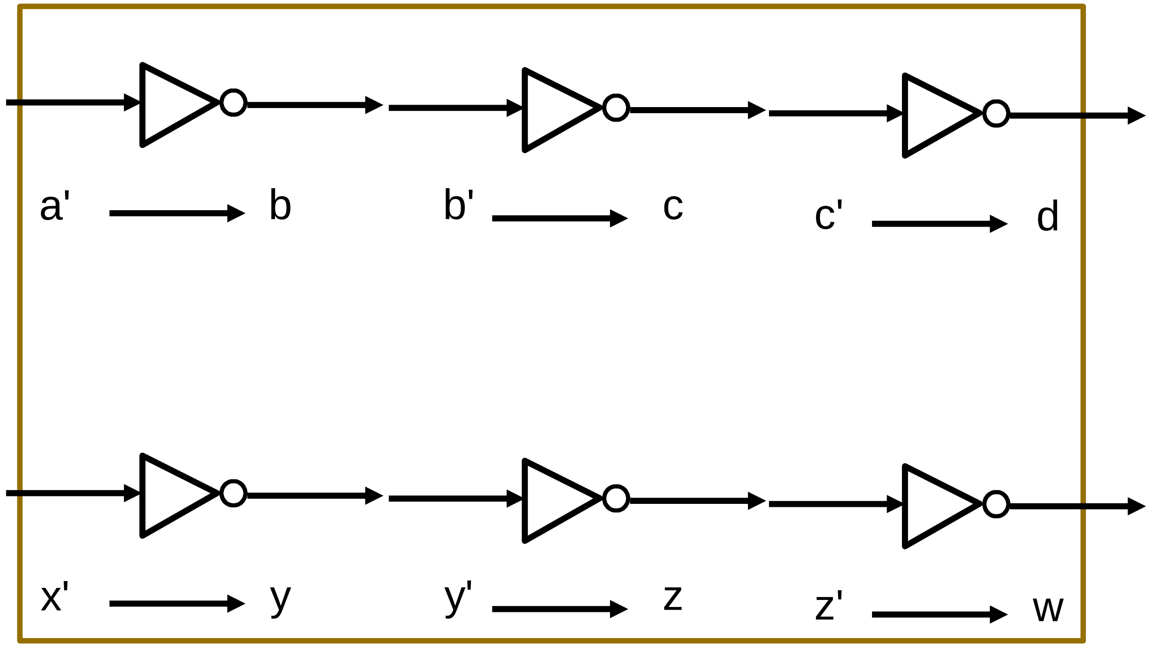
Behavioral Combinational Logic: Sequentiality



```
assign y=~x;

assign z=~y;

assign w=~z;
```

```
always_comb

Begin

      y=~x;

      z=~y;

      w=~z;

end
```

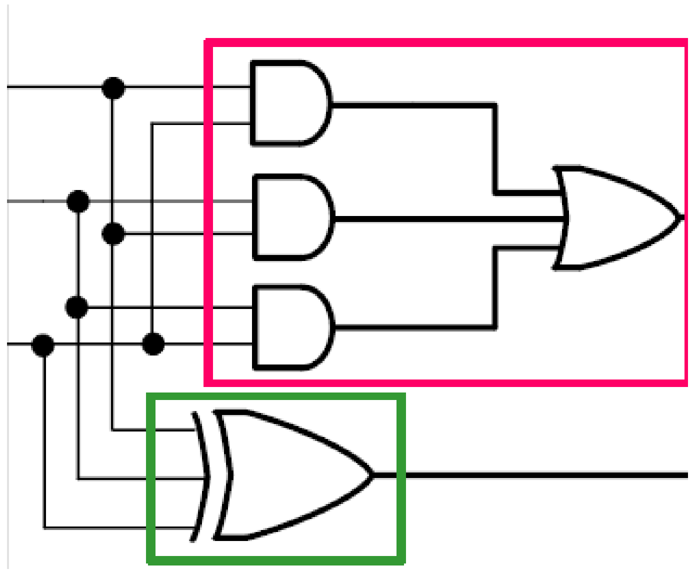
Behavioral Combinational Logic: Some Degree of Concurrency



```
assign y=~x;
assign z=~y;
assign w=~z;
assign b=~a;
assign c=~b;
assign d=~c;
```

```
always_comb
begin
    y=~x;
    z=~y;
    w=~z;
    b=~a;
    c=~b;
    d=~c;
end
```

Concurrency



```
module Adder (Cin, x, y, S, Cout)
   input  x, y, Cin;
   output  S, Cout;
   wire S, Cout;

   assign S = x ^ y ^ Cin;

   assign Cout = (x & y)|(x & Cin)|(y & Cin);

endmodule
```

Continuous Assignment in Module

```
module And2 (output wire z, input wire x, y);
assign z = x & y;
endmodule
```

- A two-input AND gate.
- The model has only one statement.
- The bitwise AND of x and y is assigned to z.
- The basic unit of a SystemVerilog design is the module.
- Note that a semicolon (";") follows the module header, but that there is no semicolon following the **endmodule** keyword.

Continuous Assignment in Modules

```
module And2 (output wire z, input wire x, y);
assign z = x & y;
endmodule
```

- The header contains the inputs and outputs of the module.
- One **output** of type **wire** is declared, z, followed by x and y, defined by the keywords **input** and **wire**.
- Inputs and outputs can appear in any order
  - Convention with SV: outputs are declared before inputs.
- Because x and y have the same direction and the same type, they can be listed together after the keywords.

Data Object: wire

```
module And2 (output wire z, input wire x, y);
assign z = x & y;
endmodule
```

- Characteristics of wire:
  - Wires are used for connecting different modules
  - They can be treated as a physical wire
  - They can be read or assigned
  - No values get stored in them
  - They need to be driven by either continuous assign statement or from a port of a module

Continuous Statement: assign

- The left-hand side can be a variable or a net.

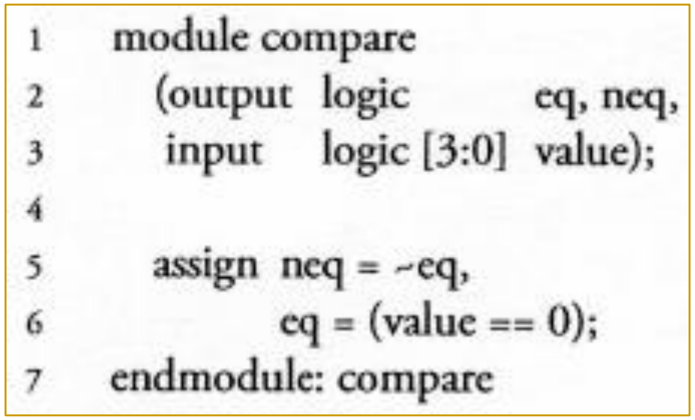
Multiple Continuous Assignment

```
module gates(input logic [3:0] a, b,
             output logic [3:0] y1, y2, y3, y4, y5);
/* five different two-input logic
gates acting on 4-bit busses */
   assign y1 = a & b;       // AND
   assign y2 = a | b;       // OR
   assign y3 = a ^ b;       // XOR
   assign y4 = ~(a & b);    // NAND
   assign y5 = ~(a | b);    // NOR
endmodule
```

```
module fulladder(input logic a, b, cin,
                 output logic s, cout);
logic p, g;
assign p = a ^ b;
assign g = a & b;
assign s = p ^ cin;
assign cout = g | (p & cin);
endmodule
```

- Anytime the inputs on the right side of the = in a continuous assignment statement change, the output on the left side is recomputed.
- Continuous assignment statements describe combinational logic.

Multiple Continuous Assignment: More Examples



```
module compare
   (output  logic       eq, neq,
    input   logic [3:0] value);

   assign  neq = ~eq,
                eq = (value == 0);
endmodule: compare
```

- assign S1, S2, …, Sm;
- =
  - assign S1;
  - assign S2;
  - …
  - assign Sm;

```
module and8 (input logic [7:0] a,
                   output logic y);
assign y = a[7] & a[6] & a[5] & a[4] &a[3] & a[2] & a[1] & a[0];
endmodule
```

```
module and8 (input logic [7:0] a,
                   output logic y);
assign y = &a;
endmodule
```

A Ternary Operator => 4:1 Multiplexer

```
module mux2(input logic [3:0] d0, d1,
            input logic s,
            output logic [3:0] y);
assign y = s ? d1 : d0;
endmodule
```

```
module mux4(input logic [3:0] d0, d1, d2, d3,
            input logic [1:0] s,
            output logic [3:0] y);
assign y = s[1] ? (s[0] ? d3 : d2) : (s[0] ? d1 : d0);
endmodule
```

Internal Signals: logic type

```
module fulladder(input logic a, b, cin,
                 output logic s, cout);
logic p, g;
assign p = a ^ b;
assign g = a & b;
assign s = p ^ cin;
assign cout = g | (p & cin);
endmodule
```

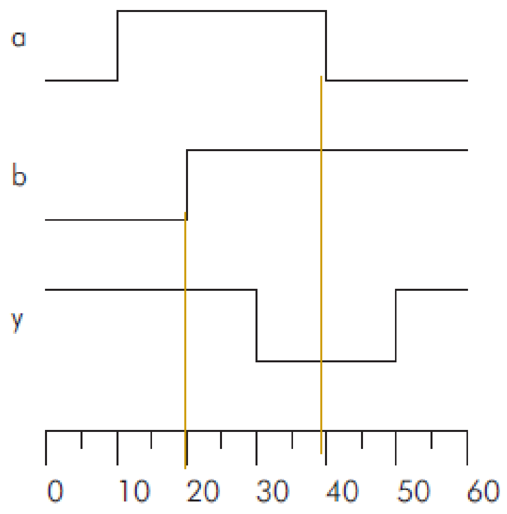
- In SystemVerilog, internal signals are usually declared as logic.

Delays

- Propagation delays can be specified for combinational circuits, but in modern design flows where logic synthesis tools are used, they are ignored by the synthesis tool.
- When generating a netlist is from a synthesis tool or by extracting the circuit *after* physical layout of the circuit, you verify your design by simulation.
- To verify the timing, the simulation model must include timing information.

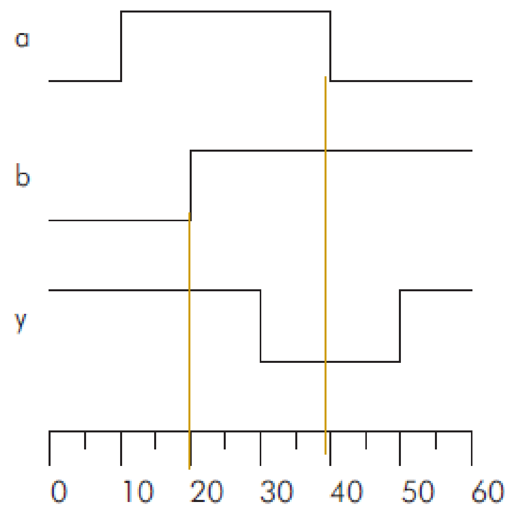
Delay Specification

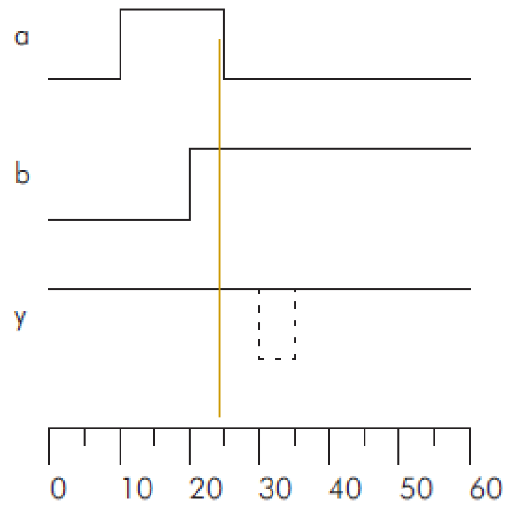
- A delay of 10 ps through a NAND gate:
- nand #10ps g1 (y, a, b);
- The hash symbol (#) is used to denote a parameter.
- The output is at logic 1 if either or both inputs is at logic 0.
    - Therefore, the output will only go to logic 0 after the second of the two inputs has gone to 1.
    - This change will be delayed by 10 ps.



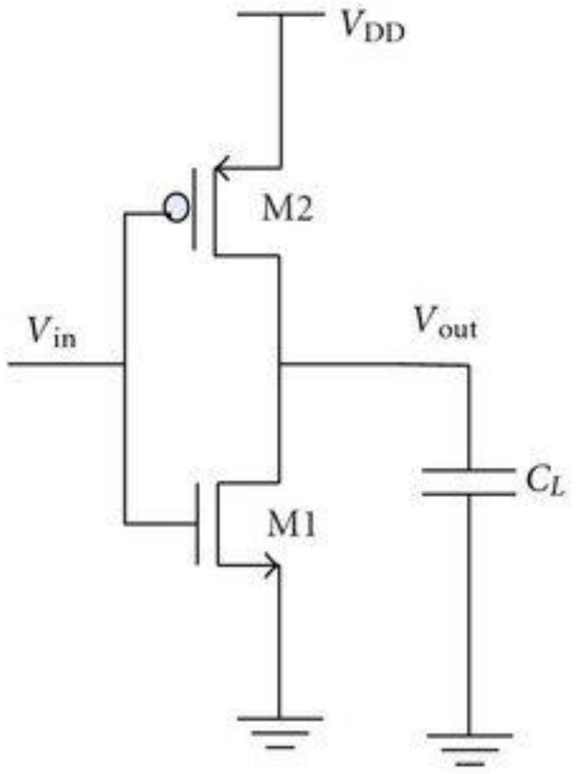
Delays

- **nand** #10ps g1 (y, a, b);
- If signal b goes to 1 at time 20 ps; signal a goes back to 0 at time 40 ps.
  - The pulse on y is 20 ps wide, delayed by 10 ps.
- Suppose that a changes back to 0 at time 25 ps.
  - This would suggest that a pulse 5 ps wide would appear at y, again delayed by 10 ps.
  - In fact, the delay has a second meaning:
    - any pulse less than 10 ps wide is suppressed.





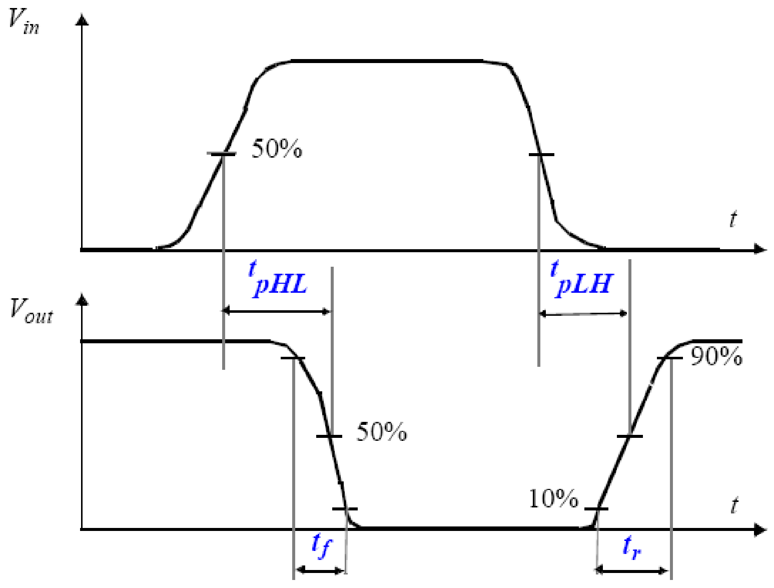
Inertial Delays



- This is known as an *inertial delay.*
- Hence, a pulse is suppressed by *inertial cancellation.*
- This delay model assumes that the delay through a gate is the same for a 0 to 1 transition on the output as for a 1 to 0 transition.
- This assumption is probably valid for CMOS technologies, but may not be true in other cases.

Rise/Fall Delays

- If the 0 to 1 and 1 to 0 delays differ, the two values may be specified.
- For example,
  - **nand** #(10ps, 12ps) g1 (y, a, b);
  - describes a NAND gate that has
    - a 10 ps delay when the output changes to 1 (rise delay) and
    - a 12 ps delay when the output changes to 0 (fall delay).



Model Uncertainty

- **nand** #(8ps:10ps:12ps, 10ps:12ps:14ps) *g1* (y, a, b);
  - a minimum rise delay of 8 ps,
  - a typical rise delay of 10 ps and
  - a maximum rise delay of 12 ps.
- Similarly, the fall delay has three values.

- When a circuit is extracted from a silicon layout,
  - it is not possible to exactly predict the delay through each gate or between each gate, because of process variations.
- It is reasonable, however, to expect that the minimum, typical and maximum delays through a gate can the minimum, typical and maximum delays respectively.

Minimum, Typical & Maximum Delays

- **nand** #(8ps:10ps:12ps, 10ps:12ps:14ps) g1 (y, a, b);
  - describes a NAND gate that has
    - a minimum rise delay of 8 ps,
    - a typical rise delay of 10 ps and
    - a maximum rise delay of 12 ps.
  - Similarly, the fall delay has three values.
- A simulation can be performed using the minimum, typical or maximum delays for all gates.
- In principle, the functionality of the circuit can be verified under extremes of fabrication.

Delays in Continuous Assignment

- Similarly, a delay can be associated with a continuous assignment:
  - **assign** #10ps z = x & y;
  - This is an inertial delay that behaves in exactly the same way as for a gate primitive.

- By the same analogy, rising and falling delays and minimum, typical and maximum delays can be included, as for a gate:
  - **assign** #(8ps:10ps:12ps, 10ps:12ps:14ps) z = x & y;

Delays in Continuous Assignment

```
' timescale 1ns/1ps
module example(input logic a, b, c,
                        output logic y);
logic ab, bb, cb, n1, n2, n3;
assign #1 {ab, bb, cb} = ~{a, b, c};
assign #2 n1 = ab & bb & cb;
assign #2 n2 = a & bb & cb;
assign #2 n3 = a & bb & c;
assign #4 y = n1 | n2 | n3;
endmodule
```

- In SV, a # symbol is used to indicate the number of units of delay.
- It can be placed in assign statements,
  - as well as non-blocking (<=) and blocking (=) assignments.

Parameters in Continuous Assignment

```
module And2 #(parameter delay) (output wire z, input wire x, y);
assign #delay z = x & y;
endmodule
```

- **assign** #5ps z = x & y; defines the exact delay for an AND gate.
- Different technologies, and indeed different instances, will have different delays.
- We could declare a number of alternative modules for an AND gate, each with a different delay.
- It would be better to write the statement as:
  - **assign** #delay z = x & y;
  - and to define delay as a **parameter** to the SystemVerilog model.
- When the gate is used in a netlist, a value is passed to the model as a parameter:
- And2 #(5ps) g2 (p, b, q);

Concurrent Statements

- Concurrent statements mean that the operations described in each line take place in parallel.
- The commonly used concurrent constructs are
  - gate instantiation and
  - the continuous assignment statement.

```
module ex1 (output wire y, input wire a, b, c);
    wire d, e, f;
        not g1 (d, c);
        and g2 (e, a, c);
        and g3 (f, d, b);
        or g4 (y, e, f);
endmodule
```

```
module mux2(a, b, sel, f);
    output f;
    input a, b, sel;
    logic c, d;
        assign c = a & (~sel);
        assign d = b & sel;
        assign f = c | d;
endmodule
```

Continuous Assignment: Limited Concurrent Statements

- Continuous assignment statement have a limited Concurrency
- Example:
  - assign sum = a + b + c;
- can be re-written as
  - assign sum = a + b;
  - assign sum = sum + c;
- but, although equivalent sequentially, it's incorrect as both statements are concurrent, so the value of sum is indeterminate.
- Note that no net (wire) should be assigned a value more than once with concurrent statements.

Sequential Statements

- Sequential statements are similar to statements in a normal programming language.
- A sequence of statements is regarded as describing operations that take place one after the other instead of at the same time.
  - All sequential SV statements must be inside a procedural block.
  - Sequential statements are placed inside a begin/end block and executed in sequential order within the block.
  - However, the blocks themselves are executed concurrently.
  - SV statements outside any process block are interpreted as concurrent statements and different procedural blocks execute concurrently.

Combinational Procedural Block

- always_comb
  - procedural block to describe combinational logic using a series of sequential statements
- All always_comb blocks are independent and parallel to each other

```
module mymodule (……………….);
    output …………………;
    input ………………………;
    always_comb begin
    // Combinational logic described
    // in C-like syntax
    // Assignments: blocking statements
    // Decision statements: if, case, ….
    //  Loop, generate, …
    end
endmodule
```

always_comb a = b + c;

- **Simulator:**
- When either b, c, or both change,
  - then the simulator will execute the statement calculating a new value for a;
  - then wait for the next change on b, c, or both.
- If the new value of a is different from the old, then the simulator also propagates the value of a to other models on its fanout.
- It is **always** **actively waiting** for a change on its inputs, at which point it will execute its statement.
- **Synthesizer:**
- If you synthesized a module with just this statement in it, and assuming that a, b, and c are scalars, the result would be a half adder.

always_comb

```
module inv (input logic [3:0] a,
            output logic [3:0] y);
always_comb
   y = ~a;
endmodule
```

- always_comb reevaluates the statements inside the always statement any time any of the signals on the right hand side of = in the always statement change.
- From sensitivity list point of view, always_comb is equivalent to always @(*) , but is preferred in SV.
- always_comb indicates that the block models purely combinational logic at the register transfer level.
- If the code inside the always block is not combinational logic, SV will report a warning.

Blocking Assignments in always_comb

```
module inv (input logic [3:0] a,
                  output logic [3:0] y);
always_comb
    y = ~a;
endmodule
```

- The = in the always statement is called a blocking assignment.
  - A sequential assignment, implying an order.
- In SV, it is good practice to use blocking assignments for combinational logic.

Blocking Assignments

- The left-hand side must be a variable.
- The right-hand side can be a variable or a net.

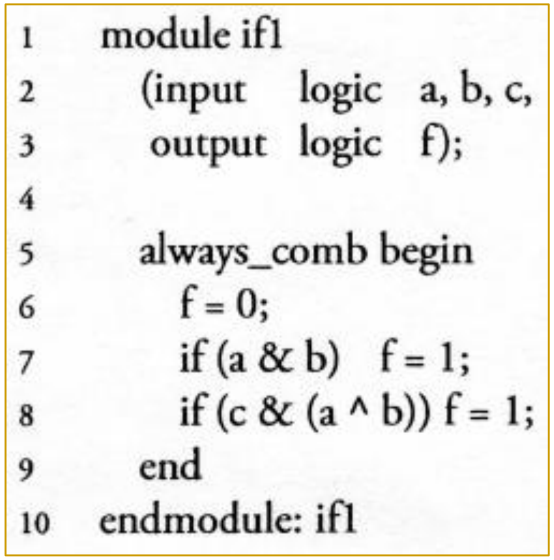
Full Adder with always_comb

```
module fulladder(input logic a, b, cin,
                 output logic s, cout);
logic p, g;
always_comb
begin
  p = a ^ b;              // blocking
  g = a & b;              // blocking
  s = p ^ cin;            // blocking
  cout = g | (p & cin);   // blocking
end
endmodule
```

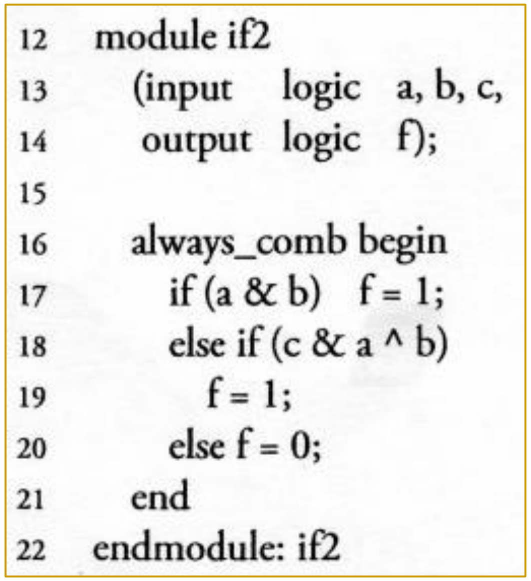
- always @(a, b, cin) would have been equivalent to always_comb.
- However, always_comb is better because it avoids common mistakes of missing signals in the sensitivity list.
- This example uses blocking assignments, first computing p, then g, then s, and finally cout.

Procedural Modeling: if, if-else

- f, a, b, and c are single-bit logic variables.
- $f = (a \cdot b) + (b \cdot c) + (a \cdot c)$
- `assign f = (a & b) | (b & c) | (a & c);`
- "^" is the logical XOR operator



```
module if1
   (input    logic    a, b, c,
    output   logic    f);

   always_comb begin
      f = 0;
      if (a & b)    f = 1;
      if (c & (a ^ b)) f = 1;
   end
endmodule: if1
```



```
module if2
   (input    logic    a, b, c,
    output   logic    f);

   always_comb begin
      if (a & b)   f = 1;
      else if (c & a ^ b)
         f = 1;
      else f = 0;
   end
endmodule: if2
```

Mux with always_comb

```
module mux2 (output logic y,
             input logic a, b, s);
always_comb
    if (s)
        y = b;
    else
        y = a;
endmodule
```

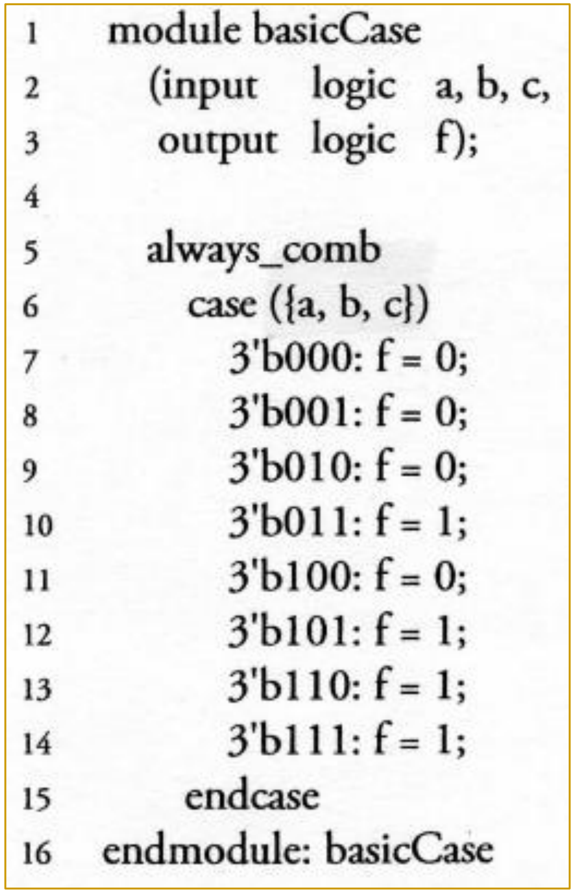
```
module mux4 (output logic y,
             input logic a, b, c, d, s0, s1);
always_comb
    if (s0)
        if (s1)
            y = d;
        else
            y = c;
    else
        if (s1)
            y = b;
        else
            y = a;
endmodule
```

Case Statement with Full Cases

```
module basicCase
   (input   logic   a, b, c,
    output  logic   f);

   always_comb
      case ({a, b, c})
         3'b000: f = 0;
         3'b001: f = 0;
         3'b010: f = 0;
         3'b011: f = 1;
         3'b100: f = 0;
         3'b101: f = 1;
         3'b110: f = 1;
         3'b111: f = 1;
      endcase
endmodule: basicCase
```

- The case statement in SystemVerilog is similar to case and switch statements found in programming languages;
  - a value is used to specify one of several statements to execute.
- In SV, sized variables (i.e., variables with known bit widths) can be concatenated with the operator { } to form new variables.
- A comma-separated list of variable names is given in the concatenation operator.

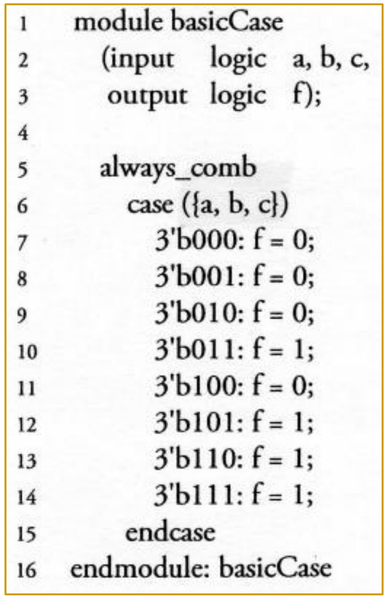
Case Statement with Full Cases



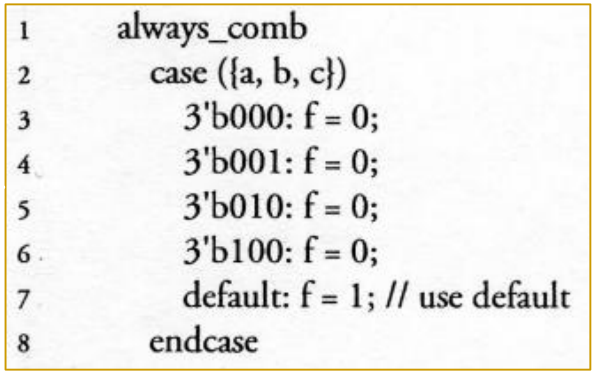
```
module basicCase
   (input    logic    a, b, c,
    output   logic    f);

   always_comb
      case ({a, b, c})
         3'b000: f = 0;
         3'b001: f = 0;
         3'b010: f = 0;
         3'b011: f = 1;
         3'b100: f = 0;
         3'b101: f = 1;
         3'b110: f = 1;
         3'b111: f = 1;
      endcase
endmodule: basicCase
```

- The case statement matches the case expression with the first case selection item that is equal to it.
- Then the statement to the right is executed and then execution continues with the statement after **endcase**.
- For always_comb, anytime a, b, or c changes, the case statement is executed, setting f to either 1 or 0, and then always_comb waits for the next change on its inputs.
- Since we have enumerated all possible values of the 3-bit variable {a, b, c} in the case statement, combinational output f will be updated anytime the loop is executed and thus a synthesis tool will treat this description as a combinational circuit.

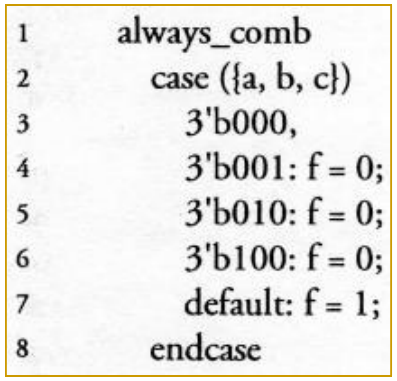
Equivalent Case Statements with Default



```
module basicCase
   (input   logic   a, b, c,
    output  logic   f);

   always_comb
      case ({a, b, c})
         3'b000: f = 0;
         3'b001: f = 0;
         3'b010: f = 0;
         3'b011: f = 1;
         3'b100: f = 0;
         3'b101: f = 1;
         3'b110: f = 1;
         3'b111: f = 1;
      endcase
endmodule: basicCase
```



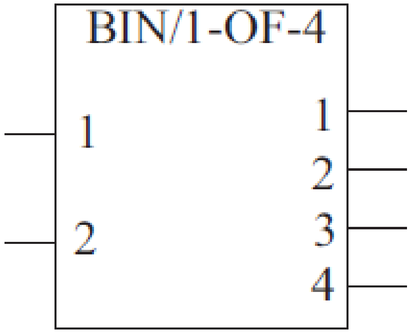
```
always_comb
   case ({a, b, c})
      3'b000: f = 0;
      3'b001: f = 0;
      3'b010: f = 0;
      3'b100: f = 0;
      default: f = 1; // use default
   endcase
```



```
always_comb
   case ({a, b, c})
      3'b000,
      3'b001: f = 0;
      3'b010: f = 0;
      3'b100: f = 0;
      default: f = 1;
   endcase
```

Decoders: Case with Full Cases

```
module decoder (output logic [3:0] y,
                        input logic [1:0] a);
always_comb
    case (a)
        0 : y = 1;
        1 : y = 2;
        2 : y = 4;
        3 : y = 8;
        default : y = 'x; // necessary only for simulation
    endcase
endmodule
```



Inputs		Outputs			
A1	A0	Z3	Z2	Z1	Z0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

- 'x fills all bits on the left-hand side with x.

Polymorphism

```
module decoder (output logic [3:0] y,
                input logic [1:0] a);
always_comb
    case (a)
        0 : y = 1;
        1 : y = 2;
        2 : y = 4;
        3 : y = 8;
        default : y = 'x;
    endcase
endmodule
```

- SV is not a strongly typed language.
- The input, a, is declared to be a two-bit variable of type logic, but it is interpreted as an integer.
- This is acceptable in SV, but would be completely illegal in many other HDLs and programming languages.
- This is ability to interpret bit patterns automatically is very powerful, but can be dangerous.
- Similarly, the output is assigned an integer value that is automatically reinterpreted as four bits.

Case Statements with Default

```
module decoder (output logic [3:0] y,
                      input logic [1:0] a);
always_comb
    case (a)
        0 : y = 1;
        1 : y = 2;
        2 : y = 4;
        3 : y = 8;
        default : y = 'x; // necessary only
                          // for simulation
    endcase
endmodule
```

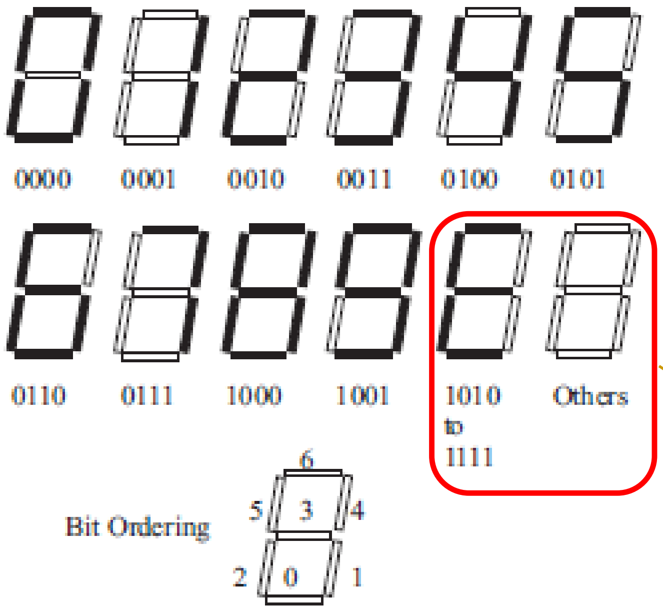
- The fifth alternative is a default.
  - This seems redundant as two bits give four values, as specified.
- The control input, a, is of type logic, however.
  - Its bits can take x or z values.
  - 16 possible values for a.
- The default line assigns an x to the output if any of the input bits is not a true binary value.
- This line will not be synthesized, but it is good practice to include it if you want to check for unusual behavior in simulation.

Decoder of 3->8
Case Statements with Default

```
module decoder3_8(input logic [2:0] a,
                  output logic [7:0] y);
always_comb
case(a)
   3'b000: y = 8'b00000001;
   3'b001: y = 8'b00000010;
   3'b010: y = 8'b00000100;
   3'b011: y = 8'b00001000;
   3'b100: y = 8'b00010000;
   3'b101: y = 8'b00100000;
   3'b110: y = 8'b01000000;
   3'b111: y = 8'b10000000;
   default: y = 8'bxxxxxxxx; // necessary only
                             // for simulation
endcase
endmodule
```

- The default statement is not strictly necessary for logic synthesis in this case because all possible input combinations are defined,
- but it is prudent for simulation in case one of the inputs is an x or z.

Seven-Segment Decoder



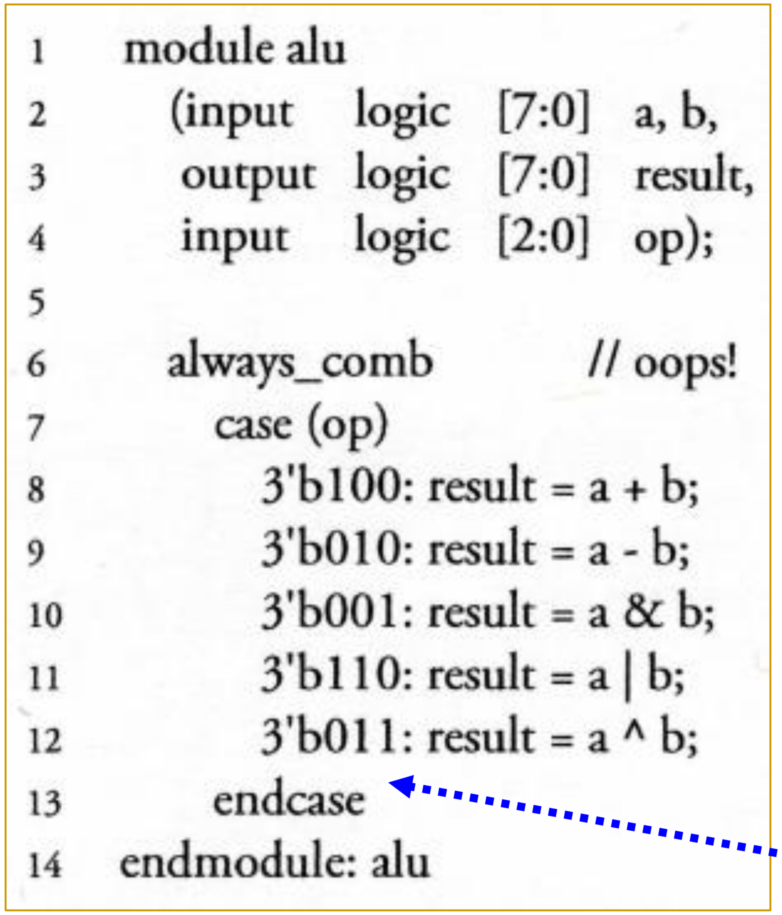
- If more than one pattern should give the same output, the patterns can be listed.
- A seven-segment decoder to display the digits '0' to '9'.
- If the bit patterns corresponding to decimal values '10' to '15' are fed into the decoder, an 'E' (for "Error") is displayed.
- If the inputs contain 'X's or other invalid values, the display is blanked.

Case Statements with Default

```
module sevenseg(input logic [3:0] data,
                output logic [6:0] segments);
always_comb
 case(data)
  // abc_defg
    0: segments = 7'b111_1110;
    1: segments = 7'b011_0000;
    2: segments = 7'b110_1101;
    3: segments = 7'b111_1001;
    4: segments = 7'b011_0011;
    5: segments = 7'b101_1011;
    6: segments = 7'b101_1111;
    7: segments = 7'b111_0000;
    8: segments = 7'b111_1111;
    9: segments = 7'b111_0011;
   default: segments = 7'b000_0000; // necessary!!
endcase
endmodule
```

- The case statement checks the value of data.
- When data is 0, the statement performs the action after the colon, setting segments to 1111110.
- The case statement similarly checks other data values up to 9 (note the use of the default base, base 10).
- The default clause is a convenient way to define the output for all cases not explicitly listed, guaranteeing combinational logic.
- In SV, case statements must appear inside always statements.

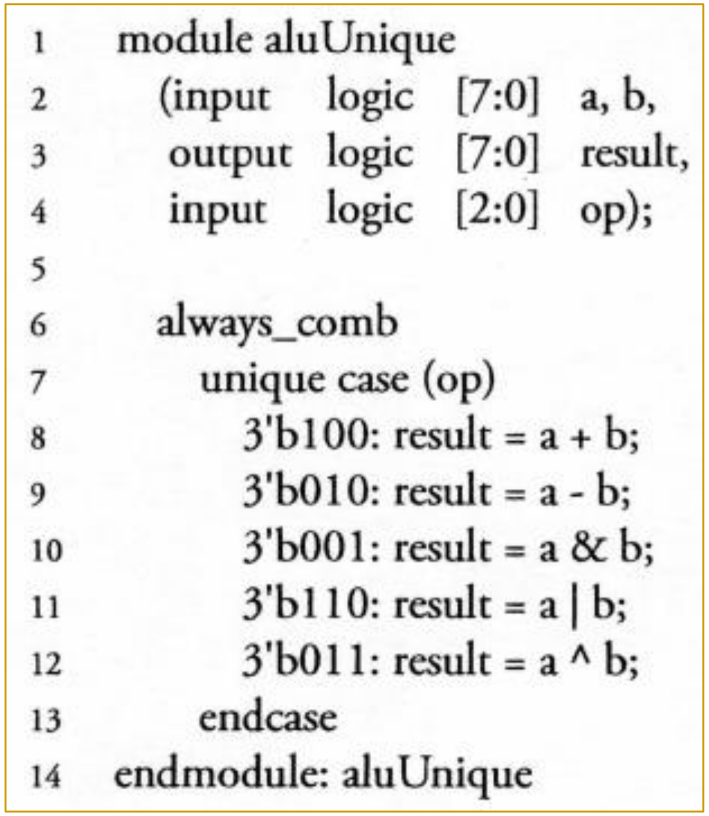
Case with Incomplete Cases



```
module alu
   (input   logic  [7:0]  a, b,
    output  logic  [7:0]  result,
    input   logic  [2:0]  op);

   always_comb          // oops!
      case (op)
         3'b100: result = a + b;
         3'b010: result = a - b;
         3'b001: result = a & b;
         3'b110: result = a | b;
         3'b011: result = a ^ b;
      endcase
endmodule: alu
```

- This is not a combinational circuit!
- A synthesis tool would analyze this description and note that if op was 3'b111, 3'b000, or 3'b101,
  - then result would have to be remembered from the previous execution of the loop.
- To fix this, we could add a **default** statement.

SV Solution: Unique Case



```
module aluUnique
   (input   logic  [7:0]  a, b,
    output  logic  [7:0]  result,
    input   logic  [2:0]  op);

   always_comb
      unique case (op)
         3'b100: result = a + b;
         3'b010: result = a - b;
         3'b001: result = a & b;
         3'b110: result = a | b;
         3'b011: result = a ^ b;
      endcase
endmodule: aluUnique
```

- When the keyword **unique** is used with case
  - it asserts that one-and-only-one of the case selection items will match.
- It provides an automatic check in simulation that only correct values are on the op input, and it also specifies to the synthesizer that all other input combinations are don't-care.
- It gives the simulator and synthesis tool the same semantic understanding of the intended logic circuit.
- The synthesis tool will use any unlisted combinations as don't-cares in its optimization.

Adders

```
module adder #(parameter N = 4)
        (output logic [N-1:0] Sum, output logic Cout,
        input logic [N-1:0] A, B, input logic Cin);
        always_comb
        {Cout, Sum} = A + B + Cin;
endmodule
```

```
module fulladder (output logic sum, cout,
                            input logic a, b, cin);
always_comb
begin
sum = a ^ b ^ cin;
cout = a & b | a & cin | b & cin;
end
endmodule
```

Ripple Adder: **generate for**

- If we know how many bits will be in our adder we simply instantiate the model several times.
- If, however, we want to create a general N-bit adder, we need some type of iterative construct.
- The **generate** construct with a **for** loop allows repetition in a dataflow description.
  - It is possible to use it for loops to mimic multiple instants.
- For loop inside always:
  - it follows the activation of always
  - only can use blocking and non-blocking assignment.
- In a generate block, you can use continuous assign, always and module instantiation.

Ripple Adder: generate for

```
module ripple #(parameter N = 4)
    (output logic [N-1:0] Sum, output logic Cout,
    input logic [N-1:0] A, B, input logic Cin);
    logic [N-1:1] Ca;
    genvar i;
    fulladder f0 (Sum[0], Ca[1], A[0], B[0], Cin);
    generate for (i = 1; i < N-1; i++)
        begin : f_loop
        fulladder fi (Sum[i], Ca[i+1], A[i], B[i], Ca[i]);
        end
    endgenerate
    fulladder fN (Sum[N-1], Cout, A[N-1], B[N-1], Ca[N-1]);
endmodule
```

- This feature (Verilog 2001) was taken from VHDL with some modification.
- This example creates N-2 instances and, through the Ca vector, wires them up.
- Notice that the loop variable, i, is declared as a **genvar**.
- We make special cases of the first and last elements, by instantiating them outside the generate block.

Tasks

```
module ripple_task #(parameter N = 4)
(output logic [N-1:0] Sum, output logic Cout,
input logic [N-1:0] A, B, input logic Cin);
logic [N-1:1] Ca;
genvar i;
task automatic fulladder (output logic sum, cout,
                                        input logic a, b, cin);
        begin
        sum = a ^ b ^ cin;
        cout = a & b | a & cin | b & cin;
        end
endtask
always_comb
        fulladder (Sum[0], Ca[1], A[0], B[0], Cin);
generate for (i = 1; i < N-1; i++)
begin : f_loop
always_comb
fulladder (Sum[i], Ca[i+1], A[i], B[i], Ca[i]);
end
endgenerate
always_comb
        fulladder fN (Sum[N-1], Cout, A[N-1], B[N-1], Ca[N-1]);
endmodule
```

- The full adder could also be implemented as a **task**.
- A SystemVerilog **task** is a **sub-routine**, like a **function**, but without a return value.
- Tasks can include more diverse constructs than functions.
- The task is declared as **automatic** to ensure that each call has its own copy of variables.
- Otherwise, variables are shared between each call, which would lead to conflicts between assignments.

Parity Checker

```
module parity_loop #(parameter N = 4)
(output logic even, input logic [N-1:0] a);
always_comb
      begin
      even = '1;
      for (int i = 0; i < N; i++)
      even = even ^ a[i];
      end
endmodule
```

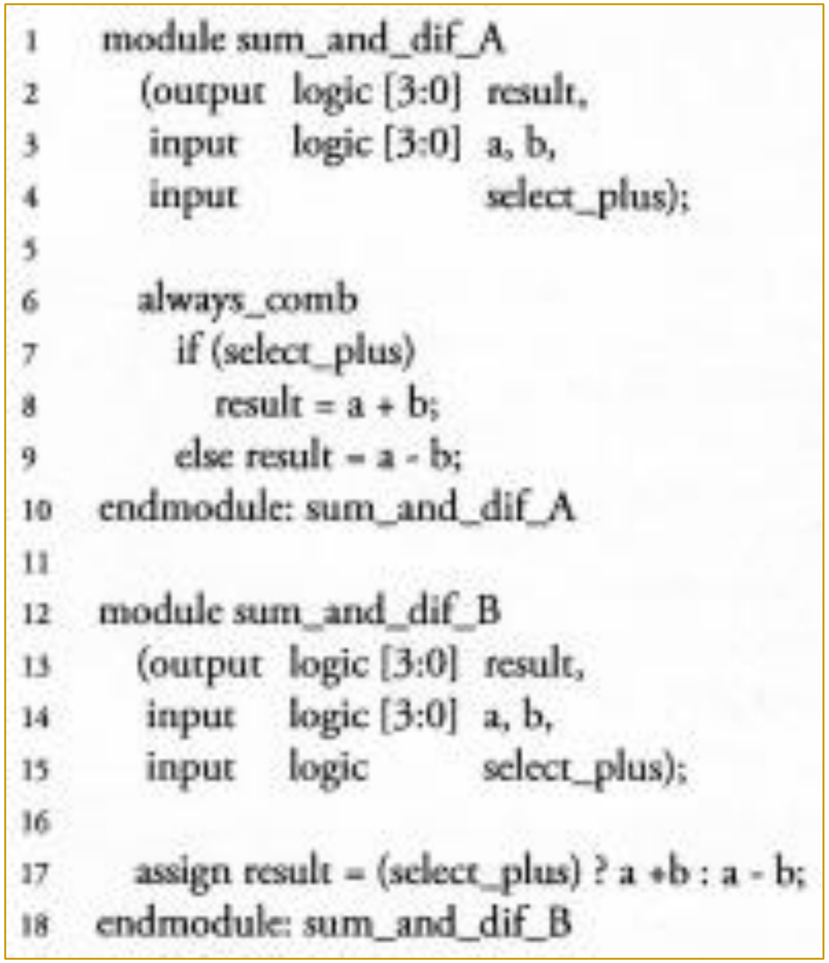
```
module parity #(parameter N = 4)
(output logic even, input logic [N-1:0] a);
always_comb
      even = ~^a;
endmodule
```

Parity Checker

```
module parity #(parameter N = 4)
(output logic even, input logic [N-1:0] a);
always_comb
    even = ~^a;
endmodule
```

- It is possible to do this in a much more concise way.
- In addition to the usual programming operators, SystemVerilog has reduction operators that can be applied to all the bits of a vector.
- For example, the even parity bit can be generated by taking the exclusive OR of all the bits of a vector and inverting.

Continuous Assign and Always_comb



```
module sum_and_dif_A
   (output  logic [3:0]  result,
    input   logic [3:0]  a, b,
    input                select_plus);

   always_comb
      if (select_plus)
         result = a + b;
      else result = a - b;
endmodule: sum_and_dif_A

module sum_and_dif_B
   (output  logic [3:0]  result,
    input   logic [3:0]  a, b,
    input   logic        select_plus);

   assign result = (select_plus) ? a +b : a - b;
endmodule: sum_and_dif_B
```

Not Combinational

```
module notCombinational
    (input logic [3:0] a, b,
    output logic [3:0] sum,
    input logic         hold)
always_comb
     if (~hold)
        sum = a + b;
endmodule: notCombinational
```

- A synthesis tool would use latches gated by hold to implement module notCombinational.
- Most synthesis tools warn that the model is not stateless and that latches are inferred for the implementation.

- Input hold is a control signal specifying if the previous value of sum is to be remembered.
- If hold is asserted (it has value 1), then a new value of sum is not calculated - the old value is held - remembered.
- If hold is not asserted (it's equal to 0), then when inputs a or b change, the loop will be executed and sum will be updated.
- The value of sum will possibly change when a and/or b change.
- This behavior is not combinational
  - because it specifies that something (sum) needs to be remembered from execution to execution of the always_comb loop.

Stateless Functions

```
1   module notCombinational
2       (input logic [3:0] a, b,
3       output logic [3:0] sum,
4       input logic          hold)
6   always comb
7         if (~hold)
8           sum = a + b;
9   endmodule: notCombinational
```

- In Example, sum is not recomputed if hold is 1, a violation of this rule.
- Essentially, there is no else clause for the if on line 7.

- Tips for writing combinational descriptions are:
  - make sure that the model executes when any of its inputs changes. After all, if an input changes, the output might too. Using assign or always_comb statements insures this.
  - make sure the output(s) of the always_comb or assign statement is always recomputed and updated when an input changes. An assign statement guarantees that the output is updated, but when using an always_comb statement the designer must make sure of this.
  - make sure the always_comb block does not have any sequential side effects. e.g., it should not be incrementing a counter, for instance.

Testbenches

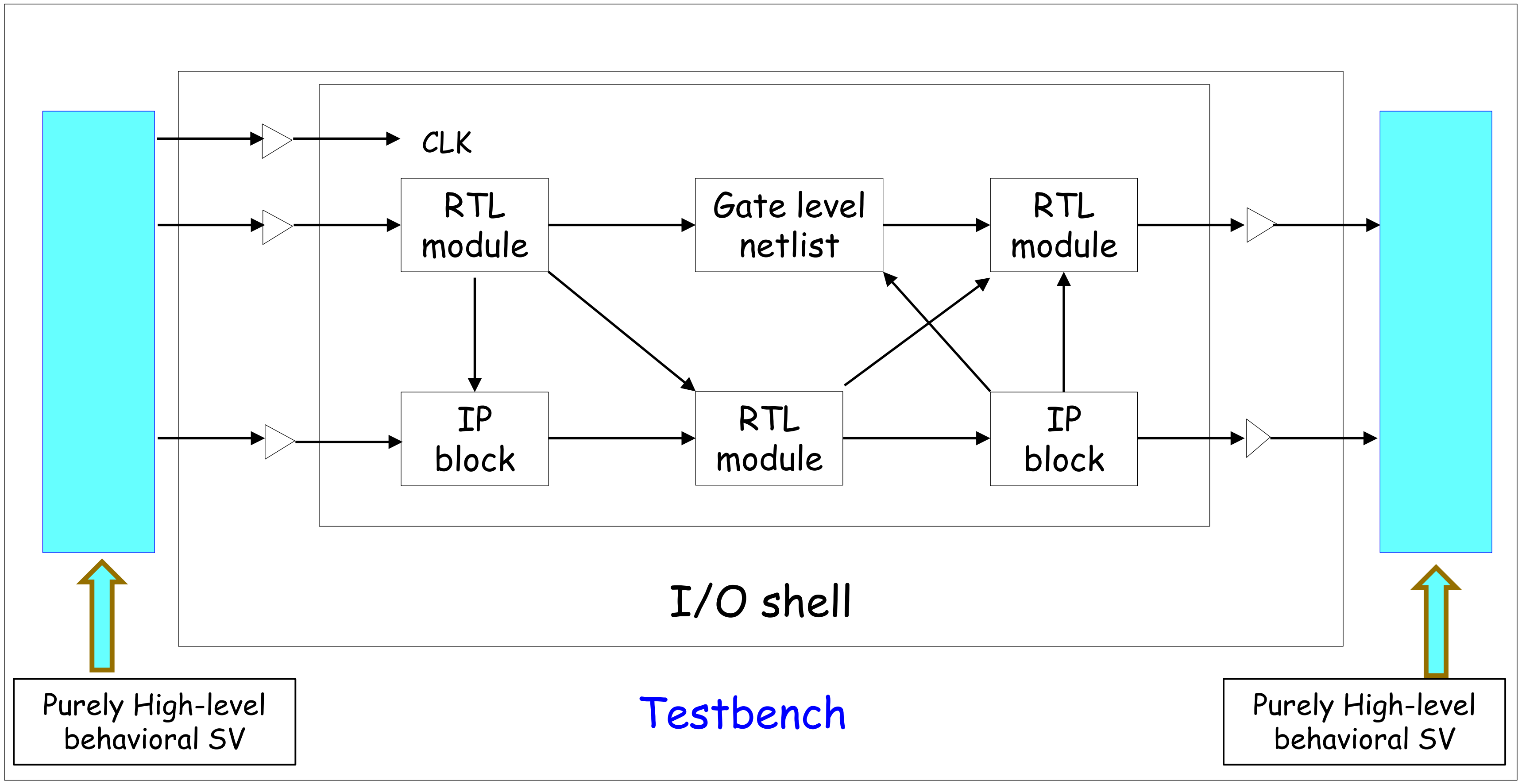
- Testbenches have a distinctive style.
  - A description of the entire world that affects the model we are testing, there are no inputs or outputs to the module.
  - The environment
- This is the characteristic of testbenches.

- "*A test bench is a software program written in any language for the purposes of exercising and verifying the functional correctness of a hardware model during simulation in a simulated environment."* Doug J. Smith, HDL Chip Design

Testbenches

- A test bench is an environment used to verify the correctness or soundness of a design or model.
- The testbench is a top level module that encompasses the entire design
- Testbenches are not synthesizable and therefore the entire scope of SystemVerilog can be used to write them.
- Two functions are generally performed in a testbench:
  - generation of input stimuli and
  - checking of results.

Testbenches



Testbenches

```
module And2 (output wire z,
             input wire x, y);
assign z = x & y;
endmodule
```

```
module TestAnd2;
    wire a,b,c;
    And2 g1 (c, a, b);
    initial
        begin
        a = '0;
        b = '0;
        #100ps a = '1;
        #50ps b = '1;
    end
endmodule
```

- An **initial** procedural block is declared, in which the initial values of a and b are defined.
    - After a delay of 100 ps, a is updated and 50 ps later b is updated.
    - Note the use of **begin** and **end** to bracket multiple statements.
- An **initial** block is executed once.
- A net should only be assigned a value from one block. It works in SystemVerilog.

Testbenches for Combinational Blocks

```
module TestNBitAdder;
parameter N = 4;
logic Cin, Cout;
logic [N-1:0] Sum, A, B;
adder #(N) s0 (.*);
initial
begin
        Cin = '0;
        A = 4'b0000;
        B = 4'b0000;
        #5ns A = 4'b1111;
        #5ns Cin = '1;
        #5ns A = 4'b0111;
        #5ns B = 4'b1111;
        #5ns Cin = '0;
end
endmodule
```

```
module adder #(parameter N = 4)
        (output logic [N-1:0] Sum, output logic Cout,
        input logic [N-1:0] A, B, input logic Cin);
        always_comb
        {Cout, Sum} = A + B + Cin;
endmodule
```

- The simple testbenches did not perform any checking.
- Moreover, input stimuli were generated using concurrent assignments.
- This style is fine for simple circuits, but is not appropriate for circuits with multiple inputs.

Testbenches

```
module TestNBitAdder;
parameter N = 4;
logic Cin, Cout;
logic [N-1:0] Sum, A, B;
adder #(N) s0 (.*);
initial
begin
      Cin = '0;
      A = 4'b0000;
      B = 4'b0000;
      #5ns A = 4'b1111;
      #5ns Cin = '1;
      #5ns A = 4'b0111;
      #5ns B = 4'b1111;
      #5ns Cin = '0;
end
endmodule
```

```
module adder #(parameter N = 4)
      (output logic [N-1:0] Sum, output logic Cout,
      input logic [N-1:0] A, B, input logic Cin);
      always_comb
      {Cout, Sum} = A + B + Cin;
endmodule
```

- The instantiation of the adder has a parameter (N) and uses a wild card (.*) to connect signals.
- This is allowable if the wire and variable names in the testbench are *exactly* the same as those in the module being instantiated.
- Time is relative (wait for 5 ns at a time), rather than absolute.
- **initial** indicates a procedure that is executed once – Not an initialization of variables.

Testbenches

```
module TestNBitAdder;
parameter N = 4;
logic Cin, Cout;
logic [N-1:0] Sum, A, B;
adder #(N) s0 (.*);
initial
begin
        Cin = '0;
        A = 4'b0000;
        B = 4'b0000;
        #5ns A = 4'b1111;
        #5ns Cin = '1;
        #5ns A = 4'b0111;
        #5ns B = 4'b1111;
        #5ns Cin = '0;
end
endmodule
```

```
initial
begin
        Cin = 0;
        A = 0;
        B = 0;
        #5ns A = 15;
        #5ns Cin = 1;
        #5ns A = 7;
        #5ns B = 15;
        #5ns Cin = 0;
end
```

- For example,
  - we try to add "0111" to "0000" with a carry-in bit of '1'.
  - The simulation tells us that the sum is "1000" with a carry-out bit of '0'.
- Instead, we could use integers.
- Now we can see that 7+0+1 is equal to 8 (with no carry out).

Testbenches

```
initial
begin
    Cin = 0;
    A = 0;
    B = 0;
    #5ns A = 15;
    #5ns Cin = 1;
    #5ns A = 7;
    #5ns B = 15;
    #5ns Cin = 0;
end
```

- We could let the testbench itself check the addition.
  - generate an error signal when unexpected behavior occurs.
- simply add an error signal:
  - **logic** error;
- together with a process that is triggered whenever one of the outputs from the adder changes:
  - **always** @(Cout, Sum)
  - error = ((A + B + Cin) != Sum);
- The idea is to check the operation by performing it in a different way.

Blocking Assignments

- It updates immediately the left-hand side.
- It models the linear order of execution,
  - as the value propagation in the data flow model of combinational logic.

References

- Digital System Design with SystemVerilog by Mark Zwolinski. Prentice Hall, 2009.
- Digital Design and Computer Architecture, by David Harris, Sarah Harris. Morgan Kaufmann, 2012.
- Logic Design and Verification Using SystemVerilog (Revised) by Donald Thomas, 2016.
- Language Reference Manual (LRM) – IEEE 1800-2005.