Verilog

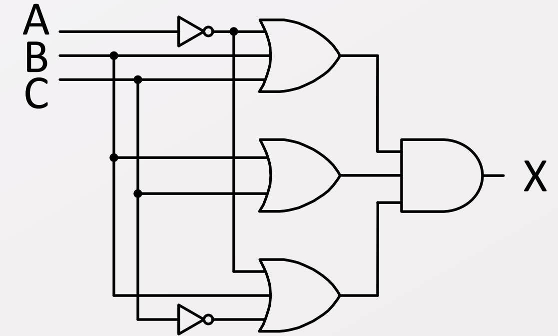
An Introduction

Introduction

Verilog is a “hardware design language” (**HDL**) that uses C-like syntax and constructs. Verilog is used to describe digital circuits and systems.

```
module example (input A, B, C, output X);
  assign X = (!A | B | C) & (B | C) & (!A | B | !C);
endmodule
```

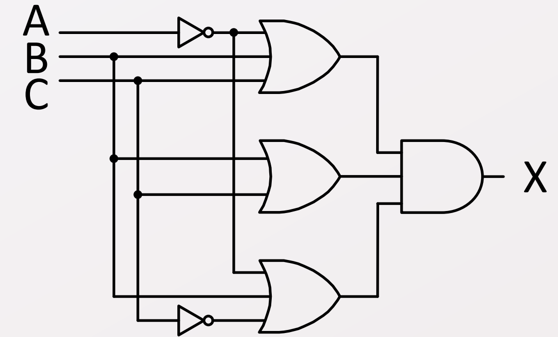
This is an example Verilog source file



Introduction

Verilog can be ***simulated*** to check logic functions…

```
module example (input A, B, C, output X);
  assign X = (!A | B | C) & (B | C) & (!A | B | !C);
endmodule
```



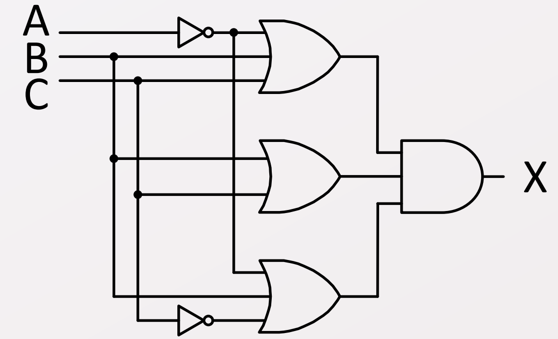

```
module simple_tb();
reg A, B, C;
example cut(.A(A),.B(B),.C(C),.X(X));
  initial begin
    A = 0; B = 0; C = 0; #10;
    A = 0; B = 0; C = 1; #10;
    A = 0; B = 1; C = 0; #10;
       .
       .
    A = 1; B = 1; C = 1; #10;
  end
endmodule
```

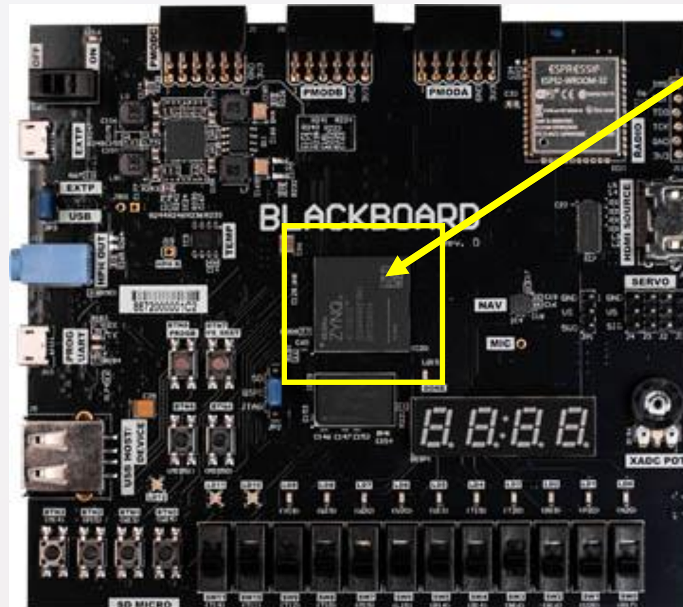
This file is a “testbench”

Introduction

Verilog can be ***synthesized*** into a physical circuit…

```
module example (input A, B, C, output X);
  assign X = (!A | B | C) & (B | C) & (!A | B | !C);
endmodule
```

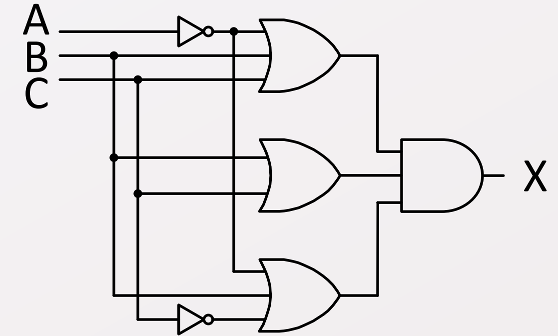




Introduction

Verilog can describe a circuit's high-level ***behavior***

```
module example (input A, B, C, output X);
  assign X = (!A | B | C) & (B | C) & (!A | B | !C);
endmodule
```

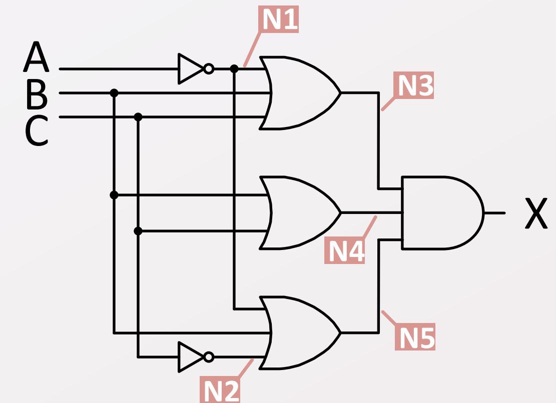


Behavioral descriptions use abstract assignments with no information to indicate how a circuit might be constructed

Introduction

Verilog can describe a circuit's high-level ***behavior…***

```
module example (input A, B, C, output X);
  assign X = (!A | B | C) & (B | C) & (!A | B | !C);
endmodule
```



… or its low-level ***structure***

```
module example (input A, B, C, output X);
  wire n1, n2, n3, n4, n5;
  not (n1, A);
  not (n2, C);
  nor (n3, n1, B, C);
  nor (n4, B, C);
  nor (n5, n1, B, n2);
  nor (X, n3, n4, n5);
endmodule
```

Structural descriptions are netlist-like, and define how components are connected to make the circuit

Introduction

Verilog ***is not*** a sequential programming language like C or Java. A sequential language describes *step-by-step* algorithms, and the sequence of instructions matters.

```
void bubbleSort(int a[], int n)
{
  int i, j;
  for (i = 0; i < n-1; i++)
    for (j = 0; j < n-i-1; j++)
      if (a[j] > a[j+1])
        {
            int temp = a[j];
            a[j] = a[j+1];
            a[j+1] = temp;
         }
}
```

This sorts

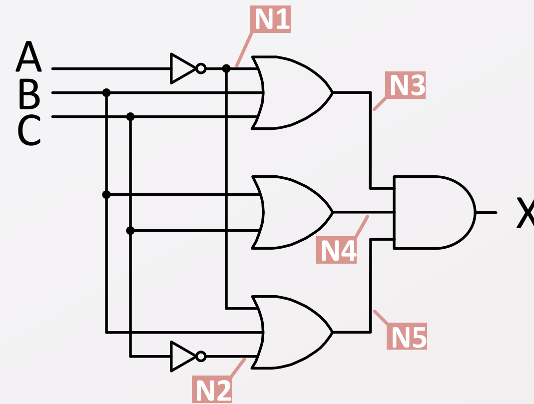
```
void bubbleSort(int a[], int n)
{
  int i, j;
  for (i = 0; i < n-1; i++)
    for (j = 0; j < n-i-1; j++)
      if (a[j] > a[j+1])
        {
            a[j] = a[j+1];
            int temp = a[j];
            a[j+1] = temp;
         }
}
```

This doesn’t

Introduction

Verilog is **concurrent**. Simulations are event-driven, so statement order does not matter.

```
module netlist(
    input A, B, C,
    output X
    );

wire N1,N2,N3,N4,N5;

assign N1 = !A;
assign N2 = !C;
assign N3 = N1 | B | C;
assign N4 = B | C;
assign N5 = N1 | B | N2;
assign X = N3 & N4 & N5;

endmodule
```



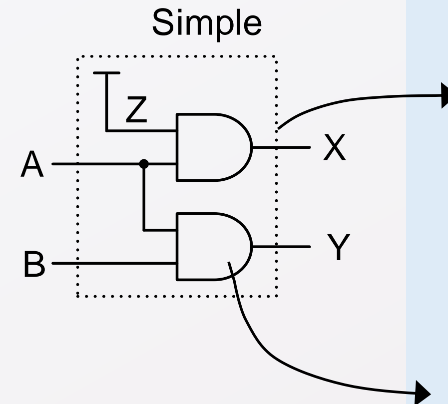
Both descriptions will simulate and synthesize identically

```
module netlist(
    input A, B, C,
    output X
    );

wire N1,N2,N3,N4,N5;

assign N5 = N1 | B | N2;
assign N1 = !A;
assign X = N3 & N4 & N5;
assign N2 = !C;
assign N3 = N1 | B | C;
assign N4 = B | C;

endmodule
```

Verilog Modules – A first look

Verilog source files contain **modules.** All circuits are described in modules – they are the building blocks of Verilog designs**.**

Modules include *port declarations* that identify input and output signals, and *assignment* statements that describe the circuit.




```
module simple(
    input A, B,
    output X, Y
);

wire Z = 1b’1;

assign X = A & Z;
assign Y = A & B;

end module
```

The keyword **module** is followed by the ***name*** and the port list. The name can be any legal text string. Modules are identified by this name, not by their file name.

Ports define the signals that transport information in and out of modules. Every port signal has a name and a type (input, output, or inout).

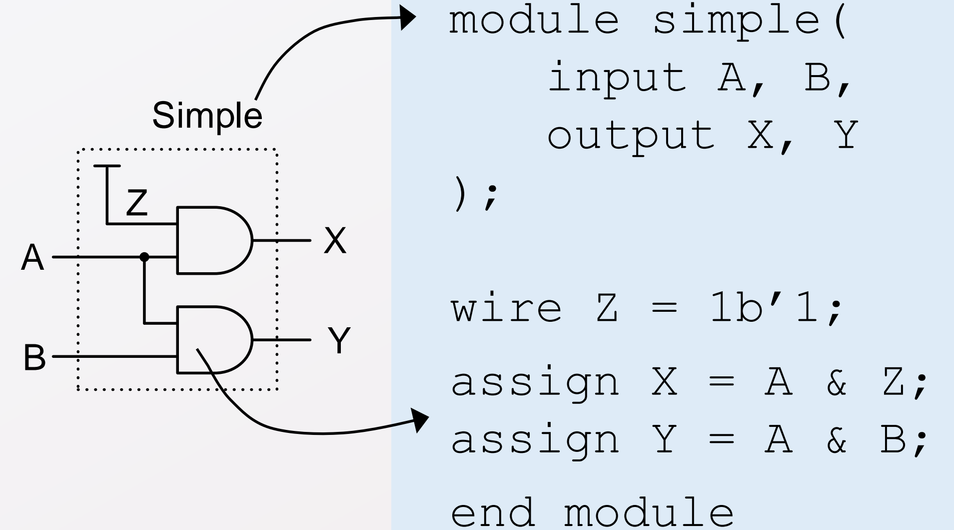
Every module can be in its own file, or any number can be in a single file.

Verilog Modules – A first look

Designs can use multiple modules, for the same reason C programs use subroutines. A top-level module contains all other modules.

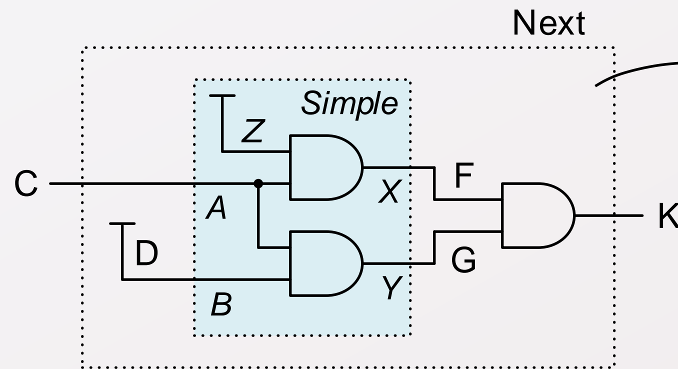
Wires transport signals inside a module. They must be declared before use.

Any module can be used as a component in any other module.




```
module simple(
    input A, B,
    output X, Y
);

wire Z = 1b’1;
assign X = A & Z;
assign Y = A & B;
end module
```

A port map statement “instantiates” an included module and defines connections



.component_port(topmodule_port)

```
module next(
    input C,
    output K
);

wire F,G,D = 1’b1;

simple(.A(C),.B(D),.F(X),.G(Y));
assign K = F & G;
end module
```

Identifiers and Numbers

Identifiers: space-free upper and lower case letters (Verilog ***is*** case sensitive), numbers, _, and $. Can't begin with a digit; limited to 1024 characters.

Numbers: <size>' <base> <value>

size: decimal number that specifies number of bits

base: ' character followed by b (binary), d (decimal), h (hex)

value: set of digits

Examples: **`x=4'b0101`** **`y4_b=4'd12`** **`z$12=16'hAB12`**

Spaces, tabs, and new lines and form feeds can separate words and are ignored.

Note: If no size is given, a 32-bit decimal number is assumed

Operators

<table>
<tr><td>+</td><td>Add</td><td>!</td><td>Negation</td><td>></td><td>Greater than</td></tr>
<tr><td>-</td><td>Subtract</td><td>~</td><td>Bit-wise negation</td><td><</td><td>Less than</td></tr>
<tr><td>*</td><td>Multiply</td><td>&</td><td>Bit-wise And</td><td>>=</td><td>Greater or equal</td></tr>
<tr><td>/</td><td>Divide</td><td>|</td><td>Bit-wise Or</td><td><=</td><td>Lesser or equal</td></tr>
<tr><td>%</td><td>Modulus</td><td>^</td><td>Bit-wise Xor</td><td>==</td><td>Case equality</td></tr>
<tr><td><<</td><td>Shift left</td><td>&&</td><td>Logical And</td><td>!=</td><td>Case inequality</td></tr>
<tr><td>>></td><td>Shift right</td><td>||</td><td>Logical Or</td><td>?</td><td>Conditional</td></tr>
<tr><td>[ ]</td><td>Bit select</td><td>{ }</td><td>Concatenation</td><td>{{ }}</td><td>Replication</td></tr>
</table>

Data Types

Verilog includes two data types to aid circuit model construction: **Net** types to model wires, and **Variable** types that can store data.

Net types model wires, and do not store values. They are continuously driven, and their value is determined by the values of their drivers. Net types are used to transport data between circuit elements.

Variables types store values. Variables are used when a circuit's next state depends on current state. A **procedure** checks relevant signals and updates a variable when certain conditions are met.

Net and variable values can only be 1, 0, X, and Z.

Net Types

Net types include wire, tri, supply0 (0), and supply1 (1).

The **wire** type (most common) is used for nets that are driven by a single gate. They transport signals between circuit elements within a module, between modules, or between modules and I/O pins. Wires are assigned new values using the keyword **assign**.

The **tri** type is very similar to wire and can be used to model **three-state** signals, where multiple gates can drive the same net. It is not commonly used.

Supply1/supply0 model Vdd and Gnd connections, and are also not commonly used.

Wire types

The **Wire** type models physical wires. Wires must be declared before use.

```
wire  a, b, c;   //multiple wires can be defined at once
wire  d = 1'b0;  //declares wire d and assigns to 0
```

Busses are groupings of wires

```
wire   [3:0] bus4;             //bus definition
wire   [4:0] X, [7:0] Y;       //multiple definitions per line OK
wire   [7:0] Z = 0;            //value can be assigned
```

Arrays are groupings of busses

```
wire   [7:0] A [3:0];        //Define four 8-bit busses A[1], A[2],…
```

Assigning values to wires

Wires are assigned new values using an **assign** statement.

```
assign X = 1;                assign Y = 4'b0011;
assign Z = A & B;            assign W = (A | B) & ~(C | ~D);
```

Busses can also be assigned using an **assign** statement. Busses are assigned “left-most to left-most”. They do not need to be the same size.

```
wire [7:0] X, Y   // Declare two 8-bit busses
wire [4:0] I, J   // Declare two 5-bit busses
wire [2:0] Z      // Declare 3-bit bus

assign X = Y      // X[7:0] <= Y[7:0];
assign X = I      // X[7:3] <= Z[4:0]; X[2:0]
assign X = {Z,J}  // X[7:5] <= Z and X[4:0] <= J
assign X = {I,J}  // X[7:3] <= I and X[2:0] <= J[4:2]
```

{,} is concatenation operator

Wires connecting one module to another must have matching widths.

Assigning values to wires

A conditional (or **ternary**) operator can be used to drive a wire from one of two sources. Y is the output of a 2:1 mux:

```
assign Y = (Sel == 0) ? I0 : I1;
```

The sources can be logic functions:

```
assign Y = (X == 0) ? (A|B) : (C|(D&!E));
```

Nesting is permitted:

```
assign Y = (X1 == 0) ? I1 : ((X2 == 1) ? I2 : 0);
```

Note: parenthesis are optional, but they help readability

Variables

Variable types (reg, integer, real, time, realtime) define memory in the simulator (not in the circuit being described). Variables let the simulator preserve the state of a net, which is needed when a **procedure** determines how to drive a signal (e.g., as the result of an if statement).

Variables are modified using **blocking** (=) or **nonblocking** (<=) assignments . Blocking assignments update variables immediately, before any further statements are executed (like C).

Non-blocking assignments schedule updates to occur at the end of the procedure - this makes them concurrent. Subsequent statements are not blocked, so there is no implied sequence and their order does not matter (the order of blocking assignments very much matters).

Variable types

Only **reg** and **integer** types define synthesizable variables – the others are used only in simulations. Like wires, variables must be declared before use.

```
reg X, Y;           // multiple regs can be assigned at once
reg [15:0] I;       // I is a 16-bit variable
reg Z = 1           // Z is a variable initialized to 1
integer J;          // J is a 32-bit integer variable
integer K = 27;     // K is a 32-bit integer initialized to decimal 27
```

Procedural Blocks

Variables are assigned in “procedural blocks”. Procedural blocks house simulator routines that modify variables. Variables can only be changed in procedural blocks. Wires cannot be changed in procedural blocks.

Statements in procedure blocks execute in order (sequentially, like C). Procedures can include **if** and **case** statements – if **if/case** statements are not needed, then a procedure is not needed.

Example: Assign Q=D if a rising clock edge occurred. A procedure must check a condition (clk edge?) before the assignment happens:

```
if (posedge(clk)) Q = D;
```

Procedural Blocks

Two primary procedural blocks are available: the **initial** block used to define simulator inputs, and the **always** block used to define circuits.

An **initial** block always runs once at the start of a simulation. It is used to define simulator inputs, and will be discussed later.

An always block executes whenever a ***sensitivity list*** signal changes (so the block is *always* active watching for a change in the sensitivity list).

```
always @ (X,Z,C,D) begin
     if (X == 1) Y = Z;
     W = (C | D);
end
```

The sensitivity list follows **always** and the **@** sign. If one of the signals X, Z, C, or D change, the block executes.

Procedural Assignments

```
always @ (X,Z,C,D) begin
    if (X == 1) Y = Z;
    W = (C | D);
end
```

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

If more than one signal is in the sensitivity list, parenthesis are required.

Function calls like “posedge(clk)” may be used in the list. Function calls must be in parenthesis.

Note: The **posedge** function is very commonly used when writing code to infer a flip-flop.

Procedural Assignments

```
always @ (X,Z,C,D) begin
    if (X == 1) Y = Z;
    W = (C | D);
end
```

In the example code, Y and W are being assigned values inside the always block, and so must be memory variables (type **reg**).

Y gets a new value (Z) if X == 1, otherwise Y will remain unchanged. How can Y remain unchanged? Memory! (a flip-flop).

Unspecified conditions in an if/case always infer memory in a synthesized circuit.

If statement

If/else statements can only be used inside a procedure block. They assert signals in response to specified conditions.

They are commonly used to infer memory, but they can describe combinational logic if all conditions are checked.

Any conditions with unspecified action ***will*** result in memory.

```
always @ (A) begin
  if (A==1’b1) X = I1;
end
```

```
always @ (A) begin
  if (A == 1’b0) X = I1;
  else if (A==1’b1) X = I2;
end
```

```
always @ (A) begin
  if (A) begin
     X = I1; Y = I2;
  end
     else X = I2;
end
```

```
always @ (A) begin
  if (A==1’b1) X = I1;
  else X = 1’b0;
end
```

If statement

If/else statements can only be used inside a procedure block. They assert signals in response to specified conditions.

They are commonly used to infer memory, but they can describe combinational logic if all conditions are checked.

Any conditions with unspecified action ***will*** result in memory.

```
always @ (A) begin
  if (A==1'b1) X = I1;
end
```

Memory

```
always @ (A) begin
  if (A == 1'b0) X = I1;
  else if (A==1'b1) X = I2;
end
```

```
always @ (A) begin
  if (A) begin
     X = I1; Y = I2;
  end
     else X = I2;
end
```

Memory

```
always @ (A) begin
  if (A==1'b1) X = I1;
  else X = 1'b0;
end
```

Case statement

The case statement is essentially a compound if statement, and so the same rules apply: if you don’t want memory, specify all conditions.

```
always @(I)
case(I)
	3'd0: Y = 8'd1;
	3'd1: Y = 8'd2;
	3'd2: Y = 8'd4;
	3'd3: Y = 8'd8;
	3'd4: Y = 8'd16;
	3'd5: Y = 8'd32;
	3'd6: Y = 8'd64;
endcase
end;
```

```
always @(I)
case(I)
	3'd0: Y = 8'd1;
	3'd1: Y = 8'd2;
	3'd2: Y = 8'd4;
	3'd3: Y = 8'd8;
	3'd4: Y = 8'd16;
	3'd5: Y = 8'd32;
	3'd6: Y = 8'd64;
	3'd7: Y = 8'd128;
endcase
end;
```

Case statement

The case statement is essentially a compound if statement, and so the same rules apply: if you don’t want memory, specify all conditions.

```
always @(I)
case(I)
      3'd0: Y = 8'd1;
      3'd1: Y = 8'd2;
      3'd2: Y = 8'd4;
      3'd3: Y = 8'd8;
      3'd4: Y = 8'd16;
      3'd5: Y = 8'd32;
      3'd6: Y = 8'd64;
endcase
end;
```

Memory

```
always @(I)
case(I)
      3'd0: Y = 8'd1;
      3'd1: Y = 8'd2;
      3'd2: Y = 8'd4;
      3'd3: Y = 8'd8;
      3'd4: Y = 8'd16;
      3'd5: Y = 8'd32;
      3'd6: Y = 8'd64;
      3'd7: Y = 8'd128;
endcase
end;
```

Creating memory (flip-flops/latches)

Variable types ***may*** or ***may not*** synthesize to a memory device in a physical circuit.

Memory is inferred from **if/case** statements when assignments are not defined for all conditions.

If you want memory, check for posedge(clk), and use a non-blocking assignment (<=).

```
always @ (posedge(clk), rst)
    if (rst) Q <= 0;
    else Q <= D;
end
```

Avoiding unintentional memory/latches

If you do not want memory, do not check posedge(clk), account for all possible conditions of sensitivity list variables, and use blocking assignment (=). Using (*) in the sensitivity list helps as well – it triggers the always block if any variable used in the block changes.

Use defaults:

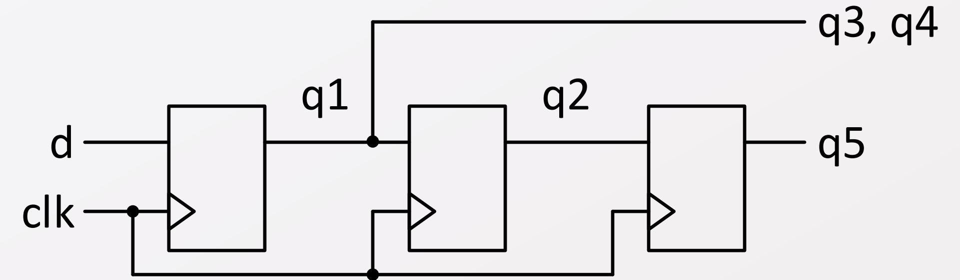
```
reg [1:0] y;

always @ (A, B) begin
      y = 2’b00;
      if (A) y = 2’b01;
    else if (B) y = 2’b11;
end
```

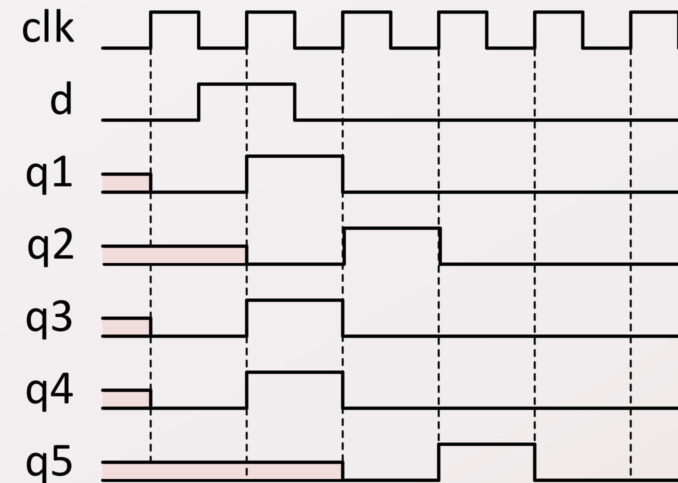
Check all conditions:

```
reg [1:0] y;
wire [1:0] x;
always @ (*) begin
    if (x==2’b00) y = 2’b01;
        else if (x==2’b10) y = 2’b11;
    else y = 2’b00;
end
```

Procedural Assignment Examples

```
module blocking(
  input d, clk,
  output reg q1,q2,q3,q4,q5
  );

always @ (posedge clk) begin
  q1 <= d;
  q2 <= q1;
  q3 = d;
  q4 = q3;
  q5 = q2;
end
endmodule
```





Procedural Assignments

Don’t assign the same variable from more than one always block.

A module can have any number of always blocks. Statements in a procedural block are sequential, but the blocks themselves are concurrent to other blocks and assign statements.

If an always block has more than one line, **begin** and **end** are needed.

Memory circuits can only be inferred inside always blocks.

Combinational circuits never need to use an always block, but they can (if/case can make circuits clearer and easier to read, but watch out!).

Do not mix <= and = assignments in same always block (better to use two blocks)

Operators: A more detailed look

+	Add	!	Negation	>	Greater than
-	Subtract	~	Bit-wise negation	<	Less than
*	Multiply	&	Bit-wise And	>=	Greater or equal
/	Divide	|	Bit-wise Or	<=	Lesser or equal
%	Modulus	^	Bit-wise Xor	==	Case equality
<<	Shift left	&&	Logical And	!=	Case inequality
>>	Shift right	||	Logical Or	?	Conditional
[ ]	Bit select	{ }	Concatenation	{{ }}	Replication

Arithmetic Operators: add (+), subtract (-), multiply (*), divide (/), modulus (%);

```
module arith(
    input [7:0] A,
    input [7:0] B,
    output [15:0] Y1, Y2, Y3, Y4, Y5
    );

assign Y1 = A + B;   // Y1[8:0]  = A + B; Y1[15:9] = 0
assign Y2 = A - B;   // Y2[7:0]  = A - B; Y1[15:8] = 0
assign Y3 = A * B;   // Y3[15:0] = A * B
assign Y4 = A / B;   // Y4[15:0] = A / B fractional part discarded
assign Y5 = A % B;   // Y5[7:0]  = A mod B remainder

endmodule
```

If any operand bit has a value ”x”, the result of the expression is all ”x”.

For modulus operation, results take the sign of the first operand.

Arithmetic Operators with negative numbers

Adding and subtracting two's compliment numbers works as expected, but multiplying 2's compliment numbers requires a different hardware circuit.

Using the keyword "signed" will ensure values are represented in 2's compliment and that proper circuits are used.

```
wire [7:0] A,B;
wire [15:0] Y;
assign Y = A * B;
```

```
wire signed [7:0] A,B;
wire signed [15:0] Y;
assign Y = A * B;
```

Logical/Bit-wise Logic Operators: and (&&, &), or (||, |), not (!, ~), xor (^), xnor(^~)

```
module logic(input [7:0] A, B, output Y1, Y2, Y3, [7:0] Y4, Y5, Y6);

assign Y1 = A && B;
assign Y2 = A || B;
assign Y3 = !A;

assign Y4 = A & B;
assign Y5 = A | B;
assign Y6 = ~A;
```

		Logical Operators (result in 0,1,x)			Bit-wise Operators (every bit)		
A[7:0]	B[7:0]	Y1	Y2	Y3	Y4[7:0]	Y5[7:0]	Y6[7:0]
7	0	0	1	0	0	7	F8
0	0	0	0	1	0	0	FF
5	5	1	1	0	5	5	FA
5	F	1	1	0	5	F	FA

Logic operators (&&, ||, !) return one bit. Bit-wise operators (&, |, ~) return vectors and operate bit by bit (leftmost to leftmost); mismatched length operands are zero-extended

Using bit-wise operators with a LHS wire and RHS bus operates only on bus LSB

Using logical operators with a LHS bus and RHS wires operates only on bus LSB

Expressions like (A == 2) && (B == 3) evaluate to 1 if all comparisons are true

Reduction Operators: and (&), nand (~&), or (|), nor (~|), xor (^), xnor (^~)

```
module reduction(input [7:0] A, output Y1, Y2, Y3);

assign Y1 = &A;
assign Y2 = |A;
assign Y2 = ~|A;
assign Y3 = ^A;
```

A[7:0]	Y1	Y2	Y3	Y4
FF	1	1	0	0
01	0	1	0	1
00	0	0	1	0

Reduction operators act on each bit of an input vector. Operations proceed right-to-left on all operand bits and return a 1-bit result.

Relational Operators: equal (==), not equal (!=), greater than (>), less than (<), less-than-or-equal (<=), greater than or equal (>=), case equality (===, !==)

```
module equal(input [7:0] A, B, output Y1, Y2, Y3, [7:0] Y4, Y5, Y6);

assign Y1 = (A==B) ? 1 : 0;
assign Y2 = (A!=B) ? 1 : 0;
assign Y3 = (A>=B) ? 1 : 0;

if (A < B) Y4 = 8'hFF;

case (B)
    A > B :  Y5 = 8'hAA;
```

Operands are compared bit by bit (leftmost to leftmost), with zero filling for unequal lengths

Synthesize to very large, very slow circuits

Shift Operators: left (<<), right (>>), arithmetic right (>>>)

```
module shift(input [7:0] A, [2:0] amount, output [7:0] Y1);

assign Y1 = (A << amount);
assign Y2 = (A >> amount);
assign Y3 = (A >>> amount);
```

A[7:0]	Amount[2:0]	Y1[7:0]	Y2[7:0]	Y3[7:0]
F0	010	C0	3C	FC
0F	010	3C	03	03
0A	100	A0	00	00
80	111	00	01	FF

Shift: Vacated bits zero-filled

Arithmetic right shift: Vacated bits filled with copies of MSB (sign extend)

Concatenation and Replication Operators:{,} and {{}}

```
module reduction(input D,E,F, [4:0] A,[3:0] B,[2:0] C,output [7:0] Y1,Y2,[9:0] Y3);

assign Y1 = {A, C};     // Y1[7:3] = A, Y1[2:0] = C
assign Y2 = {C, A};     // Y2[7:5] = C, Y1[4:0] = A
assign Y1 = {A, B};     // Y1[7:3] = A, Y1[2:0] = B[3:1]
assign Y2 = {D,E,A,F};  // Y2[7]= D, Y2[6]= E, Y2[5:1]= A, Y2[0] =F
assign Y1 = {2{D},2{B}} // Y1[7:6] = D, D, Y2[5:0] = B, B
```

Joins/concatenates busses and/or wires bit by bit, leftmost to left most.

Replication allows repetitive concatenation

A quick review

Combinational (wire) assignments use **assign**; variable assignments use an **always block.**

Wire assigns are **continuous**; variables are updated in **always blocks** and may result in memory.

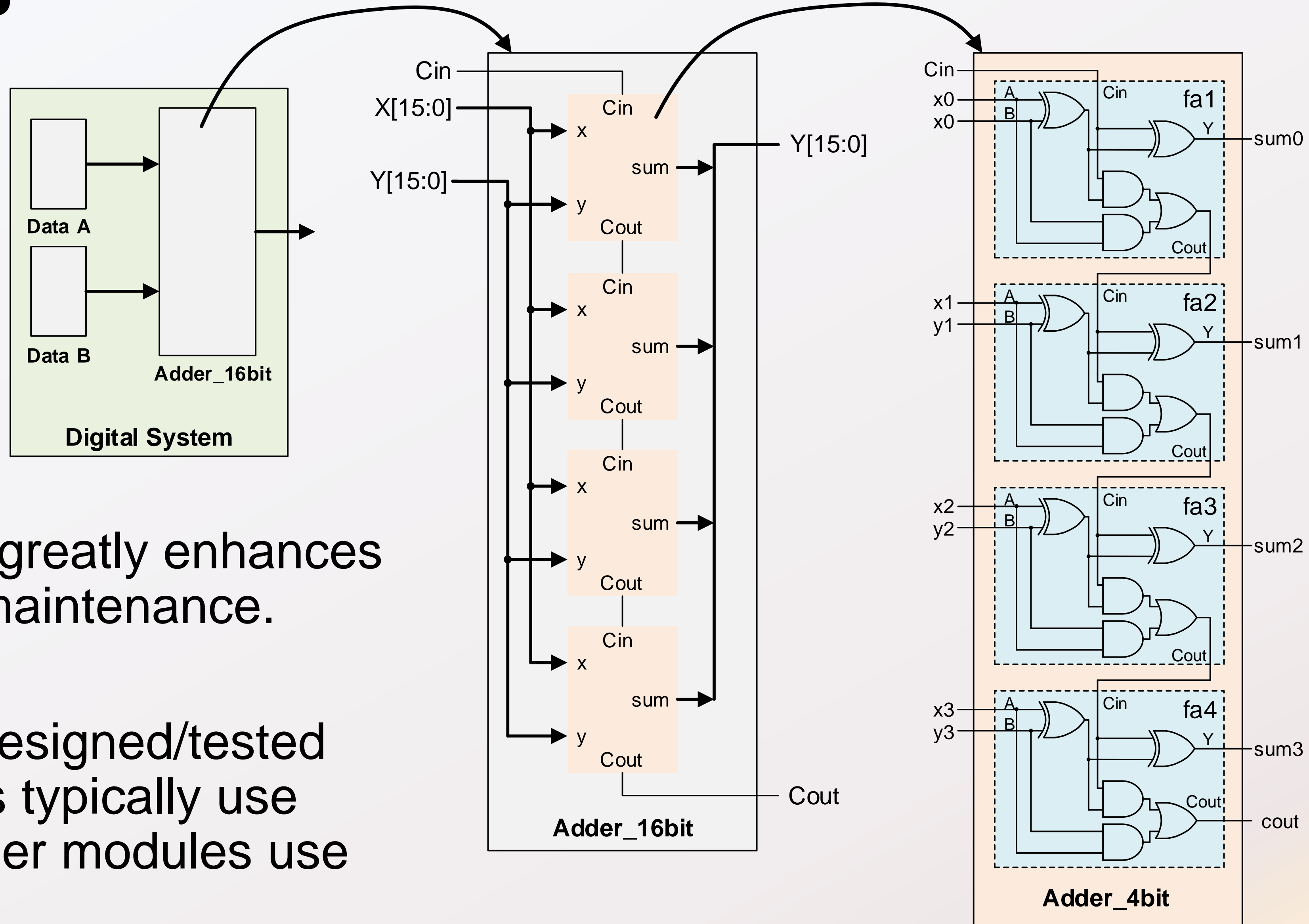
Assign statements/always blocks are **concurrent.** This example contains four concurrent statements/blocks.

Z, W, and T are variables and must be type reg. Note they are **typecast** to reg in the module statement.

```
module demo(
  input A,B,C,clk,
  output X, Y, reg Z, W, T
);

wire D;

assign X = A & B;
assign Y = X | C;

always @ (clk) Z <= B &! C;

always @ (!clk) begin
  T <= A;
  if (B == C) W <= A;
  end

endmodule
```

Hierarchical Design

Larger systems use structural Verilog and hierarchical design. A top-level module instantiates lower-level modules. Complex designs may have several levels of hierarchy.



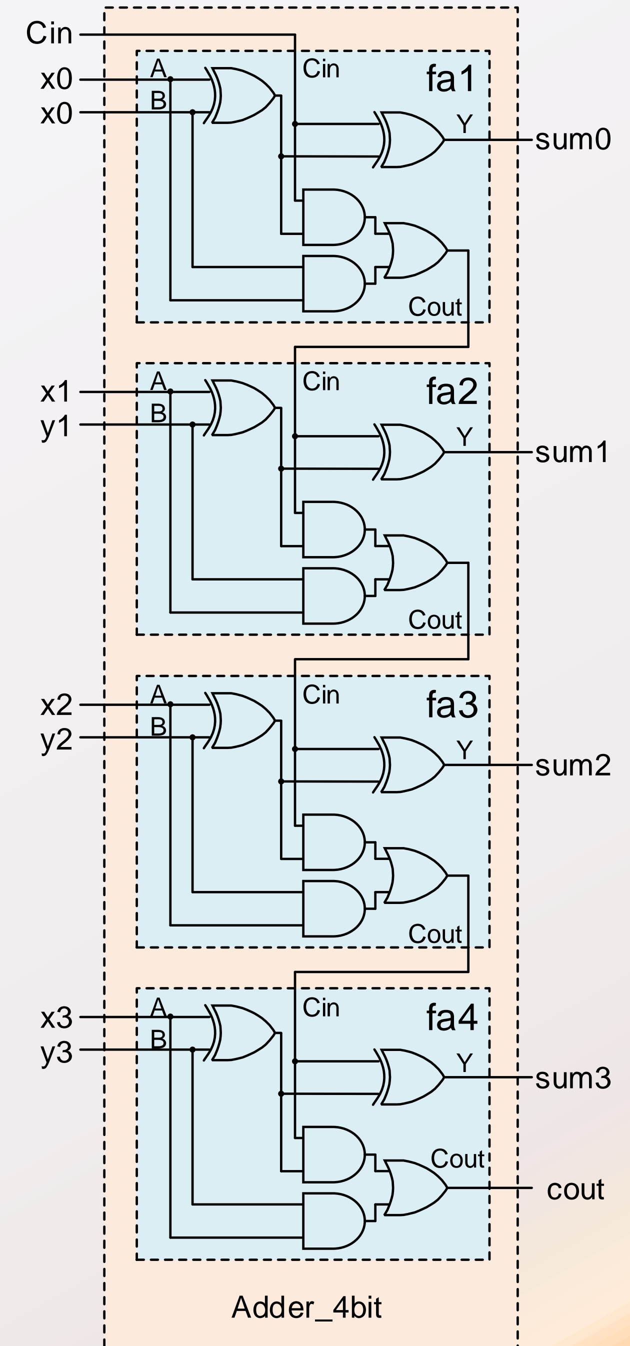
A good **partition** (hierarchy) greatly enhances debugging, readability, and maintenance.

Modules at every level are designed/tested independently. Leaf modules typically use **behavioral** design, and higher modules use **structural** design.

Structural Verilog

```
module full_adder(
    input a,b,cin;
    output y,cout
);
    wire s1,c1,c2;

    xor(s1,a,b);
    xor(y,cin,s1);
    and(c1,s1,cin);
    and(c2,a,b);
    or(cout,c1,c2);
endmodule

module adder_4bit(
    input [3:0] x,y;
    input cin;
    output[3:0] sum;
    output cout
);
    wire c1,c2,c3;

    full_adder fa1(.a(x[0]),.b(y[0]),.cin(cin),.y(sum[0]),.cout(c1));
    full_adder fa2(.a(x[1]),.b(y[1]),.cin(c1),.y(sum[1]),.cout(c2));
    full_adder fa3(.a(x[2]),.b(y[2]),.cin(c2),.y(sum[2]),.cout(c3));
    full_adder fa4(.a(x[3]),.b(y[3]),.cin(c3),.y(sum[3]),.cout(cout));
endmodule
```

This example uses the Verilog “built in” logic modules of and, or, and xor. These module instantiations use “positional association”, which means the order of signals must match in the module and in the instantiation – they are matched left to right. In this case, the format is output name, followed by inputs.



Structural Verilog

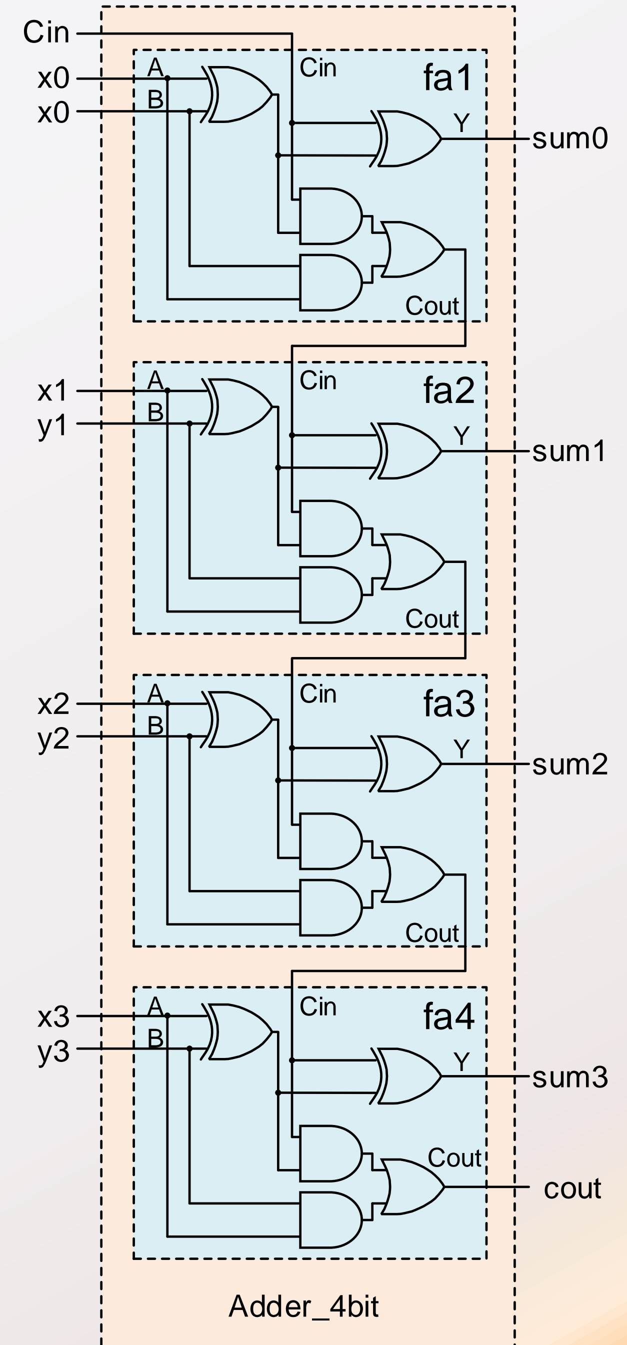
```
module full_adder(
    input a,b,cin;
    output y,cout
);
    wire s1,c1,c2;

    xor(s1,a,b);
    xor(y,cin,s1);
    and(c1,s1,cin);
    and(c2,a,b);
    or(cout,c1,c2);
endmodule
```

This example uses the Verilog “built in” logic modules of and, or, and xor. These module instantiations use “positional association”, which means the order of signals must match in the module and in the instantiation – they are matched left to right. In this case, the format is output name, followed by inputs.

```
module adder_4bit(
    input [3:0] x,y;
    input cin;
    output[3:0] sum;
    output cout
);
    wire c1,c2,c3;

    full_adder fa1(.a(x[0]),.b(y[0]),.cin(cin),.y(sum[0]),.cout(c1));
    full_adder fa2(.a(x[1]),.b(y[1]),.cin(c1),.y(sum[1]),.cout(c2));
    full_adder fa3(.a(x[2]),.b(y[2]),.cin(c2),.y(sum[2]),.cout(c3));
    full_adder fa4(.a(x[3]),.b(y[3]),.cin(c3),.y(sum[3]),.cout(cout));
endmodule
```

This is an example to illustrate a point. You could implement a 4-bit adder using behavior Verilog:

assign sum = a + b + cin;



Functions

Verilog functions are similar to functions in other languages: they encapsulate often-used code, and they return a value that can be used in an expression.

They can have any number of inputs, and must have at least one. Positional association is used.

Non-blocking assignments are not allowed. Re-entry is allowed with keyword “automatic”.

```
module func_example(input [7:0] A, B, output [7:0] Y);

function [7:0] sum (input [7:0] x, y);
  sum = x + y;
endfunction

assign Y = sum (A, B);

endmodule
```

Observations when creating source files

Source files describe circuits, and circuits are defined in **modules**.

Source files use the following format:

```
module module_name (port list)
declarations (wires, regs, functions)
items        (assign statements, always blocks, instantiations)
endmodule
```

The port list defines the signals that transport information in and out of modules. Every port signal has a name and a type (input, output, or inout).

Observations when creating source files

Modules can be simple (e.g., one line) or complex (hundreds of lines), and any module can be used as a component in any other module. Many modules can be in a single source file, or they can all be in their own source file.

Simple systems can use behavioral code, in a single module, in a single source file

For anything beyond the simplest designs, partition your design into sensible blocks, write a (behavioral) module for each block, and use structural Verilog at the top level to assemble the blocks.

Try to visualize the circuit you are defining when writing Verilog code.

Creating a Verilog design: A 4-bit adder

```
module FA(
   input A,B,Cin,
   output S,Cout);
   wire s1,c1,c2,c3;
   xor(s1,A,B);
   xor(S,Cin,s1);
   and(c1,A,B);
   and(c2,A,Cin);
   and(c3,B,Cin);
   or(Cout,c1,c2,c3);

endmodule
```

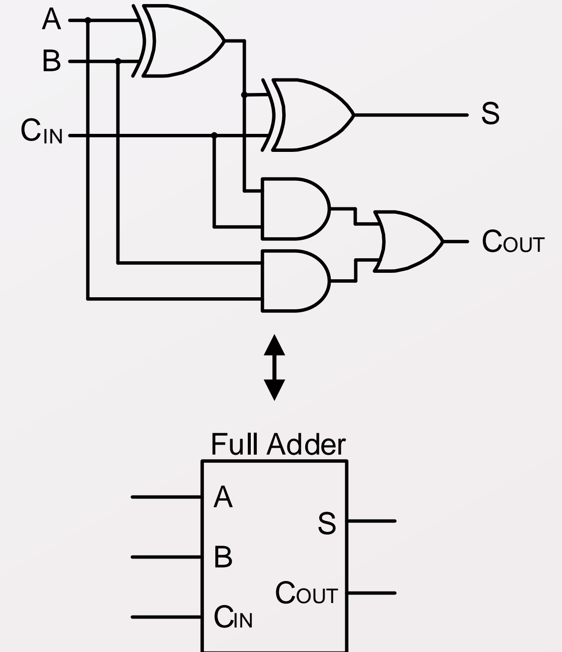
Structural Verilog
positional port mapping

```
module FA(input A,B,Cin,
                  output S,Cout);
   assign S = A ^ B ^ Cin
   assign Count = ((A ^ B) & Cin) | (A & B);
Endmodule
```

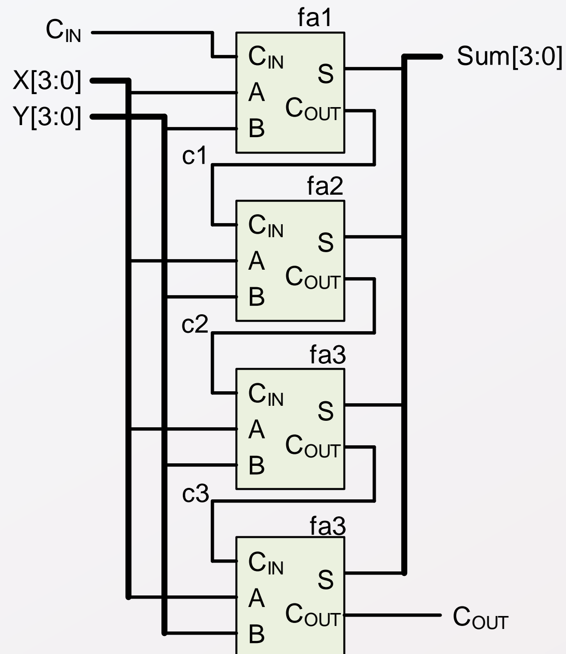
"Semi-behavioral" Verilog

```
module FA(input A,B,Cin, Cout, S);
   assign {Cout, S} = A + B + Cin;
endmodule
```

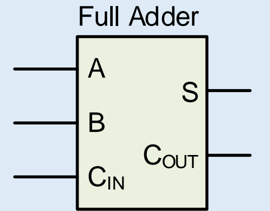
Behavioral Verilog



Design of a 4-bit adder: Structural




```
module full_adder(
  input A,B,Cin,
  output S,Cout
  );

  assign S = A ^ B ^ Cin
  assign Count = ((A ^ B) & Cin) | (A & B);

endmodule
```




```
module adder_4bit(
    input [3:0] X,Y,
    input Cin,
    output [3:0] Sum,
    output Cout
    );

   wire c1,c2,c3;

   full_adder fa1(.A(X[0]),.B(Y[0]),.Cin(Cin),.S(Sum[0]),.Cout(c1));
   full_adder fa2(.A(X[1]),.B(Y[1]),.Cin(c1),.S(Sum[1]),.Cout(c2));
   full_adder fa3(.A(X[2]),.B(Y[2]),.Cin(c2),.S(Sum[2]),.Cout(c3));
   full_adder fa4(.A(X[3]),.B(Y[3]),.Cin(c3),.S(Sum[3]),.Cout(Cout));

endmodule
```

Design of a 4-bit adder: Behavioral

```
module adder_4bit(input [3:0] A,B, output [3:0] Sum);
    assign Sum = A + B;
endmodule
```

```
module adder_4bit(input [3:0] A,B, output [4:0] Sum);
    assign Sum = A + B;
endmodule
```

Adding two n-bit numbers can yield n+1 bit result

```
module adder_4bit(input Cin, [3:0] A,B, output [4:0] Sum);
    assign Sum = A + B + Cin;
endmodule
```

May want carry-in

```
module adder_4bit(input Cin, [3:0] A,B, output Cout, [3:0] Sum);
    assign {cout, Sum} = A + B + Cin;
endmodule
```

Sum[4] is also carry out

Use Comments! // on any line, or /*   */ for a block

```
/*
Seven Segment Controller
EE214 Project 3
Created 11/19/19
Johnny B. Goode
*/

Module 7segcont(
    input [15:0] bcd_in   // Four packed BCD digits
    input clk             // Requires 100MHz input clock
    output [6:0] seg_out  // 7seg segment drivers
    output [3:0] an_out   // 7seg digit enables
    );

etc.
```

Verilog Code Examples

Verilog contains a few built-in logic functions including and (&), or (|) and not (!). You can use these to create more complex functions.

Verilog Multiplexor Examples

```
module mux4(
   input [3:0] I,
   input [1:0] Sel,
   output Y
);
```

“Conventional”

```
module mux4(
   input [3:0] I,[1:0] Sel,
   output Y);
```

```
module mux4(input [3:0] I,[1:0] Sel, output Y);
```

More compact

Verilog Multiplexor Examples

```
module mux4(
   input [3:0] I,
   input [1:0] Sel,
   output Y
);
```

```
assign Y= !Sel[1]&!Sel[0]&I[0] | !Sel[1]&Sel[0]&I[1] | Sel[1]&!Sel[1]&I[2] | Sel[0]&Sel[1]&I[3];
endmodule
```

Verilog Multiplexor Examples

```
module mux4(
   input [3:0] I,
   input [1:0] Sel,
   output Y
);
```

```
assign Y= !Sel[1]&!Sel[0]&I[0] | !Sel[1]&Sel[0]&I[1] | Sel[1]&!Sel[1]&I[2] | Sel[0]&Sel[1]&I[3];
endmodule
```

```
assign Y=( Sel==2’d0) ? I[0] : ((Sel==2’d1) ? I[1] : ((Sel==2’d2) ? I[2] : I[3]));
endmodule
```

```
assign Y=( Sel==2’d0) ? I[0] :
         ((Sel==2’d1) ? I[1] :
         ((Sel==2’d2) ? I[2] : I[3]));
endmodule
```

Verilog Multiplexor Examples

```
module mux4(
   input [3:0] I,
   input [1:0] Sel,
   output Y
);
```

```
assign Y= !Sel[1]&!Sel[0]&I[0] | !Sel[1]&Sel[0]&I[1] | Sel[1]&!Sel[1]&I[2] | Sel[0]&Sel[1]&I[3];
endmodule
```

```
assign Y=( Sel==2’d0) ? I[0] : ((Sel==2’d1) ? I[1] : ((Sel==2’d2) ? I[2] : I[3]));
endmodule
```

```
wire n1, n0, a3, a2, a1, a0, nSel1, nSel0;

not inv1(nSel1, Sel[1]);
not inv0(nSel0, Sel[0]);
and ag0(a0, nSel1, nSel0, I[0]);
and ag1(a1, nSel1, Sel[0], I[1]);
and ag2(a2, Sel[1], nSel0, I[2]);
and ag3(a3, Sel[1], Sel[0], I[3]);
or og0(Y, a0, a1, a2, a3);
endmodule
```

Verilog Multiplexor Examples

```
module mux4(
   input [3:0] I,
   input [1:0] Sel,
   output Y
);
```

```
assign Y= !Sel[1]&!Sel[0]&I[0] | !Sel[1]&Sel[0]&I[1] | Sel[1]&!Sel[1]&I[2] | Sel[0]&Sel[1]&I[3];
endmodule
```

```
assign Y=( Sel==2’d0) ? I[0] : ((Sel==2’d1) ? I[1] : ((Sel==2’d2) ? I[2] : I[3]));
endmodule
```

```
wire n1, n0, a3, a2, a1, a0, nSel1, nSel0;

not inv1(nSel1, Sel[1]);
not inv0(nSel0, Sel[0]);
and ag0(a0, nSel1, nSel0, I[0]);
and ag1(a1, nSel1, Sel[0], I[1]);
and ag2(a2, Sel[1], nSel0, I[2]);
and ag3(a3, Sel[1], Sel[0], I[3]);
or og0(Y, a0, a1, a2, a3);
endmodule
```

```
assign Y= I[Sel];
endmodule
```

Verilog Multiplexor Examples

```
module mux4(input [3:0] I,[1:0] Sel, output reg Y);
```

Using a Case statement

```
always @ (Sel,I)
   case (Sel)
      2'd0: Y = I[0];
      2'd1: Y = I[1];
      2'd2: Y = I[2];
      2'd3: Y = I[3];
  endcase
  endmodule
```

```
always @ (Sel,I)
   case (Sel)
      2'd0: Y = I[0];
      2'd1: Y = I[1];
      2'd2: Y = I[2];
      default: Y = I[3];
  endcase
  endmodule
```

```
always @ (Sel,I)
   case (Sel)
      2'd0: Y = I[0];
      2'd1: Y = I[1];
      2'd2: Y = I[2];
  endcase
  endmodule
```

! Memory

```
always @ (Sel,I)
case (Sel)
  Sel: Y = I[Sel];
endcase
endmodule
```

Verilog Multiplexor Examples

```
module mux4(input [3:0] I,[1:0] Sel, output reg Y);
```

Using an If statement

```
always @ (Sel, I)
  if       (Sel == 2’d0) Y = I[0];
  else if (Sel == 2’d1) Y = I[1];
  else if (Sel == 2’d2) Y = I[2];
  else if (Sel == 2’d3) Y = I[3];
endmodule
```

```
always @ (Sel, I)
  if       (Sel == 2’d0) Y = I[0];
  else if (Sel == 2’d1) Y = I[1];
  else if (Sel == 2’d2) Y = I[2];
  else                  Y = I[3];
endmodule
```

```
always @ (Sel, I) begin
  if (Sel == 2’d0) Y = I[0];
  if (Sel == 2’d1) Y = I[1];
  if (Sel == 2’d2) Y = I[2];
  if (Sel == 2’d3) Y = I[3];
 end
 endmodule
```

Bus Multiplexor

```
module mux4(input [7:0] I3, I2, I1, I0, [1:0] Sel, output [7:0] Y);
  assign Y= (Sel==2'd0) ? I0 : ((Sel==2'd1) ? I1 : ((Sel==2'd2) ? I2 : I3));
endmodule
```

Parameters can hold values used to define circuits during simulation and synthesis. For example, the mux data path width could be defined by a parameter so it could more easily be changed:

```
module mux4 #(parameter W=1) (
    input [W-1:0] I0,I1,I2,I3,
    input [1:0] Sel,
    output [W-1:0] Y);

assign Y = (Sel==2'd0) ? I0 : ((Sel==2'd1) ? I1 : ((Sel==2'd2) ? I2 : I3));

endmodule
```

Verilog Decoder Examples

```
module dec_2_4(input [1:0] I, output [3:0] Y);
```

```
assign Y= (I==2’d0) ? 4’b0001 :
         ((I==2’d1) ? 4’b0010 :
         ((I==2’d2) ? 4’b0100 : 4’b1000));
endmodule
```

Using Case and If statements

```
module dec_2_4(input [1:0] I, output reg [3:0] Y);
```

```
always @ (I)
  case (I)
     2’d0: Y = 4’b0001;
     2’d1: Y = 4’b0010;
     2’d2: Y = 4’b0100;
     2’d3: Y = 4’b1000;
     default: Y = 4’b00001;
   endcase
endmodule
```

```
always @ (I)
  if       (I == 2’d0) Y = 4’b0001;
  else if (I == 2’d1) Y = 4’b0010;
  else if (I == 2’d2) Y = 4’b0100;
  else                 Y = 4’b1000;
endmodule
```

Verilog Random Truth Table

```
module TT_3_2(
   input  [2:0] I,
   output [1:0] Y
);
```

Truth Table

A	B	C	X	Y
0	0	0	1	0
0	0	1	1	1
0	1	0	0	1
0	1	1	0	1
1	0	0	0	0
1	0	1	1	0
1	1	0	0	1
1	1	1	0	1

Using a Case statement

```
reg [1:0] Y;

always @ (I)
  case (I)
      3’d0: Y = 4’b10;
      3’d1: Y = 4’b11;
      3’d2: Y = 4’b01;
      3’d3: Y = 4’b01;
      3’d4: Y = 4’b00;
      3’d5: Y = 4’b10;
      3’d6: Y = 4’b01;
      3’d7: Y = 4’b01;
      default: Y = 4’b00;
   endcase
endmodule
```

Using Assign statement

```
assign X= (I==3’d0 || I==3’d1 || I==3’d5 ) ? 1’b1 : 1’b0;
endmodule
```

ALU Example

```
module alu(
   input [7:0] A,B,
   input [3:0] Sel,
   output reg [7:0] Y
);

always @ (Sel, A, B)
 case (Sel)
     3’d0: Y = A + B;
     3’d1: Y = A - B;
     3’d2: Y = A + 1;
     3’d3: Y = A - 1;
     3’d4: Y = A | B;
     3’d5: Y = A & B;
     3’d6: Y = A ^ B;
     3’d7: Y = ~A;
     default: Y = 7’d0;
   endcase
endmodule
```

Flip-Flops and Latch

```
module dlatch (input  gate, rst, D, output reg Q);
```

```
always @ (gate, rst, D)
   if (rst == 1) Q <= 1'b0;
   else if (gate) Q <= D;

endmodule
```

```
module dff (input  clk, rst, D, output reg Q);
```

```
always @ (posedge(clk), posedge(rst))
if (rst == 1) Q <= 1'b0;
   else Q <= D;
endmodule
```

D Flip-flop
Asynch reset

```
always @ (posedge(clk))
if (rst == 1) Q <= 1'b0;
   else Q <= D;
endmodule
```

D Flip-flop
Synch reset

Counters

Counter-based clock divider

```
module clkdivider (
     input clk, rst,
     output reg divclk
     );

localparam terminalcount = (10000 - 1);
reg [15:0] count;

always @ (posedge(clk), posedge(rst))
begin
     if (rst) begin
       count <= 0;
       divclk <= 0;
     end
     else if (count == terminalcount) begin
       count <= 0;
       divclk = !divclk;
     end
     else count <= count + 1;
end
endmodule
```

Binary Counter

```
module count(
  input cen, clk, rst,
  output reg [7:0] cnt
);

always @ (posedge(clk), posedge(rst))
   if (rst) cnt <= 0;
   else if (cen) cnt <= cnt+1;
endmodule
```

Counters

Counter-based clock divider

```
module clkdivider (
     input clk, rst,
     output reg divclk
     );

localparam terminalcount = (10000 - 1);
reg [15:0] count;

always @ (posedge(clk), posedge(rst))
begin
     if (rst) begin
       count <= 0;
       divclk <= 0;
     end
     else if (count == terminalcount) begin
       count <= 0;
       divclk = !divclk;
     end
     else count <= count + 1;
end
endmodule
```

"localparam" defines a local constant – helpful for documentation and making changes in centralized location.

Binary Counter

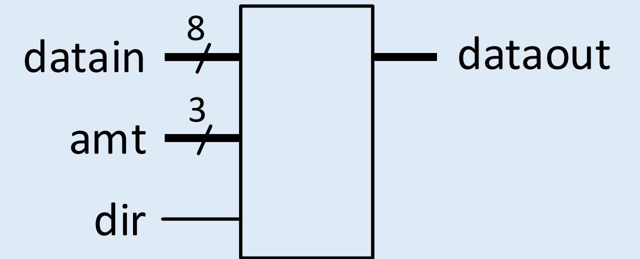
```
module count(
  input cen, clk, rst,
  output reg [7:0] cnt
);

always @ (posedge(clk), posedge(rst))
   if (rst) cnt <= 0;
   else if (cen) cnt <= cnt+1;
endmodule
```

Shifters

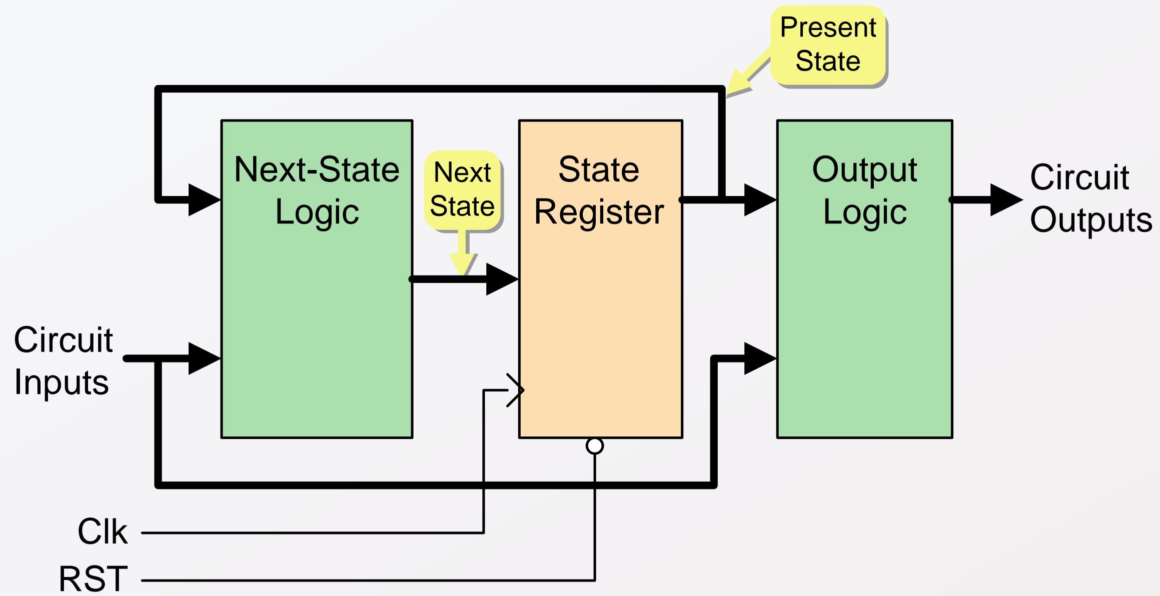
```
module shiftleft(
    input [7:0] Xin,
    input [2:0] Amt,
    output[7:0] Xout
    );

assign Xout = Xin << Amt;

endmodule
```

```
module shiftright(
    input [7:0] Xin,
    input [2:0] Amt,
    output[7:0] Xout
    );

assign Xout = Xin >> Amt;

endmodule
```

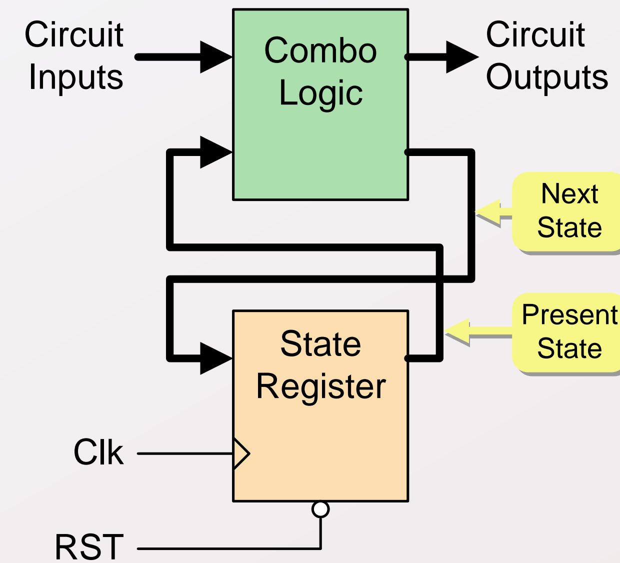
```
module shifter(
    input [7:0] datain,
    input [2:0] amt,
    input dir,
    output[7:0] dataout
    );

assign dataout = (dir==1) ? datain << amt : datain >> amt;

endmodule
```



State Machine Models

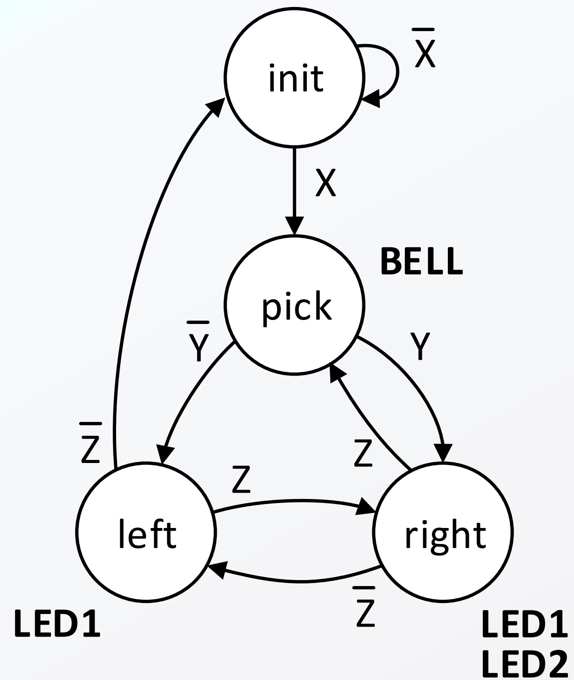


Mealy Model State Machine



Combined Model State Machine

State Machines

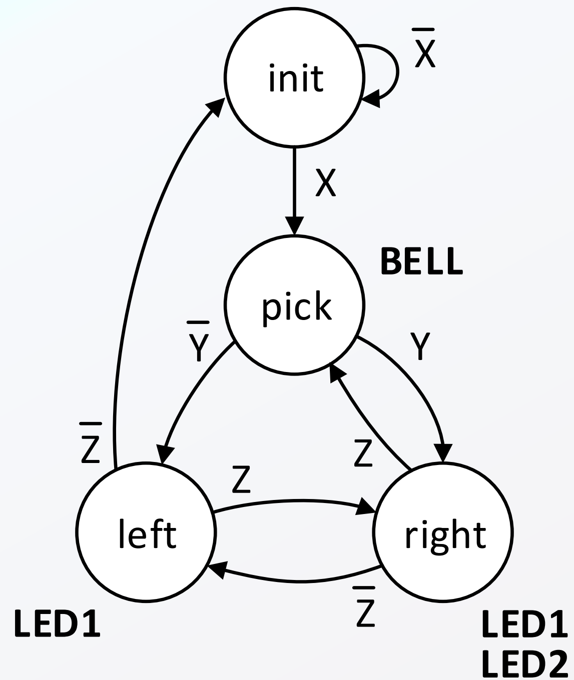


This is a common (and good!) way to code state machines:

1) Choose state codes, define state localparams, and define ps and ns busses;
2) Use two always blocks – one for next-state logic (combinational), and one for the state register; 3) assign outputs.

```
module fsm1 (input x, y, z, clk, rst,
             output bell, led1, led2);

localparam init = 2'b00;   // define state identifiers
localparam pick = 2'b01;   // These are optional -
localparam left = 2'b10;   // you could just use state
localparam right = 2'b11;  // codes (binary numbers)

reg [1:0] ps, ns            // define ps, ns bus signals

case (ps)
  init:  if (x == 1) then ns = pick;  else ns = init;
  pick:  if (y == 1) then ns = right; else ns = left;
  left:  if (z == 1) then ns = right; else ns = init;
  right: if (z == 1) then ns = pick;  else ns = left;
  default: ns = init;
endcase

always @(posedge(clk),posedge(rst))
  if (rst) ps <=0; else ps <= ns;
end

bell = (ps == pick);
assign led1 = (ps == left || ps == right);
assign led2 = (ps == right);
end module
```

State Machines



Another method is to include all combinational assignments in the case statement.

```
case (ps)
  init: begin
    if (x == 1) then ns = pick;
      else ns = init;
    assign bell = 0;
    assign led1 = 0;
    assign led2 = 0;
    end
  pick: begin
    if (y == 1) then ns = right;
      else ns = left;
    assign bell = 1;
    assign led1 = 0;
    assign led2 = 0;
    end
  left: begin
    if (z == 1) then ns = right;
      else ns = init;
    assign bell = 0;
    assign led1 = 1;
    assign led2 = 0;
    end
  right: begin
    if (z == 1) then ns = pick;
      else ns = left;
    assign bell = 0;
    assign led1 = 1;
    assign led2 = 1;
    end
endcase
```

Simulating Verilog Source Files

Testbenches

Testbench – the basics

A testbench is a separate Verilog module written specifically to test other Verilog modules. It uses the same statements, structures and syntax as any Verilog source file.

Testbenches do not define circuits and are not synthesized. They can use additional unsynthesizable statements (like initial blocks and for loops) to create stimulus inputs.

- Initial blocks are like always blocks (i.e., the statements in the block are sequential), but they are only executed once at the start of the simulation.
- For loops are useful for creating patterns, like a counting sequence that can drive all possible input combinations, or repeating signals, like a clock signal.

A testbench module has no inputs or outputs – it doesn't need them. Instead, it will connect to the inputs and outputs of the module being tested.

A testbench is the “top” module, and it includes/instantiates the circuit to be tested (just like structural Verilog)

Testbench – the basics

“Executing” a testbench will cause the simulator window to open, and the simulator will start running.

Assignment statements in the testbench define input signal values, and time statements define how much time passes between input signal assignments.

The simulator applies the defined input values to the circuit under test and determines how circuit nets should change in response.

Circuit inputs and outputs are shown in a waveform view window. The simulation stops when there are no more defined input signal changes.

Input signals are treated as simulator variables that are assigned in an “initial” procedural block. All inputs must be therefore be defined as type reg at the top of the testbench source file.

All outputs will automatically be added to the waveform viewer; however, only the least-significant signal in a bus will be added automatically. To see the entire bus, it must be declared as type wire at the top of the testbench source file.

Testbench Example 4:1 Mux

Source Code

```
module mux4_1(
  input [3:0] I4,[1:0] S4,
  output Y4);

  assign Y4 = I4[S4];

endmodule
```

Testbench Example 4:1 Mux

Source Code

```
module mux4_1(
  input [3:0] I4,[1:0] S4,
  output Y4);

  assign Y4 = I4[S4];

endmodule
```

Testbench

```
module mux4_1_tb();

reg [3:0] IT;
reg [1:0] ST;


mux4_1 cut(.I4(IT),.S4(ST),.Y4(YT));

integer k;

initial begin

  IT = 4'b0110;

  for (k=0; k<4; k=k+1) begin
    ST = k;
    #10;
    end
  end
endmodule
```

Testbench Example 4:1 Mux

Source Code

```
module mux4_1(
  input [3:0] I4,[1:0] S4,
  output Y4);

  assign Y4 = I4[S4];

endmodule
```

A testbench starts with an empty module statement.

Testbench

```
module mux4_1_tb();

reg [3:0] IT;
reg [1:0] ST;


mux4_1 cut(.I4(IT),.S4(ST),.Y4(YT));

integer k;

initial begin

  IT = 4'b0110;

  for (k=0; k<4; k=k+1) begin
    ST = k;
    #10;
    end
  end
endmodule
```

Testbench Example 4:1 Mux

Source Code

```
module mux4_1(
   input [3:0] I4,[1:0] S4,
   output Y4);

   assign Y4 = I4[S4];

endmodule
```

Simulation inputs must be declared as **regs**, so they can be assigned values in the “initial” procedural block that follows

Individual output signals will automatically be shown in the simulator output window; busses must be declared as wires following the reg declarations, or only the least significant bus signal will be displayed

Testbench

```
module mux4_1_tb();
reg [3:0] IT;
reg [1:0] ST;


mux4_1 cut(.I4(IT),.S4(ST),.Y4(YT));

integer k;

initial begin

   IT = 4'b0110;

   for (k=0; k<4; k=k+1) begin
      ST = k;
      #10;
      end
   end
endmodule
```

Testbench Example 4:1 Mux

Source Code

```
module mux4_1(
   input [3:0] I4,[1:0] S4,
   output Y4);

   assign Y4 = I4[S4];

endmodule
```

The “circuit under test” (cut) is instantiated using named port association.

Testbench

```
module mux4_1_tb();

reg [3:0] IT;
reg [1:0] ST;


mux4_1 cut(.I4(IT),.S4(ST),.Y4(YT));

integer k;

initial begin

   IT = 4'b0110;

   for (k=0; k<4; k=k+1) begin
     ST = k;
     #10;
     end
   end
endmodule
```

Testbench Example 4:1 Mux

Source Code

```
module mux4_1(
   input [3:0] I4,[1:0] S4,
   output Y4);

   assign Y4 = I4[S4];

endmodule
```

Simulator inputs are defined in an initial procedural block using blocking (=) assignments.

Testbench

```
module mux4_1_tb();

reg [3:0] IT;
reg [1:0] ST;


mux4_1 cut(.I4(IT),.S4(ST),.Y4(YT));

integer k;

initial begin

   IT = 4'b0110;

   for (k=0; k<4; k=k+1) begin
     ST = k;
     #10;
     end
   end
endmodule
```

Testbench Example 4:1 Mux

Source Code

```
module mux4_1(
   input [3:0] I4,[1:0] S4,
   output Y4);

   assign Y4 = I4[S4];

endmodule
```

Here, the mux inputs are defined.

In a more proper/complete simulation, mux inputs would be driven to a 1 and a 0 so that proper response to ***all*** combinations of inputs could be verified.

Testbench

```
module mux4_1_tb();

reg [3:0] IT;
reg [1:0] ST;


mux4_1 cut(.I4(IT),.S4(ST),.Y4(YT));

integer k;

initial begin

   IT = 4'b0110;

   for (k=0; k<4; k=k+1) begin
      ST = k;
      #10;
      end
   end
endmodule
```

Testbench Example 4:1 Mux

Source Code

```
module mux4_1(
   input [3:0] I4,[1:0] S4,
   output Y4);

   assign Y4 = I4[S4];

endmodule
```

A for loop is often used to create sequences of inputs.

Testbench

```
module mux4_1_tb();

reg [3:0] IT;
reg [1:0] ST;


mux4_1 cut(.I4(IT),.S4(ST),.Y4(YT));

integer k;

initial begin

   IT = 4'b0110;

   for (k=0; k<4; k=k+1) begin
      ST = k;
      #10;
      end
   end
endmodule
```

Testbench Example 4:1 Mux

Source Code

```
module mux4_1(
   input [3:0] I4,[1:0] S4,
   output Y4);

   assign Y4 = I4[S4];

endmodule
```

The same sequence created by the for loop could be created manually… but with much greater effort.

Testbench

```
module mux4_1_tb();

reg [3:0] IT;
reg [1:0] ST;


mux4_1 cut(.I4(IT),.S4(ST),.Y4(YT));

integer k;

initial begin

   IT = 4'b0110;
   ST = 2'b00;
   #10;
   ST = 2'b01;
   #10;
     .
   end
endmodule
```

Testbench Example 4:1 Mux

Source Code

```
module mux4_1(
  input [3:0] I4,[1:0] S4,
  output Y4);

  assign Y4 = I4[S4];

endmodule
```

Testbench

```
module mux4_1_tb();

reg [3:0] IT;
reg [1:0] ST;


mux4_1 cut(.I4(IT),.S4(ST),.Y4(YT));

integer k;

initial begin

  IT = 4'b0110;

  for (k=0; k<4; k=k+1) begin
    ST = k;
    #10;
    end
  end
endmodule
```

Testbench Example 8:1 Mux Source (built from two 4:1 muxes)

Source Code

```
module mux4_1(
  input [3:0] I4,[1:0] S4,
  output Y4);

  assign Y4 = I4[S4];

endmodule

module mux8_1(
  input [7:0] IN8, [2:0] S8,
  output Y8);

wire Y4L, Y4H;

mux4_1 Hi (.I4(IN8[7:4]),.S4(S8[1:0]),.Y4(Y4H));
mux4_1 Lo (.I4(IN8[3:0]),.S4(S8[1:0]),.Y4(Y4L));

assign Y8 = S8[2] == 0 ? Y4L : Y4H;

endmodule
```

Testbench

```
module mux8_1_tb();

reg [7:0] IT;
reg [2:0] ST;


mux8_1 cut(.IN8(IT),.S8(ST),.Y8(YT));

integer k;

initial begin

  IT = 8’b10010110;

  for (k=0; k<8; k=k+1) begin
    ST = k;
    #10;
    end
  end
endmodule
```

Testbench Example 8:1 Mux Source (built from two 4:1 muxes)

Source Code

```
module mux4_1(
  input [3:0] I4,[1:0] S4,
  output Y4);

  assign Y4 = I4[S4];

endmodule


module mux8_1(
  input [7:0] IN8, [2:0] S8,
  output Y8);

wire Y4L, Y4H;

mux4_1 Hi (.I4(IN8[7:4]),.S4(S8[1:0]),.Y4(Y4H));
mux4_1 Lo (.I4(IN8[3:0]),.S4(S8[1:0]),.Y4(Y4L));

assign Y8 = S8[2] == 0 ? Y4L : Y4H;

endmodule
```

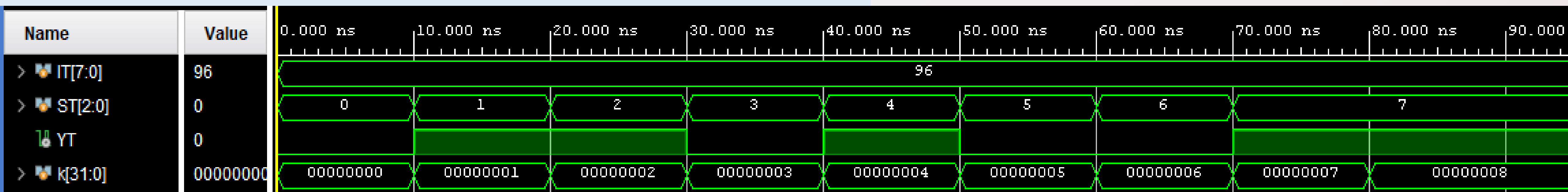
Testbench

```
module mux8_1_tb();

reg [7:0] IT;
reg [2:0] ST;


mux8_1 cut(.IN8(IT),.S8(ST),.Y8(YT));

integer k;

initial begin

  IT = 8’b10010110;

  for (k=0; k<8; k=k+1) begin
    ST = k;
    #10;
    end
  end
endmodule
```

Testbench Example 8:1 Mux Source (built from two 4:1 muxes)

```
module mux4_1(
  input [3:0] I4,[1:0] S4,
  output Y4);

  assign Y4 = I4[S4];

endmodule

module mux8_1(
  input [7:0] IN8, [2:0] S8,
  output Y8);

wire Y4L, Y4H;

mux4_1 Hi (.I4(IN8[7:4]),.S4(S8[1:0]),.Y4(Y4H));
mux4_1 Lo (.I4(IN8[3:0]),.S4(S8[1:0]),.Y4(Y4L));

assign Y8 = S8[2] == 0 ? Y4L : Y4H;

endmodule
```

```
module mux8_1_tb();

reg [7:0] IT;
reg [2:0] ST;


mux8_1 cut(.IN8(IT),.S8(ST),.Y8(YT));

integer k;

initial begin

  IT = 8'b10010110;

  for (k=0; k<8; k=k+1) begin
    ST = k;
    #10;
    end
  end
endmodule
```

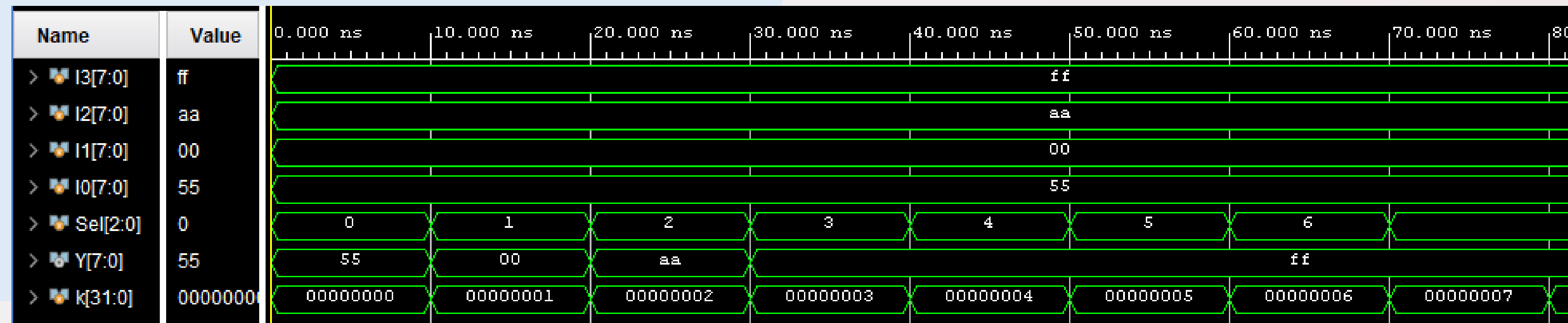


Testbench Example 4:1 Bus Mux

```
module mux4_8 (input [7:0] I3, I2, I1, I0, [2:0] Sel, output [7:0] Y);
  assign Y= (Sel==2'd0) ? I0 : ((Sel==2'd1) ? I1 : ((Sel==2'd2) ? I2 : I3));
endmodule
```

```
module mux4_8_tb();

reg [7:0] I3, I2, I1, I0;
reg [2:0] Sel;
wire [7:0] Y;
mux4_8 cut(.I3(I3),.I2(I2),.I1(I1),.I0(I0),.Sel(Sel),.Y(Y));
integer k;
initial begin
I3 = 8'hFF;
I2 = 8'hAA;
I1 = 8'h00;
I0 = 8'h55;
for (k=0; k<8; k=k+1) begin
   Sel = k; #10;
   end
end
endmodule
```

Note the output bus (Y) is declared as a wire/bus at the top of the testbench file; this directs the simulator to show the entire bus in the waveform output window



Adding a clock

A clock signal can be created using a one-line “always block”

A 4-bit decimal counter and test bench illustrate.

```
module dec_cnt(
  input cen, clk, rst,
  output reg [7:0] cnt
);

always @ (posedge(clk), posedge(rst))
   if (rst) cnt <= 0;
   else begin
     if (cen) cnt <= cnt+1;
     if (cnt == 4'b1001) cnt <= 4'b0000;
   end
endmodule
```

```
module dec_cnt_tb();

reg clk, cen, rst;
wire [7:0] cnt;

dec_cnt c(.cen(cen),.clk(clk),.rst(rst),.cnt(cnt));

initial begin

clk = 0;
cen = 1;
rst = 1;
#10;
rst = 0;
#150;

end

always #10 clk = !clk;

endmodule
```

Projects and Source Files

Verilog tools organize the workspace using “Projects”. Projects contain all the source files needed for a given design. The project is simply a directory, and all new sources are saved in the current project.

For most projects, you will create one or more Verilog source files and one or more Verilog testbench files. You will also need a ***constraints*** file to identify external pin connections (you can use the same constraints file for all projects because all pin locations on your circuit board are fixed).

All source files you create are in …/*pname*/*pname*.srcs/, where *pname* is your chosen project name (the entire project path is shown at the top of the Vivado window). All other directories and files in the project are created by the tool – you should not modify any of them.

You can add/copy any source file from any other project into your current project from within the Vivado tool.

End.