
Computer Aided Digital Systems Design - EE 4743/6743

Sherif Abdelwahed

Combinational Verilog

**Department of Electrical and Computer Engineering
Mississippi State University**

Hardware Description Languages

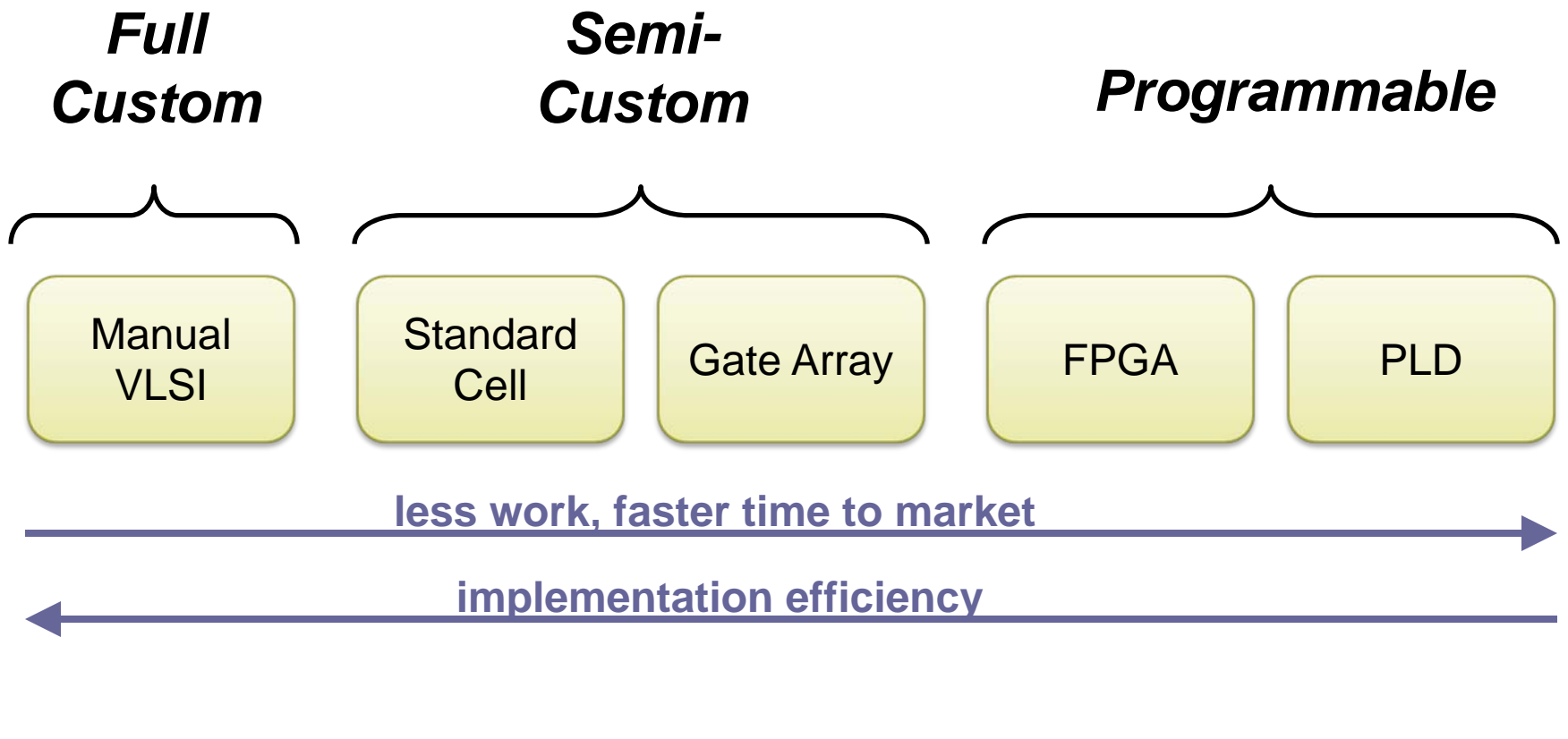
- Hardware description languages (HDLs)
 - Are computer-based hardware programming languages
 - Have high-level language constructs to describe the functionality and connectivity of the circuit
 - Allow modeling and simulating the functional behavior and timing of digital hardware
 - Synthesis tools take an HDL description and generate a technology-specific netlist
 - Can describe a design at different levels of abstraction
 - Behavioral, RTL (Register Transfer Level), Gate-level, Switch
 - Two main HDLs used by industry
 - Verilog HDL (C-based, industry-driven)
 - VHSIC HDL or VHDL (Ada-based, defense/industry/university-driven).
-

Why Use an HDL?

- Simplified and faster design process
 - Abstract large hardware designs
 - Describe what you need the hardware to do
 - Tools then design the hardware for you
 - Explore larger solution space
 - Smaller, faster, lower power
 - Examine design tradeoffs: ex., throughput vs. latency
 - Lessen the time spent debugging the design
 - Design errors still possible, but in fewer places
 - Generally easier to find and fix
 - Other features:
 - Highly portable: can reuse design to target different technologies
 - Self-documenting (when commented well).
-

Hardware Implementations

HDLs can be compiled to semi-custom and programmable hardware implementations



What is Verilog?

- It is a Hardware Description Language
 - A language used for simulation and synthesis of digital logic
 - Verilog HDL provides:
 - Mixed level modeling:
 - Behavioral: algorithmic,
 - Dataflow: register transfer,
 - Structural: gates, switches, modules
 - Built-in primitives, logic functions, and data types
 - A Verilog description of a digital system can be transformed into a gate level implementation.
 - This process is known as synthesis
 - Official Language Document: “Verilog Hardware Description Language Reference Manual”, IEEE Std 1364-1995, IEEE.
-

Basic Unit: Module

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

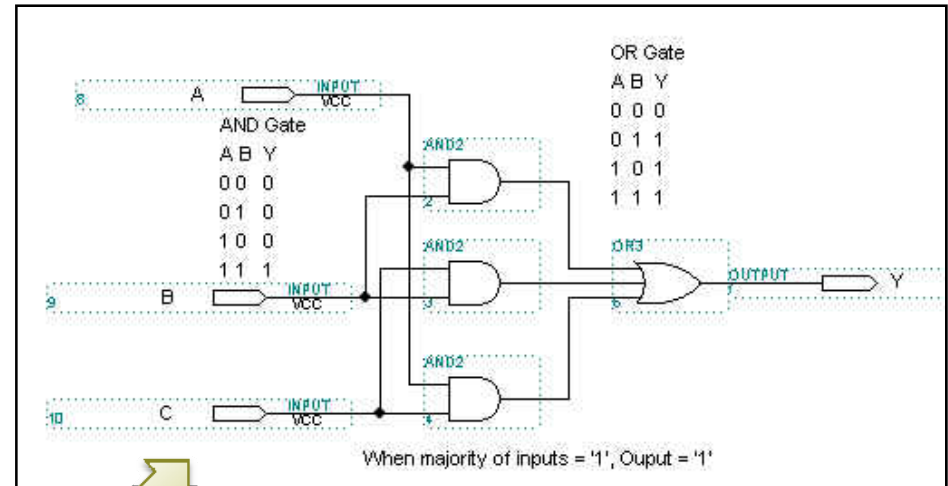
```
module module_name (port_list);  
declarations:  
    port declaration (input, output, inout, ...)  
    data type declaration (reg, wire, parameter, ...)  
    task and function declaration  
statements:  
    initial block  
    always block  
    module instantiation  
    gate instantiation  
    continuous assignment  
endmodule
```

The diagram illustrates the mapping of Verilog statements to different modeling styles. Three boxes on the left, each containing a list of statements, have arrows pointing to labels on the right:

- The first box contains "initial block" and "always block", with an arrow pointing to "Behavioral".
- The second box contains "module instantiation" and "gate instantiation", with an arrow pointing to "Structural".
- The third box contains "continuous assignment", with an arrow pointing to "RTL (Dataflow)".

Verilog Example

Implementation



module name *ports names of module*

```
module majconc (a,b,c,y);  
  input a,b,c;  
  output y;  
  
  assign y = (a & b) | (a & c) | (b &c) ;  
  
endmodule
```

port types

*statement
(continuous assignment)*

Code

Declaring a Module

- Name the module
 - Can't use Verilog keywords as module, port or signal names (see the Verilog standard or Ciletti book, appendix A for list of keywords).
 - Choose a *descriptive* module name
 - List the port names (module interface)
 - Choose *descriptive* port names
 - Declare the type and size of ports
 - Declare any internal signals
 - Write the internals of the module (functionality)
-

Declaring Ports

- A signal is attached to every port
 - Only the ports are accessible from outside the module
 - Declare type of port
 - **input**
 - **output**
 - **inout** (bidirectional)
 - Scalar (single bit) - don't specify a size
 - **input** cin;
 - Vector (multiple bits) - specify size using range
 - Range is MSB to LSB (left to right)
 - Don't have to include zero if you don't want to... (**D[2:1]**)
 - **output** OUT [7:0];
 - **input** IN [0:4];
-

Module Port List

- Multiple ways to declare the ports of a module

```
module Add_half(c_out, sum, a, b);  
    output sum, c_out;  
    input a, b;  
    ...  
endmodule
```

```
module Add_half(output c_out, sum, input a, b);  
    ...  
endmodule
```

Four-Value Logic

- A single bit can have one of FOUR values
 - 0 Numeric 0, logical FALSE
 - 1 Numeric 1, logical TRUE
 - x Unknown or ambiguous value
 - z No value (high impedence)

 - Why **x**?
 - Could be a conflict, could be lack of initialization
 - Why **z**?
 - Nothing driving the signal
 - Tri-states
-

The x and z values

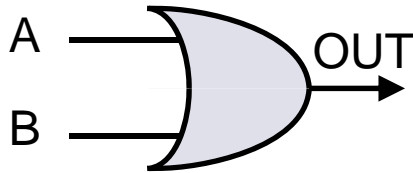
■ In Simulation

- Can detect x or z using special comparison operators
- x is useful to see:
 - Uninitialized signals
 - Conflicting drivers to a wire
 - Undefined behavior

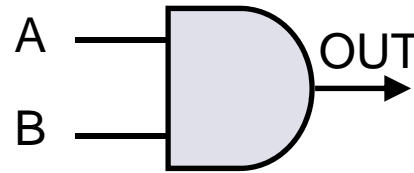
■ In Reality

- Cannot detect x or z
 - No physical 'x' – electrically just isn't 0, 1, or z
 - Except for some uninitialized signals, x is bad!
 - Multiple strong conflicting drivers => short circuit
 - Weak signals => circuit can't operate, unexpected results
 - z means nothing is driving a net (tri-state)
-

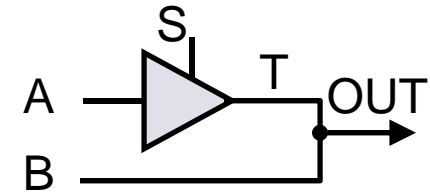
Resolving 4-Value Logic



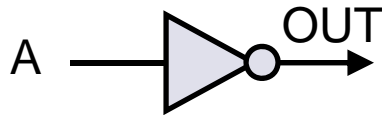
A	B	OUT
0	0	0
0	1	1
1	1	1
0	x	x
0	z	x
1	x	1
1	z	1



A	B	OUT
0	0	0
0	1	0
1	1	1
0	x	0
0	z	0
1	x	x
1	z	x



S	A	T	B	OUT
0	0	z	z	z
0	1	z	x	x
0	x	z	1	1
0	z	z	0	0
1	0	0	1	x
1	0	0	z	0
1	1	1	z	1
1	x	x	z	x
1	z	x	0	x



A	OUT
0	1
1	0
x	x
z	x

Verilog Statements

- We will explore and use a small subset of the language
 - Some Verilog constructs:
 - Signal Assignment: `assign A <= B;`
 - Conditional assignment:
`(conditional_exp) ? (value-if-true) : (value-if-false);`
 - Comparisons: `==` (eq.), `!=` (not eq.), `>` (greater than), `<` (less than)
 - Boolean operations: `&` (and) `|` (or) `~` (not) `^` (xor)
 - Logical operators: `&&` (and), `||` (or), `!` (not)
 - Shift operators: `<<` (shift left), `>>` (shift right)
 - Arithmetic: `+` (addition), `-` (sub.), `*` (mult.), `/` (div.), `%` (mod)
 - Sequential statements: (`case`, `if`, `for`)
 - Concurrent statements: (`when-else`)
-

Constants in Verilog

- Syntax

```
[size][`radix]constant
```

- Radix can be d, b, h, or o (default d)

- Examples

- `assign Y = 10;` // Decimal 10
- `assign Y = `b10;` // Binary 10, decimal 2
- `assign Y = `h10;` // Hex 10, decimal 16
- `assign Y = 8`b0100_0011` // Underline ignored

- Size defaults to at least 32...

- You should specify the size explicitly!
 - Why create 32-bit register if you only need 5 bits?
 - May cause compilation errors on some compilers
-

Verilog Data Types

- Two basic categories
 - **Net variables**
 - Nets establish connectivity
 - Nets must be driven by something
 - We will mainly use the net type `wire`
 - **Register variables**
 - Registers store variables (states)
 - We will mainly use the register type `reg` and `integer`
 - Only dealing with nets in structural Verilog
 - The “register” data type doesn’t necessarily imply an actual register...
 - More about this later
-

Net Types

- **wire**: most common, establishes connections
 - **Default value for all signals**
 - **tri**: indicates will be output of a tri-state
 - Same functionality as “wire”, used for tri-state buses
 - **supply0, supply1**: ground and power connections
 - Can imply this by saying “0” or “1” instead
 - **wand, wor, triand, trior, tri0, tri1, triereg**
 - Nets with special behavior (open-drain, pull-ups, etc.)
 - Not used in this course
-

Variable Datatypes

- **reg** – scalar or vector binary values
 - **integer** – 32 or more bits
 - **time** – time values represented in 64 bits (unsigned)
 - **real** – double real values in 64 or more bits
 - **realtime** - stores time as double real (64-bit +)

 - Assigned value only within a behavioral block
 - **CANNOT USE AS:**
 - Output of primitive gate or instantiated submodule
 - LHS of continuous assignment
 - input or inout port within a module

 - **real** and **realtime** initialize to 0.0, others to x...
 - Just initialize yourself!
-

Examples Of Variables

reg signed	[7:0] A_reg;	// 8-bit signed vector register
reg	Reg_Array[7:0];	// array of eight 1-bit registers
integer	Int_Array[1:100];	// array of 100 integers
real	B, Real_Array[0:5];	// scalar & array of 6 reals
time	Time_Array[1:100];	// array of 100 times
Realtime	D, Real_Time[1:5];	// scalar & array of 5 realtimes

```
initial begin
  A_reg = 8'ha6;           // Assigns all eight bits
  Reg_Array[7] = 1;       // Assigns one bit
  Int_Array[3] = -1;      // Assign integer -1
  B = 1.23e-4;           // Assign real
  Time_Array[20] = $time; // Assigned by system call
  D = 1.25;              // Assign real time
end
```

Verilog Module Styles

- Modules can be specified different ways
 - **Structural**: connect primitives and modules (to be covered later)
 - **RTL**: use continuous assignments to specify combinational logic
 - **Behavioral**: use initial and always blocks to describe the behavior of the circuit, not its implementation
 - A single module can (and often does) use more than one method!
 - We will cover all these different approaches starting with RTL and ending with structural
-

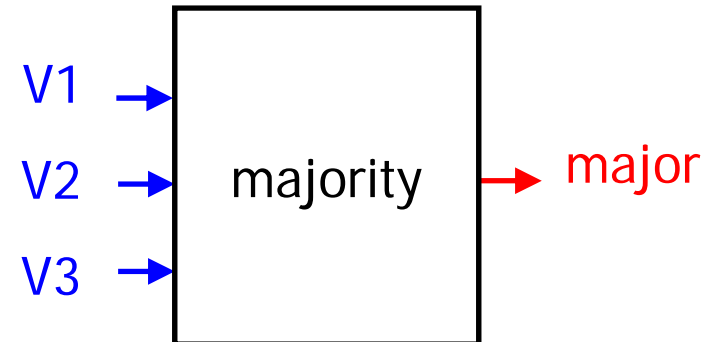
RTL Verilog

- Higher-level of description than structural
 - Don't always need to specify each individual gate
 - Can take advantage of operators
 - More hardware-explicit than behavioral
 - Doesn't look as much like software
 - Frequently easier to understand what's happening
 - Very easy to synthesize
 - Supported by even primitive synthesizers
-

RTL Example: Majority Detector

RTL models specify combinational logic using continuous assignments.

```
module majority (major, V1, V2, V3);  
  
    output major;  
    input V1, V2, V3;  
  
    assign major = V1 & V2 | V2 & V3 | V1 & V3;  
  
endmodule
```



Continuous Assignment

- Implies structural hardware
 `assign <LHS> = <RHS expression>;`
 - Example
 `wire out, a, b;`
 `assign out = a & b;`
 - If RHS result changes, LHS is updated with new value
 - Constantly operating (“continuous”!). It’s **hardware!**
 - Used to model combinational logic and latches
 - Can assign values to:
 - Scalar nets, vector nets, single bits of vector nets, part-selects of vector nets, concatenation of any of the above
 - Examples:
 `assign out[7:4] = a[3:0] | b[7:4];`
 `assign val[3] = c & d;`
 `assign {a, b} = stimulus[15:0];`
-

Behavioral Verilog

- Use procedural blocks: **initial**, **always**
 - These blocks contain series of statements
 - Abstract – works *somewhat* like software
 - Be careful to still remember it's hardware!
 - Parallel operation across blocks
 - All blocks in all modules operate simultaneously
 - Sequential or parallel operation within blocks
 - Depends on the way the block is written
 - Will discuss this more later
 - LHS of assignments are variables (**reg**)
-

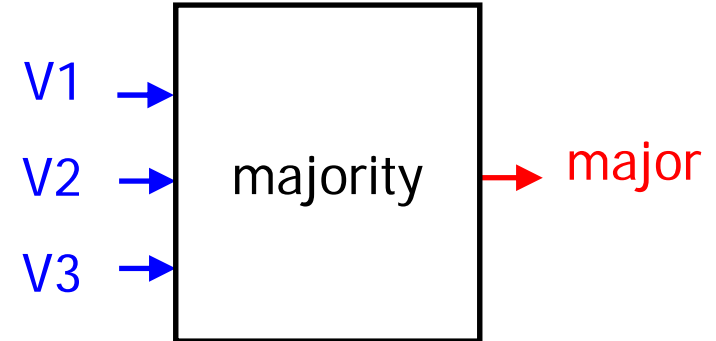
Procedural Assignments

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

Behavioral Example – Majority Detector

Behavior models specify what the logic does, not how to do it.

```
module majority (major, V1, V2, V3) ;  
  
    output reg major ;  
    input V1, V2, V3 ;  
  
    always @(V1, V2, V3) begin  
        if (V1 && V2 || V2 && V3 || V1 && V3)  
            major = 1;  
        else  
            major = 0;  
        end  
  
endmodule
```



Types of Blocks

■ **initial**

- Behavioral block operates ONCE
- Starts at time 0 (beginning of operation)
- Useful for testbenches
- Can sometimes provide initialization of memories/FFs
 - Often better to use “reset” signal
- Inappropriate for combinational logic
- Usually cannot be synthesized

■ **always**

- Behavioral block operates CONTINUOUSLY
 - Can use a *trigger list* to control operation; @(a, b, c)
-

always blocks with sensitivity lists

- Conditionally “execute” inside of **always** block
 - Always block continuously operating
 - If sensitivity list present, continuously checking triggers
 - Any change on trigger (sensitivity) list, triggers block

```
always @(a, b, c) begin
    ...
end
```

- Uses “event control operator” @
 - When net or variable in trigger list changes, **always** block is *triggered*
-

Event or

- Original way to specify trigger list
always @ (X1 or X2 or X3)
 - In Verilog 2001 can use , instead of **or**
always @ (X1, X2, X3)
 - Verilog 2001 also has * for *combinational only*
always @ (*)
 - Shortcut that includes all nets/variables used on RHS in statements in the block
 - Also includes variable used in if statements; if (x)
-

always@ Block

```
module majority (A, B, C, Y);
  input A, B, C;
  output Y;

  reg Y;

  always@(A or B or C)
  begin
    Y = 0; // default output assignment
    if ((A == 1) && (B == 1))
      Y = 1;
    if ((A == 1) && (C == 1))
      Y = 1;
    if ((B == 1) && (C == 1))
      Y = 1;
  end
endmodule
```

denotes a variable

All outputs in an always block must be defined as a variable

Conceptually, the always block runs once whenever a signal in the **sensitivity list** changes value

Statements within the always block are executed sequentially. Order matters!

Comments on `always` Block Model

- The first line in the block `always@(A or B or C)` has the sensitivity list of the process.
 - The sensitivity list should contain any signals that appear on the right hand side of an assignment (inputs) or in any Boolean for a sequential control statement.
- The `if` statement condition must return a Boolean value (`True` or `False`) so that is why the conditional is written as:

`((A == 1) && (B == 1))` → correct

However,

`(A & B)` → false

Verilog Registers

- In digital design, registers represent memory elements (we will study these in the next few lectures)
 - Digital registers need a clock to operate and update their state on certain phase or edge
 - Registers in Verilog should not be confused with hardware registers
 - In Verilog, the term register (`reg`) simply means a variable that can hold a value
 - Verilog registers don't need a clock and don't need to be driven like a net (nets are gate ports joined by wire).
 - Values of registers can be changed anytime in a simulation by assuming a new value to the register
-

Concurrent vs. Sequential

- **Concurrent** statements occur all at once
 - Each concurrent statement will synthesize to a block of logic
 - **Sequential** statements execute linearly
 - Sequential statements can ONLY appear inside of an `always` block
 - An `always` block is considered to be a single concurrent statement
 - Can have multiple `always` blocks in an architecture
 - Usually use `always` blocks to describe complex combinational or sequential logic
-

Concurrent Example

The following is an example of the Carry function implemented in Verilog

```
module majority ( A, B, C, Y );  
  input A, B, C;    // two slashes is a COMMENT in Verilog  
  output Y;  
  
  assign Y = (A & B) | (A & C) | (B & C);  
  
endmodule
```

Continuous assignment is used to assign a value to a variable in a module.

Continuous assignment is concurrent. The order of the assignment does not matter.

Concurrent Example

This is code produces the same result, but uses temporary signals and 4 concurrent statements

```
module majority ( A, B, C, Y );  
  input A, B, C;  
  output Y;  
  
  wire t1, t2, t3;  
  
  assign t1 = A & B;  
  assign t2 = A & C;  
  assign t3 = B & C;  
  assign Y = t1 | t2 | t3;  
  
endmodule
```

A **wire** represents a physical wire in a circuit. The value of wire is driven by continuous assignment only.

Use of *if-else*

```
always@(A or B or C)
  if (((A == 1) && (B == 1)) ||
      ((A == 1) && (C == 1)) ||
      ((B == 1) && (C == 1)))
    Y = 1;
  else
    Y = 0;
```

Use an “else” clause to specify what the output should be if the “if” condition test was not true.

CAREFUL! The Boolean operators (*or*, *and*) do not have any precedence so must use parenthesis to define precedence order

Unassigned Outputs in *always* Blocks

- Assign outputs!
- Any path through the process where an output is NOT assigned a value - that value is undefined

```
always@(A or B or C)
  if (((A == 1) && (B == 1)) ||
      ((A == 1) && (C == 1)) ||
      ((B == 1) && (C == 1)))
    Y = 1;
```

Bad

```
always@(A or B or C)
  begin
    Y = 0;
    if (((A == 1) && (B == 1)) ||
        ((A == 1) && (C == 1)) ||
        ((B == 1) && (C == 1)))
      Y = 1;
  end
```

Good

Blocking and Nonblocking Assignments

Suppose initially $(A,B,C) = (1,1,1)$

```
begin
  A <= 0;
  B <= A;
  C <= B;
end
```



$(A,B,C) = (0,1,1)$
nonblocking
updates only once

```
begin
  A = 0;
  B = A;
  C = B;
end
```



$(A,B,C) = (0,0,0)$
blocking
updates instantly

Blocking vs. Nonblocking

- Don't mix blocking/nonblocking within always begin/end
- Last statement always taken

```
always@(A or B or C)
begin
  Y <= ((A == 1) && (B == 1));
  Y <= ((A == 1) && (C == 1));
  Y <= ((B == 1) && (C == 1));
end
```

Incorrect
Only last line is
implemented

```
always@(A or B or C)
begin
  Y = 0;
  if (((A == 1) && (B == 1)) ||
      ((A == 1) && (C == 1)) ||
      ((B == 1) && (C == 1)))
    Y = 1;
end
```

Correct

Comments on 'Bad' Design

- In the above process, the “else” clause was left out. If the “if” statement condition is false, then the output Y is not assigned a value.
 - The synthesized logic will have a latch placed on the Y output; once Y goes to a '1', it can NEVER return to a '0'
 - This is probably the #1 student mistake in writing always blocks. Avoid this problem by one of the following things:
 - ALL signal outputs of the process should have DEFAULT assignments right at the beginning of the process (this is my preferred method, is easiest).
 - OR, all “if” statements that affect a signal must have “else” clauses that assign the signal a value if the “if” test is false.
-

Verilog Template - Combinational Logic

```
module model_name (list of all inputs and outputs );  
  input list of inputs;  
  output list of outputs;  
    concurrent statement 1;  
    concurrent statement 2;  
    ...  
    concurrent statement N;  
endmodule
```

- All text not in italics are Verilog keywords
- Verilog IS case sensitive
 - “wire” is a keyword
 - “Wire” is not a keyword

Busses Example

```
module priority (y1, y2, y3, y4, y5, y6, y7, dout);
  input y1, y2, y3, y4, y5, y6, y7;
  output [2:0] dout;

  reg dout;

  always@(y1 or y2 or y3 or y4 or y5 or y6 or y7)
    if (y7 == 1) dout = 3'b111;
    else if (y6 == 1) dout = 3'b110;
    else if (y5 == 1) dout = 3'b101;
    else if (y4 == 1) dout = 3'b100;
    else if (y3 == 1) dout = 3'b011;
    else if (y2 == 1) dout = 3'b010;
    else if (y1 == 1) dout = 3'b001;
    else dout = 3'b000;
endmodule
```

dout signal is a 3-bit output bus

- [2:0] is 3-bit bus with dout[2] is MSB, dout[0] is LSB
- [0:2] is also a 3-bit output bus with dout[0] is MSB, dout[2] is LSB

Busses Example

```
module priority (y1, y2, y3, y4, y5, y6, y7, dout);
  input y1, y2, y3, y4, y5, y6, y7;
  output [2:0] dout;

  reg dout;

  always@(y1 or y2 or y3 or y4 or y5 or y6 or y7)
    if (y7 == 1) dout = 3'b111;
    else if (y6 == 1) dout = 3'b110;
    else if (y5 == 1) dout = 3'b101;
    else if (y4 == 1) dout = 3'b100;
    else if (y3 == 1) dout = 3'b011;
    else if (y2 == 1) dout = 3'b010;
    else if (y1 == 1) dout = 3'b001;
    else dout = 3'b000;
endmodule
```

Bus assignment can be done in many ways

- `dout = 7;` assigns all three bits the decimal number seven
- `dout = 3'd7;` same as above
- `dout[2] = 1;` assigns only bit #2
- `dout[1:0] = 2'b10;` assigns only two bits the binary value 10

Priority Circuit Example

```
always@(y1 or y2 or y3 or y4 or y5 or y6 or y7)
  if (y7 == 1) dout = 3'b111;
  else if (y6 == 1) dout = 3'b110;
  else if (y5 == 1) dout = 3'b101;
  else if (y4 == 1) dout = 3'b100;
  else if (y3 == 1) dout = 3'b011;
  else if (y2 == 1) dout = 3'b010;
  else if (y1 == 1) dout = 3'b001;
  else dout = 3'b000;
```

- This priority circuit has 7 inputs; Y7 is highest priority, Y1 is lowest priority.
- Three bit output should indicate the highest priority input that is a 1. That is:
 - If Y6 == 1 and Y4 == 1, then output should be “110”.
 - If no input is asserted, output should be “000”.

Priority Circuit Using *if* Statements

```
always@(y1 or y2 or y3 or y4 or y5 or y6 or y7) begin
  dout = 0;
  if (y1 == 1) dout = 1;
  if (y2 == 1) dout = 2;
  if (y3 == 1) dout = 3;
  if (y4 == 1) dout = 4;
  if (y5 == 1) dout = 5;
  if (y6 == 1) dout = 6;
  if (y7 == 1) dout = 7;
end
```

- By reversing the order of the assignments, we can accomplish the same as the else/if priority chain.
- In an `always` block, the LAST assignment to the output is what counts.

4-to-1 Mux using *Case* Statement

```
module mux4to1_8 (a, b, c, d, sel, dout)

    input [1:0] sel;
    input a, b, c, d;
    output reg dout;

    always@(sel)
        case (sel)
            3 : dout = d;
            2 : dout = c;
            1 : dout = b;
            default : dout = a;
        endcase

endmodule
```

- There can be multiple statements for each case
- Only one statement is needed for each case in this example

Logical Shift Left

```
module Lshift (din, shift_en, dout);
  input  [7:0] din;
  input  shift_en;
  output [7:0] dout;

  reg dout;

  always@(din or shift_en) begin
    dout = din; //default
    if (shift_en == 1) begin
      dout[0] = 0;
      dout[1] = din[0];
      dout[2] = din[1];
      dout[3] = din[2];
      dout[4] = din[3];
      dout[5] = din[4];
      dout[6] = din[5];
      dout[7] = din[6];
    end
  end
endmodule
```

This is one way to do it - surely there is a better way?

Logical Shift Left (better way)

```
module Lshift ( din, shift_en, dout);
  input [7:0] din;
  input shift_en;
  output [7:0] dout;

  reg dout;

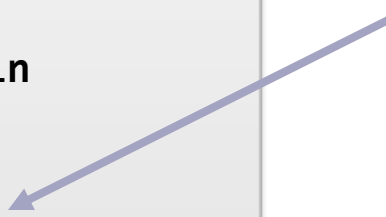
  always@(din or shift_en) begin
    dout = din; //default
    if (shift_en == 1) begin
      dout[0] = 0;
      dout[7:1] = din[6:0];
    end
  end
endmodule
```

- The assignment of a segment of one bus to another bus segment
- The bus ranges on each side of the assignment statement must be the same number of bits

Logical Shift Left (better way)

```
module Lshift ( din, shift_en, dout);  
  input [7:0] din;  
  input shift_en;  
  output [7:0] dout;  
  
  reg dout;  
  
  always@(din or shift_en) begin  
    dout = din; //default  
    if (shift_en == 1) begin  
      dout = {din[6:0], 0};  
    end  
  end  
endmodule
```

Create a unique
bus using { }



Concatenate signals using the { } operator

```
assign {b[7:0],b[15:8]} = {a[15:8],a[7:0]};
```

byte swap

Logical Shift Left (better way)

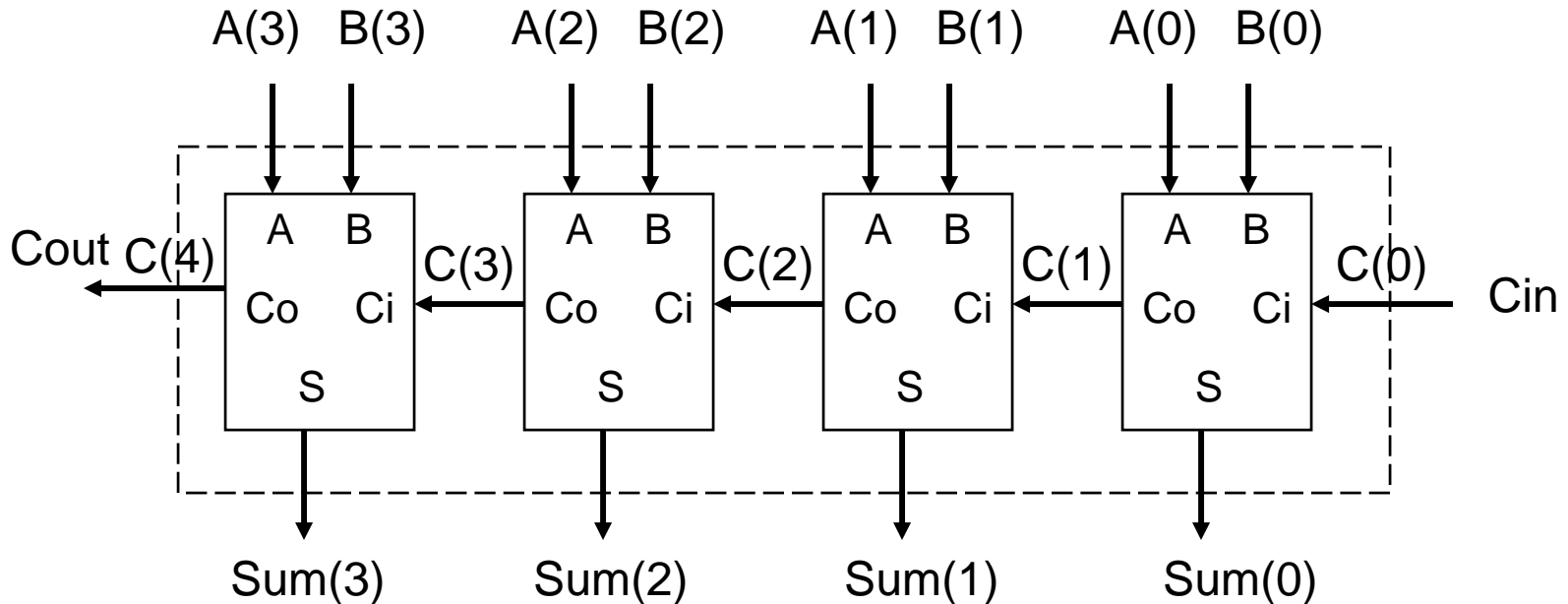
```
module Lshift ( din, shift_en, dout);
  input [7:0] din;
  input shift_en;
  output [7:0] dout;

  reg dout;

  always@(din or shift_en) begin
    dout = din; //default
    if (shift_en == 1) begin
      dout = din << 1;
    end
  end
endmodule
```

- Will shift **dout** left one bit, but implied shifting in a zero
- Can't shift in a one

4-Bit Ripple Carry Adder



Want to write a Verilog HDL model for a 4 bit ripple carry adder.

Logic equation for each full adder is:

```
sum  <=  a xor b xor ci;  
co   <=  (a and b) or (ci and (a or b));
```

Loop Statements

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

4-Bit Ripple Carry Model using *for*

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

  reg cout, sum;
  reg [4:0] c; // for internal carries
  integer i; // counter in the for loop

  always@(a or b or c or cin) begin
    c[0] = cin;
    for(i=0; i<4; i=i+1) begin
      sum[i] = a[i] ^ b[i] ^ c[i];
      c[i+1] = (a[i] & b[i]) | (c[i] & (a[i] | b[i]));
    end
    cout = c[4];
  end
endmodule
```

4-Bit Ripple Carry unrolled

```
always@(a or b or c or cin) begin
    c[0] = cin;

    sum[0] = a[0] ^ b[0] ^ c[0];
    c[1] = (a[0] & b[0]) | (c[0] & (a[0] | b[0]));

    sum[1] = a[1] ^ b[1] ^ c[1];
    c[2] = (a[1] & b[1]) | (c[1] & (a[1] | b[1]));

    sum[2] = a[2] ^ b[2] ^ c[2];
    c[3] = (a[2] & b[2]) | (c[2] & (a[2] | b[2]));

    sum[3] = a[3] ^ b[3] ^ c[3];
    c[4] = (a[3] & b[3]) | (c[3] & (a[3] | b[3]));

    cout = c[4];

end
```

The “for” loop is unrolled into code like this before it’s implemented.

Comments on *for-loop* Statement

- The `for-loop` can be used to repeat blocks of logic
 - The loop variable `i` is implicitly declared for this loop; does not have to be declared anywhere else.
 - To visualize what logic is created, 'unroll' the loop by writing down each loop iteration with loop indices replaced hard numbers.
-

Verilog Built-in Arithmetic

- Verilog's built-in arithmetic makes a 32-bit adder easy:

```
module add32(a, b, sum);  
  input[31:0] a,b;  
  output[31:0] sum;  
  assign sum = a + b;  
endmodule
```

- A 32-bit adder with carry-in and carry-out:

```
module add32_carry(a, b, cin, sum, cout);  
  input[31:0] a,b;  
  input cin;  
  output[31:0] sum;  
  output cout;  
  assign {cout, sum} = a + b + cin;  
endmodule
```

Key Notes

- There are many different ways to write Verilog synthesizable models for combinational logic
 - There is no 'best' way to write a model; for now, just use the statements/style that you feel most comfortable with and can get to work
 - There is NO WAY that we can cover all possible examples in class.
 - I have intentionally left out MANY, MANY language details. You can get by with what I have shown you, but feel free to experiment with other language features that you see discussed in the book or elsewhere.
-

More Key Notes

- Search the Internet
 - The WWW is full of Verilog examples, tutorials, etc.
 - Try it out
 - If you have a question about a statement or example, try it out in the Xilinx ISE package and see what happens!
 - This course is about Digital System Design, not Verilog. As such, we will only have 3-4 lectures about Verilog, the rest will be on important design topics.
 - Verilog is only a means for efficiently implementing your design - it is not interesting by itself.
 - You will probably learn multiple synthesis languages in your design career - it is the digital design techniques that you use that will be common to your designs, not the synthesis language.
-