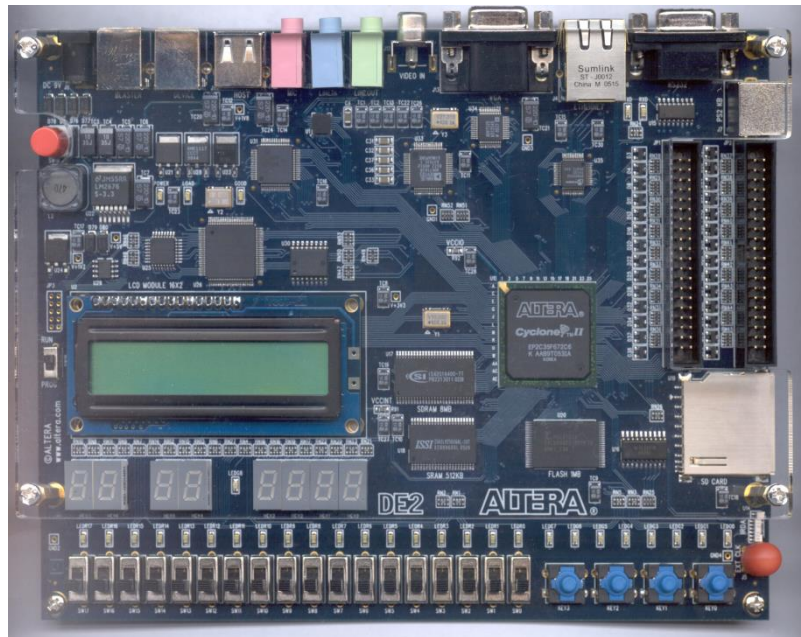

CPE 322

Digital Hardware Design Fundamentals

Electrical and Computer Engineering
UAH

Introduction to Verilog



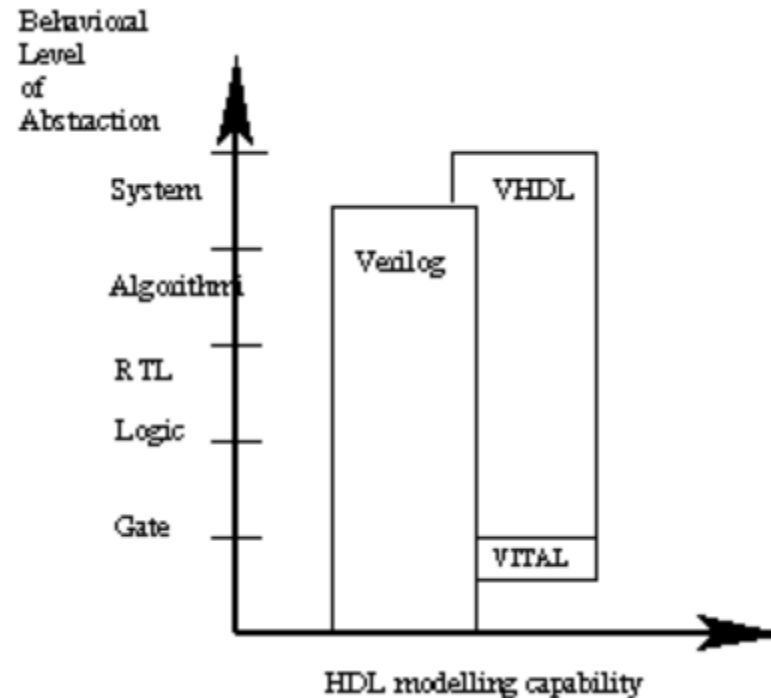
Hardware Description Languages

- Modeling Language that supports
 - Design Entry
 - Simulation
 - Rapid Prototyping of Designs
 - Synthesis (design creation)
- Promotes top-down design methodologies
- Promote the creation of technology independent Intellectual Property

Hardware Description Languages

- Support modeling at different levels of abstraction:
 - Structural (specifying interconnections of the gates)
 - Dataflow (specifying design using Boolean Logic)
 - Equations that specify output signals in terms of input signals.
 - Behavioral (specifying high-level behavior)

Common Modeling Constructs



<http://www.angelfire.com/in/rajesh52/verilogvhdl.html>

Verilog HDL Levels of Abstraction

- Structural Design (hierarchal design approach)
- Data Flow (equation based)
- Behavioral Design (modeling using highly abstract language constructs)

- **Most often Verilog HDL models contain aspects of all three levels of abstraction!**

Names (Identifiers)

- Must begin with alphabetic or underscore characters a-z A-Z _
- May contain the characters a-z A-Z 0-9 _ and \$
- May use any character by escaping with a backslash (\) at the beginning of the identifier, and terminating with a white space.

Examples	Notes
<code>adder</code>	legal identifier name
<code>XOR</code>	uppercase identifier is unique from xor keyword
<code>\reset*</code>	an escaped identifier (must be followed by a white space)

Verilog Logic Values

- Verilog HDL support 4 Logic Values

Logic Value	Description
0	zero, low, or false
1	one, high, or true
z or Z	high impedance (tri-stated or floating)
x or X	unknown or uninitialized

Verilog Logic Strengths

- Verilog HLD has 8 strengths: 4 driving, 3 capacitive, and 1 high impedance (no strength)

Strength Level	Strength Name	Specification Keyword		Display Mnemonic	
7	Supply Drive	supply0	supply1	Su0	Su1
6	Strong Drive	strong0	strong1	St0	St1
5	Pull Drive	pull0	pull1	Pu0	Pu1
4	Large Capacitive	large		La0	La1
3	Weak Drive	weak0	weak1	We0	We1
2	Med. Capacitive	medium		Me0	Me1
1	Small Capacitive	small		Sm0	Sm1
0	High Impedance	highz0	highz1	HiZ0	HiZ1

Integer Representation

size'base value – sized integer in the specified base

size -- number of bits in the number (optional)

'base -- number of bits in the number (optional)

value

Base	Symbol	Legal Values
binary	b or B	0, 1, x, X, z, Z, ?, _
octal	o or O	0-7, x, X, z, Z, ?, _
decimal	d or D	0-9, _
hexadecimal	h or H	0-9, a-f, A-F, x, X, z, Z, ?, _

Notes: ? Is same as Z or z; _ (underscore) is ignored

Integer Representation (Examples)

Examples	Size	Base	Binary Equivalent
<code>10</code>	unsized	decimal	0...01010 (32-bits)
<code>'o7</code>	unsized	octal	0...00111 (32-bits)
<code>1'b1</code>	1 bit	binary	1
<code>8'Hc5</code>	8 bits	hex	11000101
<code>6'hF0</code>	6 bits	hex	110000 (truncated)
<code>6'hF</code>	6 bits	hex	001111 (zero filled)
<code>6'hZ</code>	6 bits	hex	zzzzzz (Z filled)

Verilog Model Structure

```
module module_name (port list );  
    port declarations;  
    ...  
    variable declaration;  
    ...  
    description of behavior  
endmodule
```

Systems are
described as a set
of modules

```
module module_name (port declarations and list);  
    ...  
    variable declaration;  
    ...  
    description of behavior  
endmodule
```

Modules

- Provides external interface to other modules and/or the outside world
 - *inputs, outputs, and inouts*
- Contains internal logic that performs the module's function
- Encapsulates the internal signals, logic, and sub-modules that are referenced by the module.

Verilog Signal Types

- Two Main Types of Signals:
 - “Wire” is a simple connection between two components
 - Should have just one driver
 - Value is not retained unless driven – i.e.no persistence
 - Wire is the default if not specified in port declarations
i.e. **input x;** is the same as **input wire x;**
 - “Reg” stores its value until it is updated again
 - Required in behavioral designs that have procedural statements

Verilog Net Data Types

- The 'wire' type is an instance of the allowable net types.

Keyword	Functionality
wire or tri	Simple interconnecting wire
wor or trior	Wired outputs OR together
wand or triand	Wired outputs AND together
tri0	Pulls down when tri-stated
tri1	Pulls up when tri-stated
supply0	Constant logic 0 (supply strength)
supply1	Constant logic 1 (supply strength)
triereg	Stores last value when tri-stated (capacitance strength)

The 'wire' type is of primary importance in this course.

Verilog Net Data Types

- In general, Net data types are used to connect structural components together.
 - They transfer both logic values and logic strengths.
 - They must be used when
 - A signal is driven by the output of some device.
 - A signal is also declared as an *input* port or an *inout* port.
 - A signal is on the LHS of a continuous assignment statement.

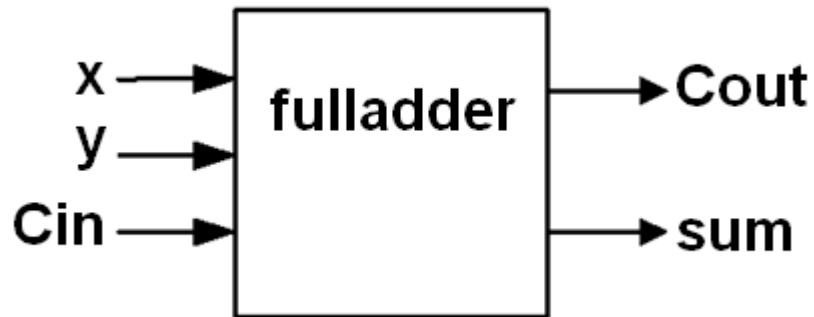
Structural Design

- Structural Modeling is a hierarchical design approach that is analogous to schematic capture.
- The design is partitioned into manageable hardware blocks that are called *components*.
- Each subcomponent is modeled separately but can reside in the same file with other component modules.
- Multiple instances of these subcomponents are then be incorporated into the module and interconnected together to create a more complex design.
- The resulting design then becomes a component itself that can be used as one of the building blocks of an even more complex design.

Lower-Level Component Model

(Structural Decomposition)

Full Adder Example



```
module fulladder(input x,y,Cin, output Cout,sum);  
  
    wire n0,n1,n2; // internal nodes  
  
    xor G0(sum,x,y,Cin);  
    and G1(n0,x,y);  
    and G2(n1,x,Cin);  
    and G3(n2,y,Cin);  
    or  G4(Cout,n0,n1,n2);  
  
endmodule
```

```
module fulladder(x,y,Cin,Cout,sum);  
    input x,y,Cin;  
    output Cout,sum;  
  
    wire n0,n1,n2; // internal nodes  
  
    xor G0(sum,x,y,Cin);  
    and G1(n0,x,y);  
    and G2(n1,x,Cin);  
    and G3(n2,y,Cin);  
    or  G4(Cout,n0,n1,n2);  
  
endmodule
```

Verilog's Built-in Primitives

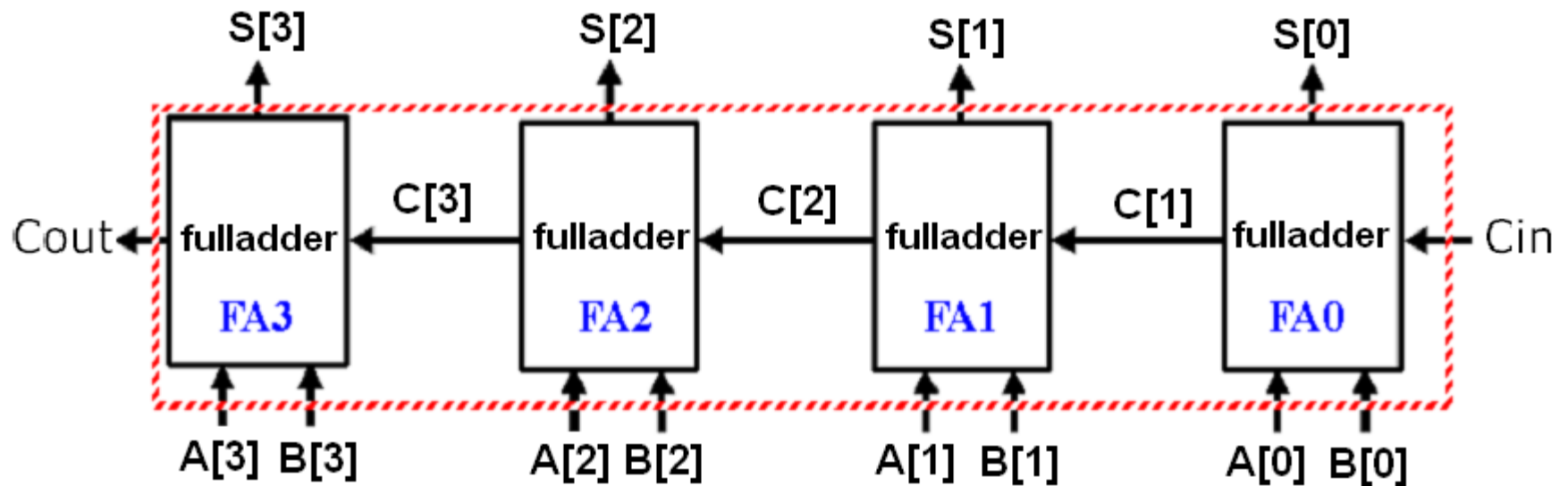
Gate Type		Terminal Order
and or xor	nand nor xnor	(1_output, 1-or-more_inputs)
buf	not	(1-or-more_outputs, 1_input)
bufif0 bufif1	notif0 notif1	(1_output, 1_input, 1_control)
pullup	pulldown	(1_output)
user-defined-primitives		(1_output, 1-or-more_inputs)

Switch Type		Terminal Order
pmos nmos	rpmos rnmos	(1_output, 1_input, 1_control)
cmos	rcmos	(1_output, 1_input, n_control, p_control)
tran	rtran	(2_bidirectional-inouts)
tranif0 rtranif0	rtranif1 rtranif1	(2_bidirectional-inouts, 1_control)

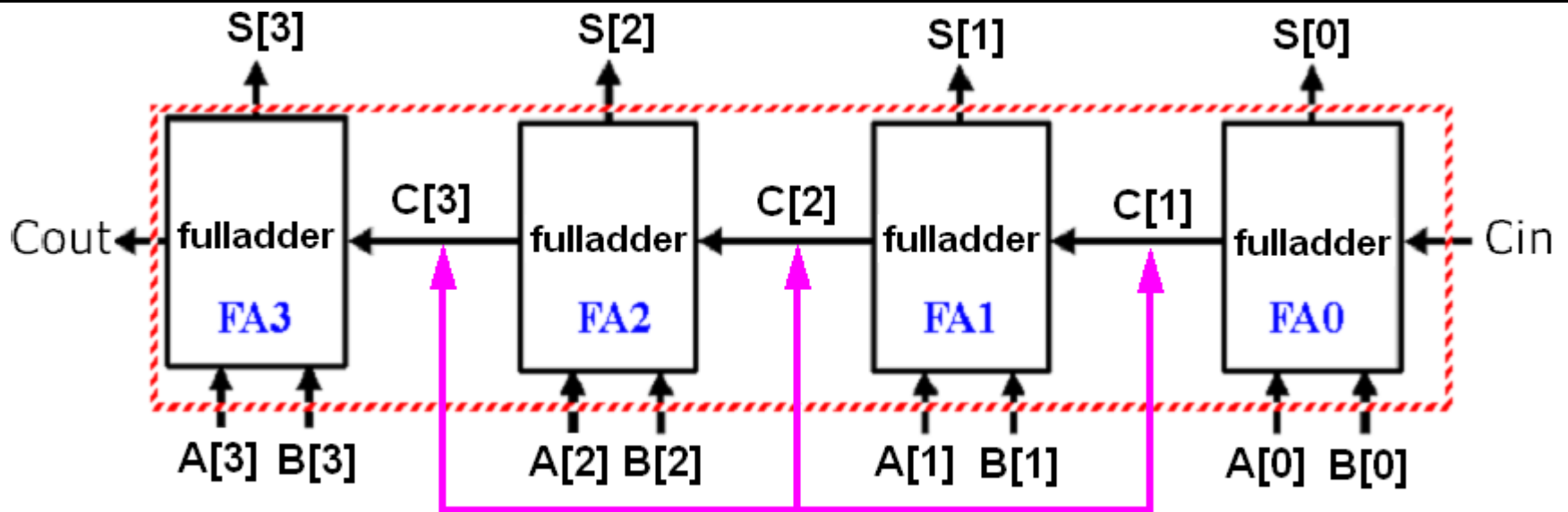
Structural Design

- Components are individually modeled as a separate modules
- A new module is then created that contains this set of components.
- Subcomponent modules are “wired” together by connecting by interconnecting them together using *actual signals*.
 - The input and output signals of the components (component port names) are called *formal signals*
- Instances of a subcomponent module are made by specifying the component type, instance name, and port/signal association parameters.
- Either *Port Order* or *Port Name* association is used to bind the component I/O port *formal* signals to the modules own internal *actual signals*.

4-bit Adder



4-bit Adder (cont'd)



Wires that are visible only within the *fulladder* module

```
module adder4(input Cin, input [3:0] A,B, output Cout, output [3:0] S);  
    wire [3:1] C;  
    *  
    *  
    *  
endmodule
```

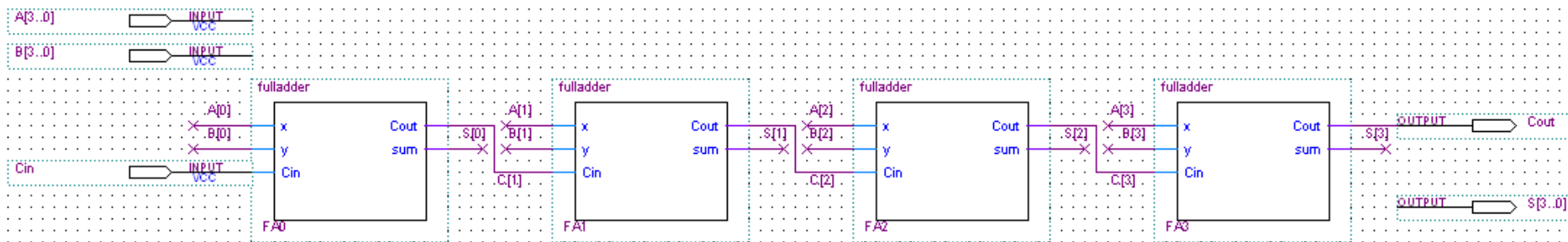
Structural Design

Model Instance Port Order Association

- The *actual* signals in the model instance statement must be placed in the same positions as the corresponding *formal* signals were listed in the component declaration.
 - *module_name* *instance_name*(*signal*, *signal*, ...);
- This is the method most often referenced in the textbook.

4-bit Adder (cont'd)

(port order association method)



```
module adder4(input Cin, input [3:0] A,B, output Cout, output [3:0] S);
```

```
    wire [3:1] C;
```

```
    fulladder    FA0(A[0],B[0],Cin,C[1],S[0]);
    fulladder    FA1(A[1],B[1],C[1],C[2],S[1]);
    fulladder    FA2(A[2],B[2],C[2],C[3],S[2]);
    fulladder    FA3(A[3],B[3],C[3],Cout,S[3]);
```

```
endmodule
```

```
module fulladder(input x,y,Cin, output Cout,sum);
```

```
    wire n0,n1,n2; // internal nodes
```

```
    xor G0(sum,x,y,Cin);
    and G1(n0,x,y);
    and G2(n1,x,Cin);
    and G3(n2,y,Cin);
    or  G4(Cout,n0,n1,n2);
```

```
endmodule
```

Note: *actual* signals must be placed in exactly the same positions in the model instance statement as the corresponding *formal* signals were listed in the component's model declaration.

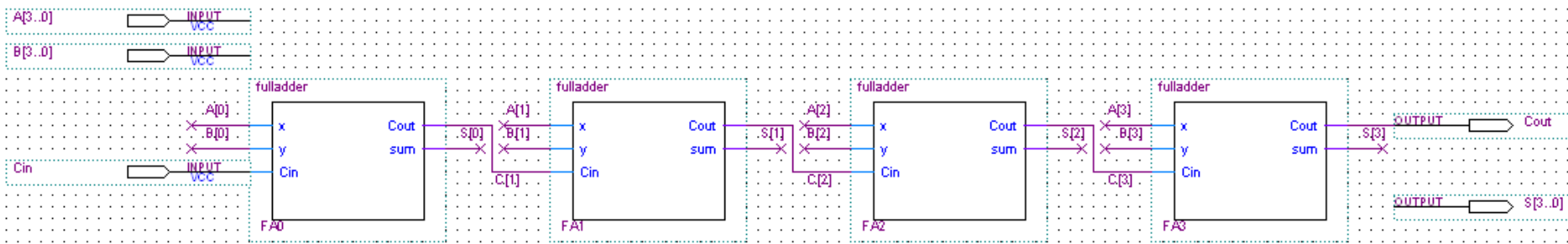
Structural Design

Module Instance Port Name Association

- The *formal signals* are those *inputs/outputs/or inout* signals that are in the port declarations of the given submodule.
- These signals must be liked (wired) to the *actual signals* in the current module.
 - This is done during component instantiation using the . operator that proceeds the formal signal name and then encapsulating the *actual signal name* in parenthesis in the component instantiation,
 - `module_name instance_name(.port_name(signal), .port_name(signal), ...);`

4-bit Adder Structural Design

(port signal name association method)



```
module adder4(input Cin, input [3:0] A,B, output Cout, output [3:0] S);  
  
    wire [3:1] C;  
  
    fulladder  FA0 (.x(A[0]), .y(B[0]), .Cin(Cin), .Cout(C[1]), .sum(S[0]));  
  
    fulladder  FA1 (.x(A[1]), .y(B[1]), .Cin(C[1]), .Cout(C[2]), .sum(S[1]));  
  
    fulladder  FA2 (.x(A[2]), .y(B[2]), .Cin(C[2]), .Cout(C[3]), .sum(S[2]));  
  
    fulladder  FA3 (.x(A[3]), .y(B[3]), .Cin(C[3]), .Cout(Cout), .sum(S[3]));  
  
endmodule
```

- It does not matter the order that these signals are placed in the port map statement.
- More intuitive for many and less error prone.

Data Flow


- Method primarily employs RTL (register transfer language) Boolean Equations to describe the functionality of the design
- Method is suitable to model both combinational (memory-less) and sequential logic
- Represents a relatively low level of abstraction. Complexity is very high for complex designs.
- Can be combined with structural methodologies to incorporate a hierarchy that will aid in managing this complexity.

Verilog Operators

Verilog Operator	Name	Functional Group
[]	bit-select or part-select	
()	parenthesis	
!	logical negation	logical
~	negation	bit-wise
&	reduction AND	reduction
	reduction OR	reduction
~&	reduction NAND	reduction
~	reduction NOR	reduction
^	reduction XOR	reduction
^ or ^~	reduction XNOR	reduction
+	unary (sign) plus	arithmetic
-	unary (sign) minus	arithmetic
{ }	concatenation	concatenation
{ () }	replication	replication
*	multiply	arithmetic
/	divide	arithmetic
%	modulus	arithmetic
+	binary plus	arithmetic
-	binary minus	arithmetic

Verilog Operator	Name	Functional Group
<<	shift left	shift
>>	shift right	shift
>	greater than	relational
>=	greater than or equal to	relational
<	less than	relational
<=	less than or equal to	relational
==	logical equality	equality
!=	logical inequality	equality
===	case equality	equality
!==	case inequality	equality
&	bit-wise AND	bit-wise
^	bit-wise XOR	bit-wise
^~ or ^~	bit-wise XNOR	bit-wise
	bit-wise OR	bit-wise
&&	logical AND	logical
	logical OR	logical
?:	conditional	conditional

Verilog Operators Order of Precedence

Operator	Precedence
<code>+, -, !, ~ (unary)</code>	Highest
<code>*, /, %</code>	
<code>+, - (binary)</code>	
<code><<, >></code>	
<code><, <=, >, >=</code>	
<code>=, ==, !=</code>	
<code>===, !==</code>	
<code>&, ~&</code>	
<code>^, ^~</code>	
<code> , ~ </code>	
<code>&&</code>	
<code> </code>	
<code>?:</code>	Lowest

Continuous Assignment Statements

(Implicit form)

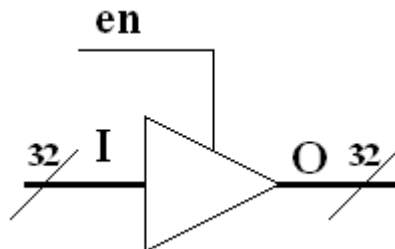
- *net_declaration* = expression;
- Example:
 - wire sum = x ^ y ^ Cin;

Continuous Assignment Statements (Explicit form)

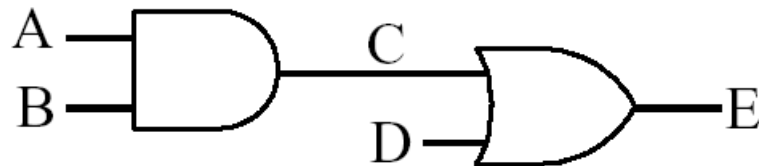
- Format: **assign** net = expression;
- Can use conditional operator ?
 - <condition> ? <if true> : <else>;
If the Condition is true, the value of <if true> will be taken, otherwise the value of <else> will be taken.

Example:

assign O = en ? I : 32'bz;



Data Flow Representation



Continuous Assignment Statements

`assign C = A & B;`

`assign E = C | D;`

Order of continuous assignment statements is not important

`assign C = A & B;`

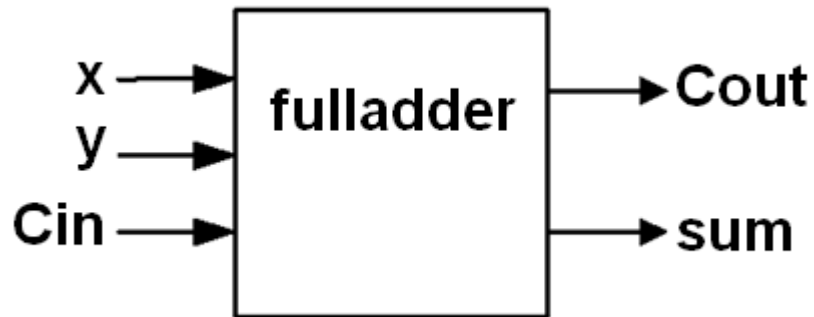
`assign E = C | D;`

- These signal are assumed to be evaluated by the Verilog HDL circuit synthesizer or simulator concurrently.

Low-Level Component Model

(Data Flow Example)

Full Adder Example



```
module fulladder(input x,y,Cin,output Cout,sum);  
  
    assign sum = x ^ y ^ Cin;  
    assign Cout = (x & y) | (x & Cin) | (y & Cin);  
  
endmodule
```


Behavioral Modeling using Procedural Constructs

- Two Procedural Constructs
 - **initial** Statement
 - **always** Statement
- **initial** Statement : Executes only once
- **always** Statement : Executes in a loop
- Example:

```
...  
initial begin  
  Sum = 0;  
  Carry = 0;  
end  
...
```

```
...  
always @(A or B) begin  
  Sum = A ^ B;  
  Carry = A & B;  
end  
...
```

Event Control

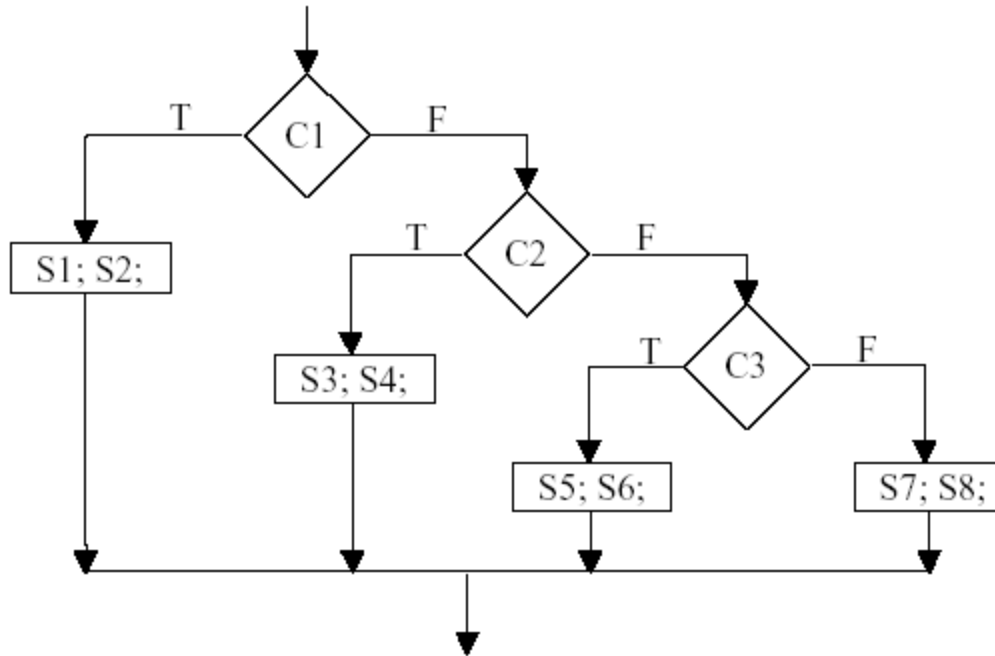
- Event Control
 - Edge Triggered Event Control
 - Level Triggered Event Control
- Edge Triggered Event Control
 - @ (posedge CLK) //Positive Edge of CLK

Curr_State = Next_state;

@ negedge	@ posedge
1 → x	0 → x
1 → z	0 → z
1 → 0	0 → 1
x → 0	x → 1
z → 0	z → 1

- Level Triggered Event Control
 - @ (A or B) //change in values of A or B
- Out = A & B;

IF Statements in Verilog HDL

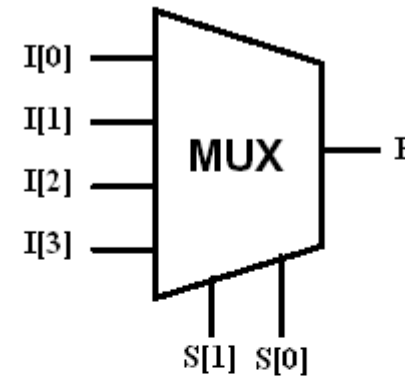


```
if (C1)
    begin
        S1;
        S2;
    end
else
    if (C2)
        begin
            S3;
            S4;
        end
    else
        if (C3)
            begin
                S5;
                S6;
            end
        else
            begin
                S7;
                S8;
            end
        end
    end
end
```

Case Statement in Verilog HDL

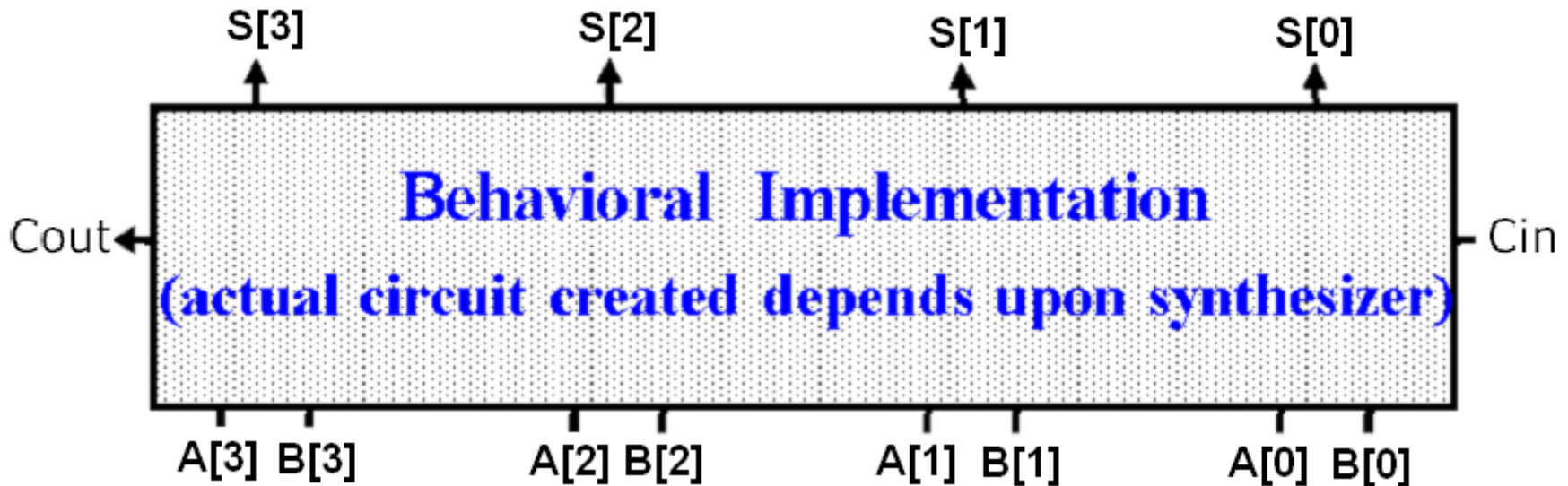
The case statement has the general form:

```
case (expression)
    choice1 : procedural statement;
    choice2 : procedural statement;
    . . .
    default : procedural statement;
endcase
```



```
case (S)
    0 : F = I[0];
    1 : F = I[1];
    2 : F = I[2];
    3 : F = I[3];
    default : F = 1'bx;
endcase
```

Behavioral Implementation



```
module adder4(input Cin, input [3:0] A,B, output Cout, output [3:0] S);  
  
    always @(A or B or Cin)  
        {Cout,S} = A + B + Cin;  
  
endmodule
```