

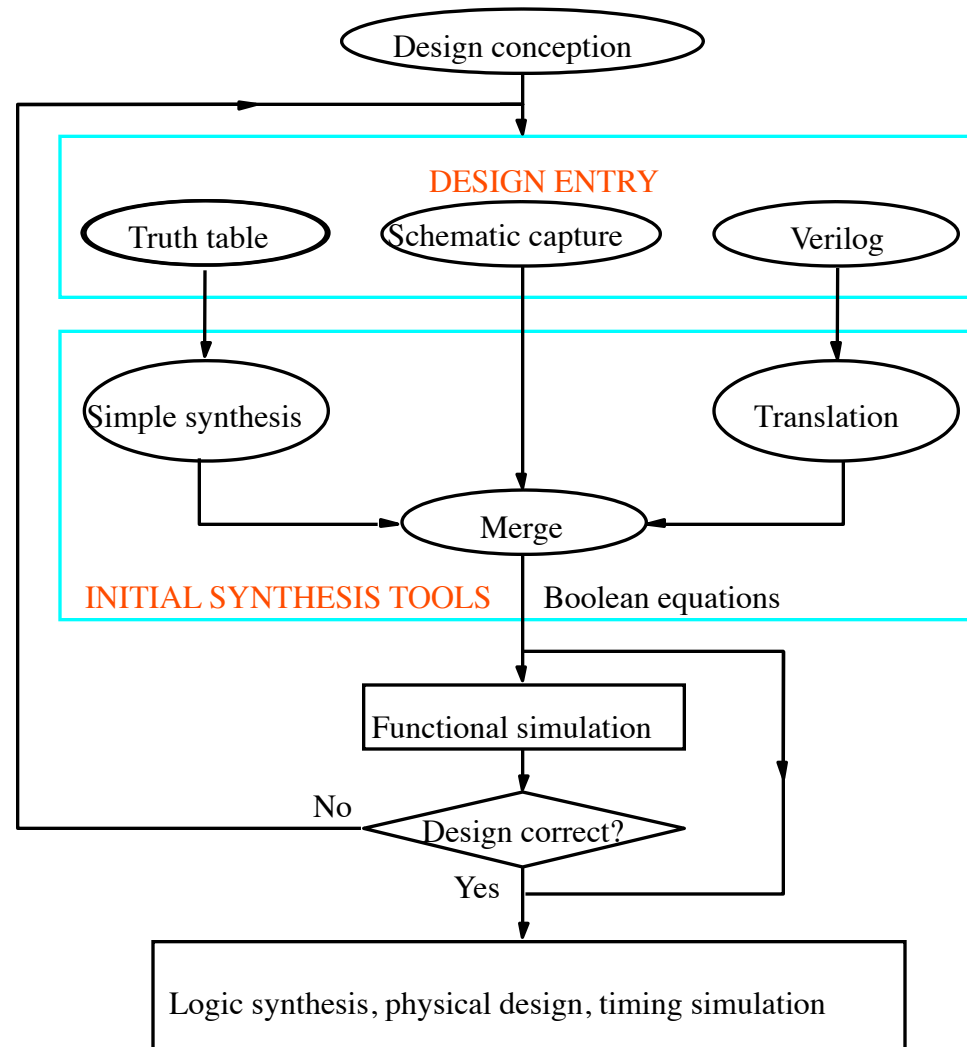
A decorative graphic consisting of a thin yellow circle. A horizontal bar with a yellow-to-white gradient is positioned across the middle of the circle. On the left side of the bar, there is a large black left square bracket. On the right side of the bar, there is a large yellow right square bracket.

Verilog HDL

# ■ Hardware Description Language

- HDL – a “language” for describing hardware
- Two industry IEEE standards:
  - Verilog
  - VHDL (Very High Speed Integrated Circuit HDL)
- Verilog
  - Originally designed for simulation and verification
    - Gateway Design Automation, 1983
  - Functionality later added for synthesis

# CAD Design Process



# Synthesis Using Verilog

- There are two ways to describe circuits in Verilog
  - Structural
    - Use Verilog “gate” constructs to represent logic gates
    - Write Verilog code that connects these parts together
  - Behavioral flow
    - Use *procedural* and “*assign*” constructs to indicate what actions to take
    - The Verilog “compiler” converts to the schematic for you

# Verilog Miscellanea

- Comments: just like C++
  - `//` this is a single line comment
  - `/*` this is a multi-line comment `*/`
- White space (spaces, tabs, blank lines) is ignored
- Identifier names
  - Letters, digits, `_`, `$`
  - Can't start with a digit
  - Case-sensitive!
  - There are also reserved words ← can't be one of these

# Verilog Types

- There are NO user-defined types
- There are only two data types:
  - wire
    - Represents a physical connection between structural elements (think of this as a wire connecting various gates)
    - This is the most common type for structural style
    - *wire* is a net type and is the default if you don't specify a type
  - reg
    - Represents an abstract storage element (think of this as an unsigned integer variable)
    - This is used in the behavioral style only

# Verilog Modules and Ports

- Circuits and sub-circuits are described in *modules*
  - `module ... endmodule`
- The arguments to the module are called *ports*
  - Ports are of type
    - `input`
    - `output`
    - `inout`      ← bidirectional port; of limited use to us
  - Ports can be scalar (single bit) or vector (multi-bit)
    - Ports are a scalar wire type unless you specify otherwise
    - Example:  
    `input [3:0] A; // a 4 bit (vector) input port (wire) called A`

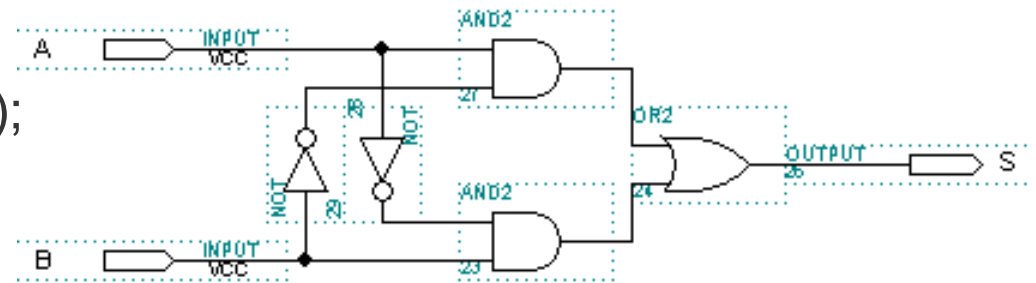
# Verilog Structural Synthesis

- The half-adder (sum part only):

```
module halfAdder(S,A,B);  
  input A, B;  
  output S;  
  wire AC, BC, X, Y;
```

```
    not(AC,A);           // AC ← ~A  
    not(BC,B);           // BC ← ~B  
    and(X,A,BC);         // X ← A & BC  
    and(Y,B,AC);         // Y ← B & AC  
    or(S,X,Y);           // S ← X | Y
```

```
endmodule
```





# Verilog Primitive Gates

- `and(f,a,b,...)`       $f = (a \cdot b \dots)$
- `or(f,a,b,...)`       $f = (a + b + \dots)$
- `not(f,a)`       $f = a'$
- `nand(f,a,b,...)`       $f = (a \cdot b \dots)'$
- `nor(f,a,b,...)`       $f = (a + b + \dots)'$
- `xor(f,a,b,...)`       $f = (a \oplus b \oplus \dots)$
- `xnor(f,a,b,...)`       $f = (a \odot b \odot \dots)$
- You can also “name” the gates if you like ...
  - `and And1(z1,x,y);`    // this gate is named And1

# ■ Comments on Structural Design

- Order of statements in structural style is irrelevant
  - This is NOT like a typical programming language
  - Everything operates concurrently (in parallel)
- Primitive gates can have any number of inputs (called *fan in*)
  - The Verilog compiler will decide what the practical limit is and build the circuit accordingly (by cascading the gates)
- All structural elements are connected with *wires*
  - Wire is the default type, so sometimes the “declarations” are omitted

## ■ Example: 3 Way Light Control

### ○ Example: 3 way light control

- 1 light L
- 3 light switches (A, B, C)
- A is a master on/off:
  - If A = off, then light L is always off
  - If A = on, the behavior depends on B and C only
- Let 0 represent “off”
- Let 1 represent “on”
- SOP Equation?

A	B	C	L
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

$$\text{SOP: } L = ABC' + AB' C$$

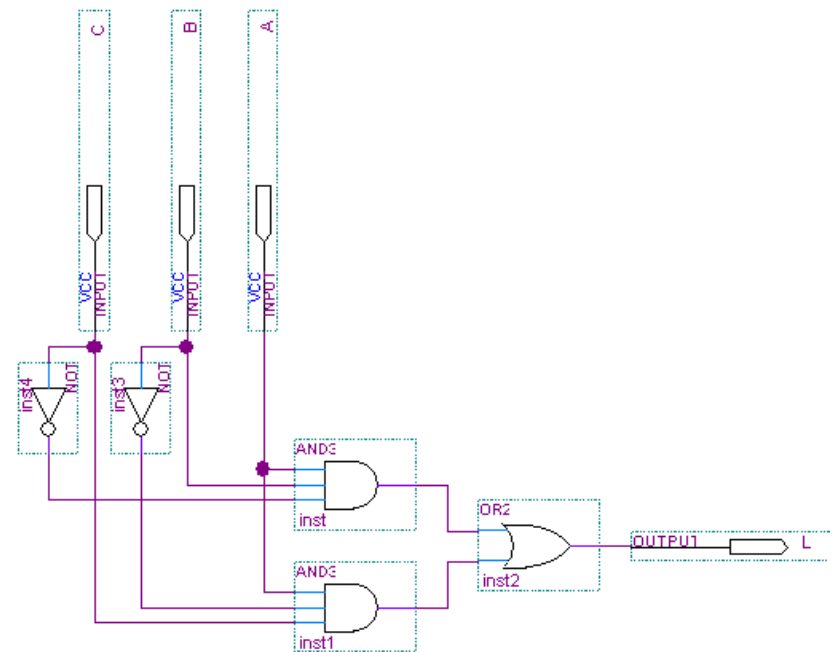
## Example: 3 Way Light Control

- The SOP implementation of the 3 way light control with gates:

with Verilog:

```
module light3Way(L,A,B,C);  
    input A, B, C;  
    output L;
```

```
    not Not1(BC,B), Not2(CC,C);  
    and And1(f,A,B,CC), And2(g,A,BC,C);  
    or Or1(L,f,g);  
endmodule
```



# ■ Behavioral Synthesis Using Verilog

- Behavioral style uses Boolean algebra to express the behavior of the circuit

- $L = (A B' C) + (A B C')$  for the 3 way light control

```
module light3Way(L,A,B,C);  
    input A, B, C;  
    output L;  
  
    assign L = (A & ~B & C) | (A & B & ~C);  
endmodule
```

# Continuous Assignment

- The *assign* statement is key in data flow synthesis
  - The RHS is recomputed when a value in the RHS changes
  - The new value of the RHS is assigned to the LHS after the propagation delay (default delay is 0)
  - The LHS must be a net (i.e. wire) type
  - Example: `assign L = (A & ~B & C) | (A & B & ~C);`
    - If A or B or C changes, then L is reassigned
- The Verilog compiler will construct the schematic for you

# Operators

Category	Examples	Bit length
Bitwise	$\sim A$ , $A \& B$ , $A \mid B$ , $A \wedge B$ (xor), $A \sim \wedge B$ (xnor)	$L(A)$
Logical	$!A$ , $A \&\&B$ , $A \parallel B$	1
Reduction	$\&A$ , $\sim\&A$ , $ A$ , $\sim A$ , $\wedge\sim A$ , $\sim\wedge A$	1
Relational	$A == B$ , $A != B$ , $A > B$ , $A < B$ , $A >= B$ , $A <= B$	1
Arithmetic	$A + B$ , $A - B$ , $-A$ , $A * B$ , $A / B$	$\text{Max}(L(A), L(B))$
Shift	$A \ll B$ , $A \gg B$	$L(A)$
Concatenate	$\{A, \dots, B\}$	$L(A) + \dots + L(B)$
Replication	$\{B\{A\}\}$	$B * L(A)$
Condition	$A ? B : C$	$\text{Max}(L(B), L(C))$

## ■ Behavioral Style for Full-Adder

```
module fullAdder(S,Cout,A,B,Cin);  
    input A,B,Cin;  
    output S,Cout;  
  
    assign S = A ^ B ^ Cin;  
    assign Cout = (A & B) | (A & Cin) | (B & Cin);  
endmodule
```



## ■ Another Behavioral Style for Full-Adder

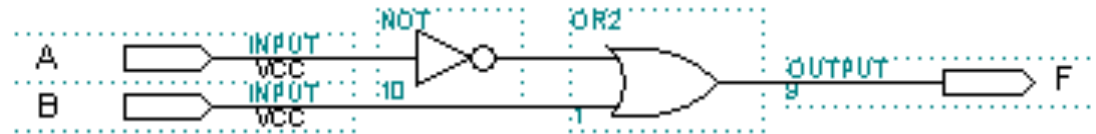
```
module fullAdder(S,Cout,A,B,Cin);  
    input A,B,Cin;  
    output S,Cout;  
  
    assign {Cout,S} = A+B+Cin; // + means addition  
endmodule
```

- Verilog also supports **arithmetic** operations!
  - compiles them down to library versions of the circuit

## Gate Delays

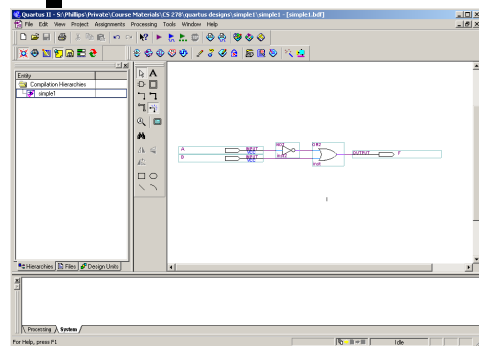
- Gates may be specified with a delay – optional

not #3 Not1(x,A);  
or #5 Or1(F,B,x);

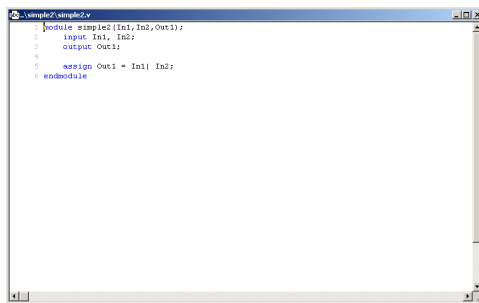


- The propagation delay is in “units” which can be set independently from the code
- Propagation delays have no meaning for synthesis, only for simulation
  - Hence, we won't use them

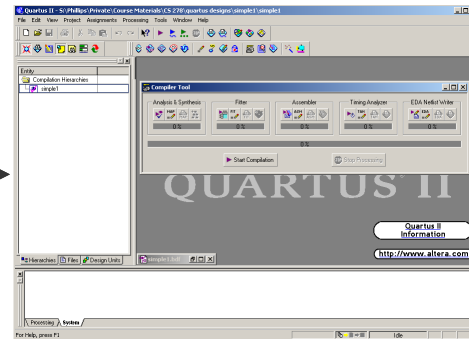
# Typical Design Process



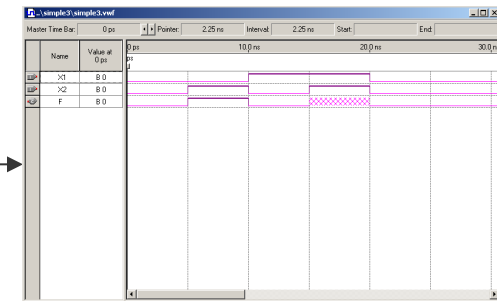
Graphical Entry



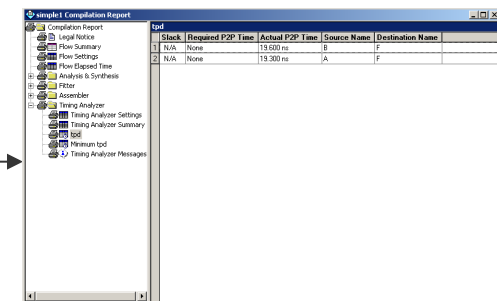
HDL Entry



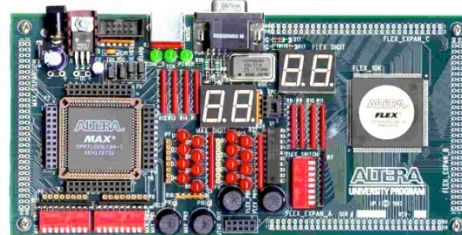
Compiler



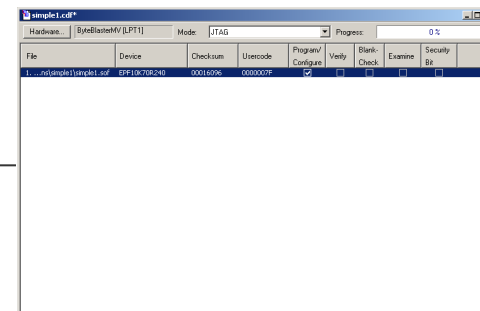
Waveform/Timing Diagram



Timing Analysis



UP2 Board



Program the FPGA



# Combinational Logic in Verilog

# ■ Verilog for Combinational Circuits

- How can Verilog be used to describe the various combinational building blocks?
  - Can always use structural style
    - This can get tedious
  - “Data flow” style is more convenient
    - `assign x = a & ~b` (etc, etc)
  - Behavioral style is also good – we’ll look at this shortly

## ■ Verilog: 2x1 MUX

```
module mux2x1(f,w0,w1,s);  
    input w0, w1, s;  
    output f;  
  
    assign f = s ? w1 : w0; // if s == 1, then w1, else w0  
endmodule
```

- Uses the conditional ?: operator
  - Software designers detest this operator
  - Hardware designers revel in its beauty

## ■ Verilog: 4x1 MUX (Data Flow Style)

```
module mux4x1(f,w0,w1,w2,w3,s1,s0);  
    input w0, w1, w2, w3, s1, s0;  
    output f;  
  
    assign f = s1 ? (s0 ? w3 : w2) : (s0 ? w1 : w0);  
endmodule
```

- This is getting complicated!
- Need a way to specify a “group” of bits like w[0:3]
- Need a way to replace ?: with “if then else”

# ■ Vectored Signals in Verilog

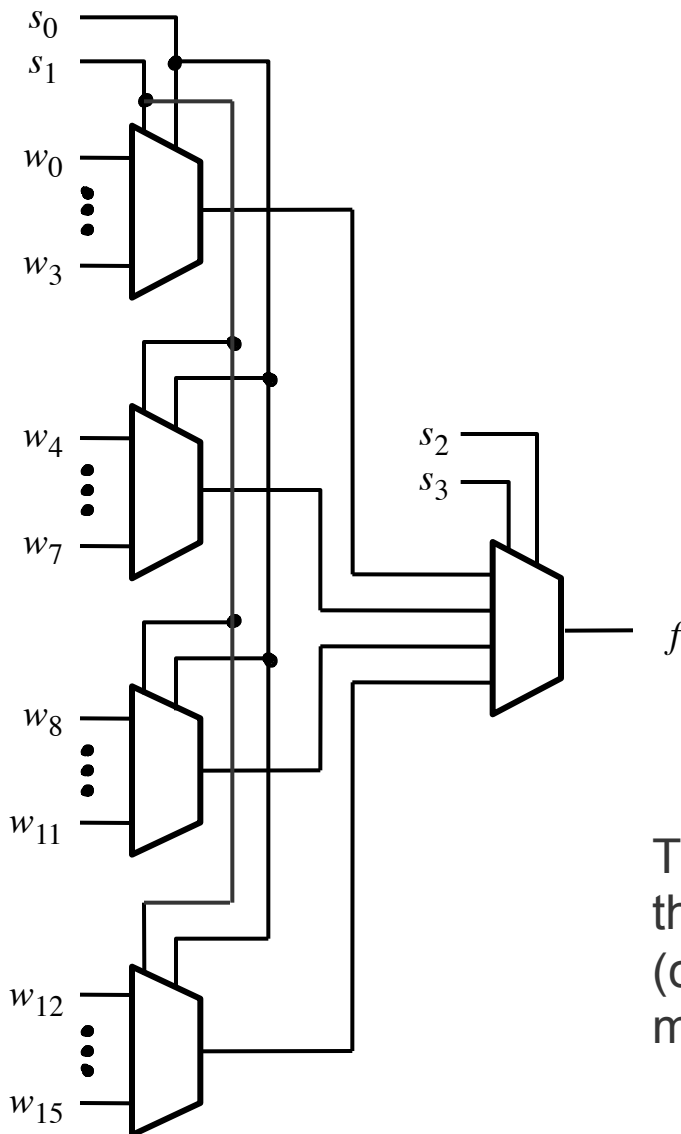
```
module mux4x1(f,W,S);  
    input [0:3] W;  
    input [1:0] S;  
    output f;  
  
    assign f = S[1] ? (S[0] ? W[3] : W[2])  
               : (S[0] ? W[1] : W[0]);  
endmodule
```

Format is [MSB:LSB]

- Signals can be grouped as bit vectors
  - The order of the bits is user determined
  - W has 4 lines with the MSB = W[0] and the LSB = W[3]
  - S has two lines with the MSB = S[1] and the LSB = S[0]



# Hierarchical Design of a 16x1 MUX

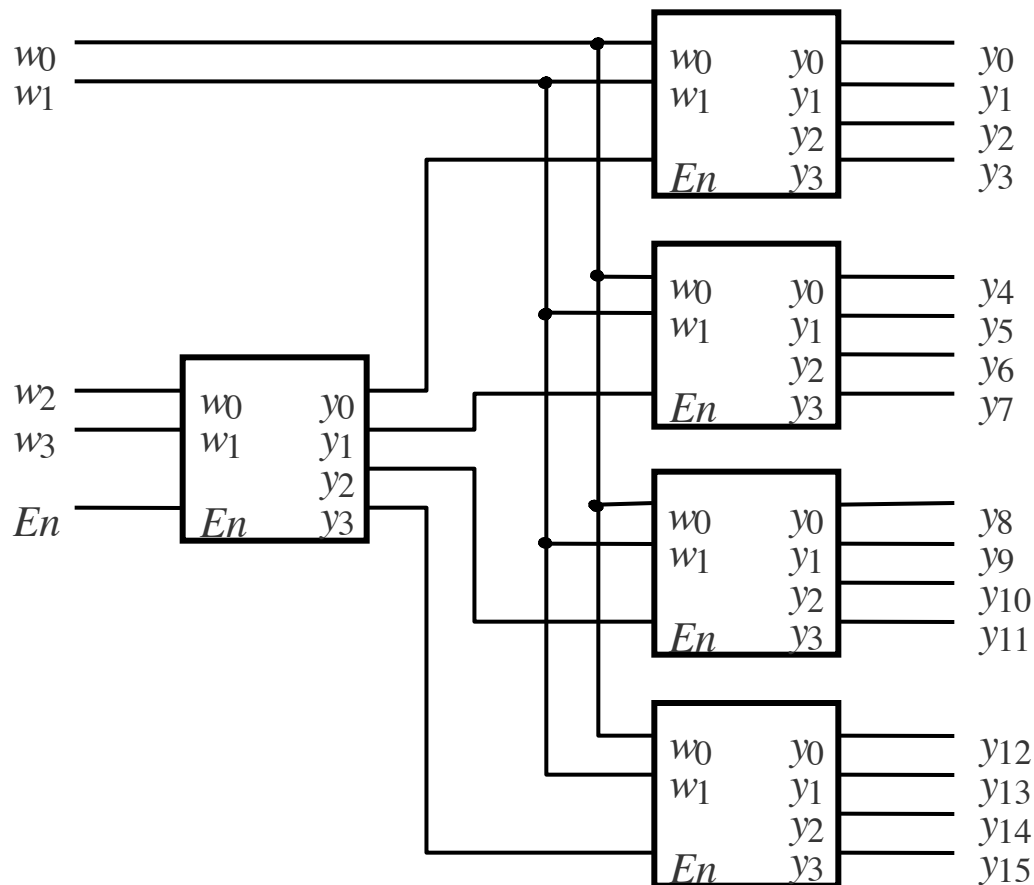


## Structural style Verilog

```
module mux16x1(f,W,S);  
    input [0:15] W;  
    input [3:0] S;  
    output f;  
    wire [0:3] M;  
  
    mux4x1 mux1(M[0],W[0:3],S[1:0]);  
    mux4x1 mux2(M[1],W[4:7],S[1:0]);  
    mux4x1 mux3(M[2],W[8:11],S[1:0]);  
    mux4x1 mux4(M[3],W[12:15],S[1:0]);  
    mux4x1 mux5(f,M,S[3:2]);  
endmodule
```

The Verilog code for mux4x1 must be either in the same file as mux16x1, or in a separate file (called mux4x1.v) in the same directory as mux16x1

# Hierarchical Design of a 4 to 16 Decoder



## Structural style Verilog

```
module dec4to16(Y,W,En);  
    input [3:0] W;  
    input En;  
    output [0:15] Y;  
    wire [0:3] M;  
  
    dec2to4(M,W[3:2],En);  
    dec2to4(Y[0:3],W[1:0],M[0]);  
    dec2to4(Y[4:7],W[1:0],M[1]);  
    dec2to4(Y[8:11],W[1:0],M[2]);  
    dec2to4(Y[12:15],W[1:0],M[3]);  
endmodule
```

dec2to4 to be presented soon ....

# Behavioral Style in Verilog

```
module mux2x1(f,W,s);  
    input [0:1] W;  
    input s;  
    output f;  
    reg f;
```

Must be reg type when used as LHS in an always block

```
    always @(W[0] or W[1] or s) // or just @(W or s)  
        if (s == 0)  
            f = W[0];  
        else  
            f = W[1];  
endmodule
```

Sensitivity list: statements inside the always block are only executed when one or more signals in the list changes value

# Always Blocks

- All LHS signals must be variables → type reg
  - The simulator *registers* the value and maintains it until the statements in the always block are executed again
  - Statements are re-executed only when a signal in the sensitivity list changes value
- The sensitivity list must include all signals on which the LHS variables depend in the always block
- Order counts inside the always block!!!!
- If ... else is called a *procedural statement*
  - All procedural statements must be inside an always block

# [ ■ Single vs Multiple Statements ]

- In Verilog, all constructs permit only a single statement
- If you need  $> 1$  statement inside a construct, use

begin

...

...

end

} You won't find much use for begin ... end →  
most constructs really do include just a single  
statement

- This is similar to Pascal syntax
  - Verilog is generally more like C though

# Representation of Numbers

- Numbers can be given as constants in

- Binary (b)
- Octal (o)
- Hex (h)
- Decimal (d)

- For numbers of a specified size:

- TFAE:

- `12' d2217`
- `12' h8A9`
- `12' o4251`
- `12' b100010101001`

Numbers are 0-extended to the left, if necessary, to fill out the number of bits

If the value exceeds the # of bits allocated, the extra bits are ignored!

 #bits given in decimal

## ■ 4x1 MUX: Behavioral Styles

```
module mux4x1(f,W,S);
    input [0:3] W;
    input [1:0] S;
    output f;
    reg f;

    always @(W or S)
        if (S == 0)
            f = W[0];
        else if (S == 1)
            f = W[1];
        else if (S == 2)
            f = W[2];
        else // if (S == 3)
            f = W[3];
endmodule|
```

```
module mux4x1(f,W,S);
    input [0:3] W;
    input [1:0] S;
    output f;
    reg f;

    always @(W or S)
        if (S == 2'b00)
            f = W[0];
        else if (S == 2'b01)
            f = W[1];
        else if (S == 2'b10)
            f = W[2];
        else // if (S == 2'b11)
            f = W[3];
endmodule|
```

# More on Representation of Numbers

- Negative numbers:

- $-4$  'b101  $\rightarrow$  the 4 bit 2's complement of 5  $\rightarrow$  1011

- For numbers of an unspecified size:

- 2217
- 'd2217
- 'h8A9
- 'o4251
- 'b100010101001

- The Verilog compiler chooses the size to fit with the other operands in the expression



# ■ Case Statement

```
module mux4x1(f,W,S);  
    input [0:3] W;  
    input [1:0] S;  
    output f;  
    reg f;  
  
    always @(W or S)  
        case (S)  
            0: f = W[0];  
            1: f = W[1];  
            2: f = W[2];  
            3: f = W[3];  
        endcase  
endmodule|
```

Comparisons in a case statement are made *bit by bit*.

No break statement needed – first match executes and then case is exited.

Use begin ... end if > 1 statement required in a case.

If not all cases are enumerated, make sure to use *default* case.

## ■ 2 to 4 Decoder: Behavioral Style

```
module dec2to4(Y,W,En);
    input [1:0] W;
    input En;
    output [3:0] Y;
    reg [3:0] Y;

    always @(W or En)
        case ({En, W}) // the concat of En with W
            3'b100: Y = 4'b0001;
            3'b101: Y = 4'b0010;
            3'b110: Y = 4'b0100;
            3'b111: Y = 4'b1000;
            default: Y = 4'b0000;
        endcase
endmodule
```

# [ ■ 4 to 2 Binary Encoder ]

```
module encoder(Y,W);  
    input [3:0] W; // input is one-hot  
    output [1:0] Y;  
    reg [1:0] Y;  
  
    always @(W)  
        case (W)  
            4'b1000: Y = 2'b11;  
            4'b0100: Y = 2'b10;  
            4'b0010: Y = 2'b01;  
            4'b0001: Y = 2'b00;  
            default: Y = 2'bx; // don't care case  
        endcase  
endmodule
```

Left extended by x to fill 2 bits

## ■ 4 to 2 Priority Encoder

```
module priority(Y,z,W);
    input [3:0] W;
    output [1:0] Y;
    output z;
    reg [1:0] Y;
    reg z;

    always @(W) begin
        z = 1; // assume valid output
        casex(W)
            4'b1xxx: Y = 3; // or 2'b11;
            4'b01xx: Y = 2; // or 2'b10;
            4'b001x: Y = 1; // or 2'b01;
            4'b0001: Y = 0; // or 2'b00;
            default: begin
                z = 0;
                Y = 2'bx; // don't care case
            end
        endcase
    end
endmodule
```

case vs caseX

## ■ Case, Casez, Casex

- Case treats each value 0, 1, x, and z literally
  - `4'b01xz` only matches `4'b01xz`
  - Example: `4'b0110` does not match `4'b01xx` in a case
- Casez treats 0, 1, and x literally
  - Casez treats z as a don't care
  - Example: `4'b0110` matches `4'b01zz`, but not `4'b01xz`  
No match here
- Casex treats 0 and 1 literally
  - Casex treats both x and z as don't cares
  - Example: `4'b0110` matches `4'b01xx` and also `4'b01xz`

## ■ BCD to 7 Segment Display Converter

```
module seg7(Leds,BCD);
    input [3:0] BCD;
    output [1:7] Leds;
    reg [1:7] Leds;

    always @(BCD)
        case (BCD) // abcdefg
            4'b0000 : Leds = 7'b1111110;
            4'b0001 : Leds = 7'b0110000;
            4'b0010 : Leds = 7'b1101101;
            4'b0011 : Leds = 7'b1111001;
            4'b0100 : Leds = 7'b0110011;
            4'b0101 : Leds = 7'b1011011;
            4'b0110 : Leds = 7'b1011111;
            4'b0111 : Leds = 7'b1110000;
            4'b1000 : Leds = 7'b1111111;
            4'b1001 : Leds = 7'b1111011;
            default : Leds = 7'bx;
        endcase
    endmodule
```

## ■ For Loop

- When a circuit exhibits regularity, a for loop can be used inside an always statement to simplify the design description (for loop is a procedural statement → only inside an always block)
- C style syntax: `for (k = 0; k < 4; k = k+1)`
  - Loop index must be type integer (not reg!)
  - Can't use the convenience of `k++`
  - Use `begin ... end` for multiple statements in the loop
- Each iteration of the loop specifies a *different piece of the circuit*
  - Has nothing to do with changes over “time”

## ■ 2 to 4 Decoder Using a For Loop

```
module dec2to4(Y,W,En);
    input [1:0] W;
    input En;
    output [3:0] Y;
    reg [3:0] Y;
    integer k;

    always @(W or En)
        for (k = 0; k < 4; k = k + 1)
            if ((W == k) && (En == 1))
                Y[k] = 1;
            else
                Y[k] = 0;
endmodule
```



## ■ 4 to 2 Priority Encoder Using a For Loop

```
module priority(Y,z,W);  
    input [3:0] W;  
    output [1:0] Y;  
    output z;  
    reg [1:0] Y;  
    reg z;  
    integer k;
```

```
    always @(W) begin  
        Y = 2'bxx;  
        z = 0;  
        for (k = 0; k < 4; k = k+1)  
            if (W[k]) begin // short for W[k] == 1  
                Y = k;  
                z = 1;  
            end  
    end
```

```
end  
endmodule
```

A signal that is assigned a value multiple times in an always block *retains its last value*  
→ priority scheme relies on this for correct setting of Y and z



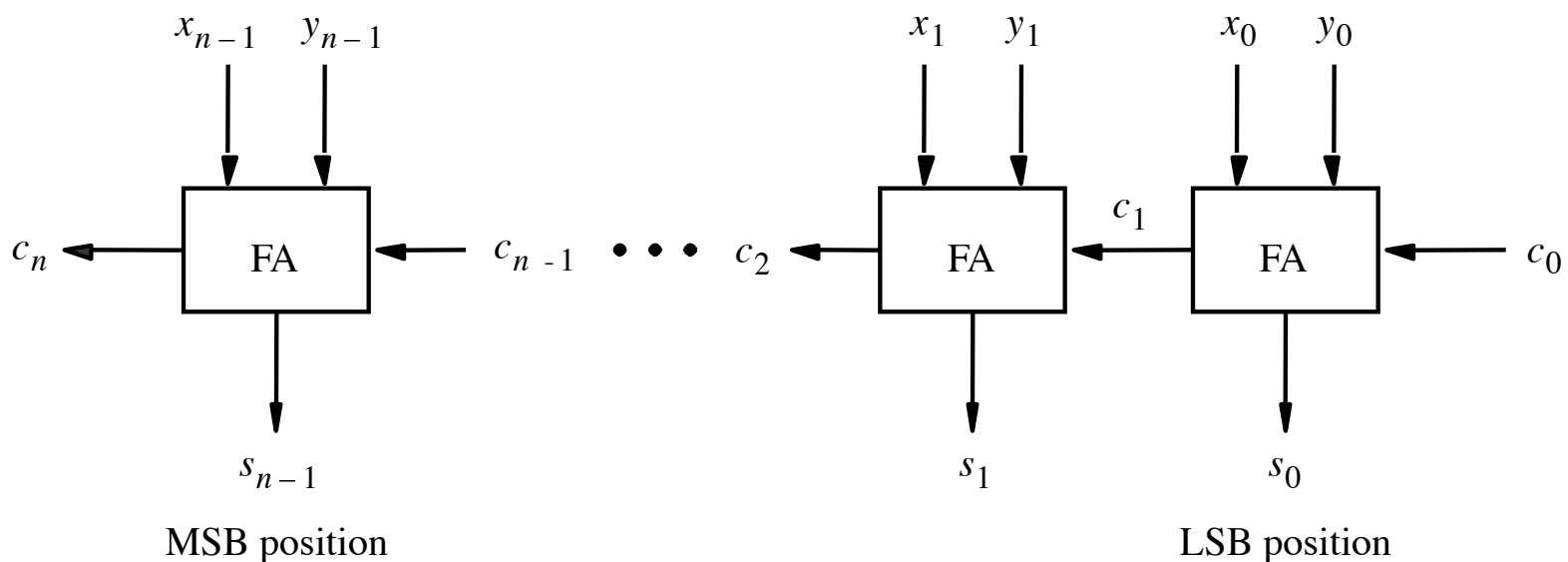
# Arithmetic Circuits

# Machine Arithmetic

- Arithmetic combinational circuits are required for
  - Addition
  - Subtraction
  - Multiplication
  - Division (hardware implementation is optional)
- Addition / subtraction can be done easily using full adders and a minimum of additional logic

# Ripple-Carry Adder

- A ripple carry adder cascades full adders together
  - Simple, but not the most efficient design
  - Carry propagate adders are more efficient



# Behavioral Style Full Adder

- We've done full adders many ways already

```
module fulladder(s,cout,x,y,cin);  
  input x, y, cin;  
  output s, cout;
```

```
  assign {cout, s} = x + y + cin;  
endmodule|
```

```
module fulladder(s,cout,x,y,cin);  
  input x, y, cin;  
  output s, cout;
```

```
  assign s = x ^ y ^ cin;  
  assign cout = (x & y) | (x & cin) | (y & cin);  
endmodule|
```

data flow styles

behavioral style

```
module fulladder(s,cout,x,y,cin);  
  input x, y, cin;  
  output s, cout;  
  reg s, cout;
```

```
  always @(x or y or cin)  
    {cout, s} = x + y + cin;  
endmodule|
```

# Behavioral Style Ripple-Carry Adder

```
module adder(S,cout,X,Y,cin);
```

```
    parameter n = 32; ← Allows a generic size to be specified
```

```
    input [n-1:0] X, Y;
```

```
    input cin;
```

```
    output [n-1:0] S;
```

```
    output cout;
```

```
    reg [n-1:0] S;
```

```
    reg cout;
```

```
    reg [n:0] C;
```

```
    integer k;
```

For  $n = 2$ , the for loop is equivalent to:

$S[0] = X[0] \wedge Y[0] \wedge C[0];$

$C[1] = (X[0] \& Y[0]) \mid (X[0] \& C[0]) \mid (Y[0] \& C[0]);$

$S[1] = X[1] \wedge Y[1] \wedge C[1];$

$C[2] = (X[1] \& Y[1]) \mid (X[1] \& C[1]) \mid (Y[1] \& C[1]);$

```
    always @(X or Y or cin) begin
```

```
        C[0] = cin;
```

```
        for (k = 0; k < n; k = k + 1) begin
```

```
            S[k] = X[k] ^ Y[k] ^ C[k];
```

```
            C[k+1] = (X[k] & Y[k]) | (X[k] & C[k]) | (Y[k] & C[k]);
```

```
        end
```

```
        cout = C[n];
```

```
    end
```

```
endmodule
```

## ■ Higher Level Behavioral Style for Adder

- This won't work! Can't loop over “subcircuits”

```
module adder(S,cout,X,Y,cin);
    parameter n = 32;
    input [n-1:0] X, Y;
    input cin;
    output [n-1:0] S;
    output cout;
    reg [n-1:0] S;
    reg cout;
    reg [n:0] C;
    integer k;
```

```
module fulladder(s,cout,x,y,cin);
    input x, y, cin;
    output s, cout;

    assign {cout, s} = x + y + cin;
endmodule
```

```
fulladder(S[0],C[1],X[0],Y[0],cin);
always @(X or Y or cin)
    for (k = 1; k < n-1; k = k + 1)
        fulladder(S[k],C[k+1],X[k],Y[k],C[k]);
fulladder(S[n-1],cout,X[n-1],Y[n-1],C[n-1]);
endmodule
```

subcircuits to be instantiated like this  
are not allowed in a for loop

## ■ Functions / Tasks

- Subcircuits can't directly be instantiated in for loops
  - Can't create a separate module for fulladder and then instantiate that inside a for loop
- Need to create a *function* or a *task* for the subcircuit
- Functions and tasks provide modular code without defining separate modules
  - Defined within a module
  - Code is placed in-line by the Verilog compiler
- Functions and tasks are behavioral only



# [ ■ Functions ]

---

- Functions provide modular code without defining separate modules
  - Defined within a module
  - Can have many inputs (must have  $> 0$ ), but only one output
    - Function is *called* like C++ functions that have a non-void return value
  - Functions can be called in a continuous assignment or in a procedural statement
    - Functions contain only procedural statements
  - Function code is placed in-line by the Verilog compiler

## ■ More on Functions

- Functions must have  $> 0$  inputs
  - Order of inputs is dictated by the order in which they are declared in the function
- Functions can call other functions, but not tasks
- May return a vectored signal by declaring the function as:

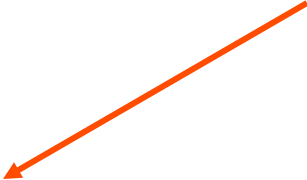
```
function [3:0] foo;    // the range indicates a 4 bit result
...
endfunction
```

## ■ Function Example: 16x1 MUX

```
module mux16x1(f,W,S);  
    input [0:15] W;  
    input [3:0] S;  
    output f;  
    reg f;  
    reg [0:3] M;
```

```
    function mux4x1;  
        input [0:3] W;  
        input [1:0] S;  
        mux4x1 = S[1] ? (S[0] ? W[3] : W[2]) : (S[0] ? W[1] : W[0]);  
    endfunction
```

begin ... end required if more than 1  
procedural statement in a function



```
    always @(W or S) begin // structural looking style in a procedural block  
        M[0] = mux4x1(W[0:3],S[1:0]);  
        M[1] = mux4x1(W[4:7],S[1:0]);  
        M[2] = mux4x1(W[8:11],S[1:0]);  
        M[3] = mux4x1(W[12:15],S[1:0]);  
        f = mux4x1(M,S[3:2]);  
    end  
endmodule
```

# [ ■ Tasks ]

---

- Tasks also provide modular code without defining separate modules
  - Also defined within a module
  - Can have many inputs and outputs
    - Task is *called* like C++ functions that have a void return type
    - Outputs are returned via the output variables (like the ports in a module)
  - Task can only be called in a procedural statement
    - Tasks contain only procedural statements
  - Task code is placed in-line by the Verilog compiler

## ■ More on Tasks

- Tasks may have any number of inputs and outputs
  - Order of inputs and outputs is dictated by the order in which they are declared in the task
- Tasks can call other tasks or functions
- All arguments to a task are implicitly of type reg
- begin ... end block required in a task if you use > 1 procedural statement

## ■ Ripple-Carry Adder Using a Task

```
module adder(S,cout,X,Y,cin);
    parameter n = 32;
    input [n-1:0] X, Y;
    input cin;
    output [n-1:0] S;
    output cout;
    reg [n-1:0] S;
    reg cout;
    reg [n:0] C;
    integer k;

    task addbit;
        output S, cout;
        input X, Y, cin;

        {cout, S} = X + Y + cin;
    endtask

    always @(X or Y or cin) begin
        C[0] = cin;
        for (k = 0; k < n; k = k + 1)
            addbit(S[k],C[k+1],X[k],Y[k],C[k]);
        cout = C[n];
    end
endmodule
```

# [ ■ Generate Statement ]

---

- Verilog 2001 provides a new statement for instantiating separate modules inside a for loop
  - Permits structural style to use for loops
  - `generate ... endgenerate`
  - Use `genvar` datatype in place in integer

# Ripple-Carry Adder Using Generate

```
module adder(S,cout,X,Y,cin);  
    parameter n = 32;  
    input [n-1:0] X, Y;  
    input cin;  
    output [n-1:0] S;  
    output cout;  
    wire [n:0] C;  
    genvar k;
```

```
    assign C[0] = cin;  
    assign cout = C[n];
```

```
    generate
```

```
        for (k = 0; k < n; k = k + 1)
```

```
        begin: addstage
```

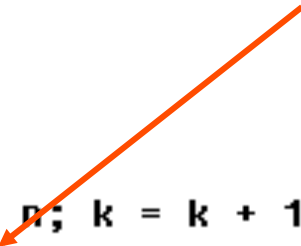
```
            fulladder addbit(S[k],C[k+1],X[k],Y[k],C[k]);
```

```
        end
```

```
    endgenerate
```

```
endmodule
```

compiler produces n modules with names  
addstage[0].addbit, addstage[1].addbit, ...,  
addstage[n-1].addbit





# ■ Macrofunctions

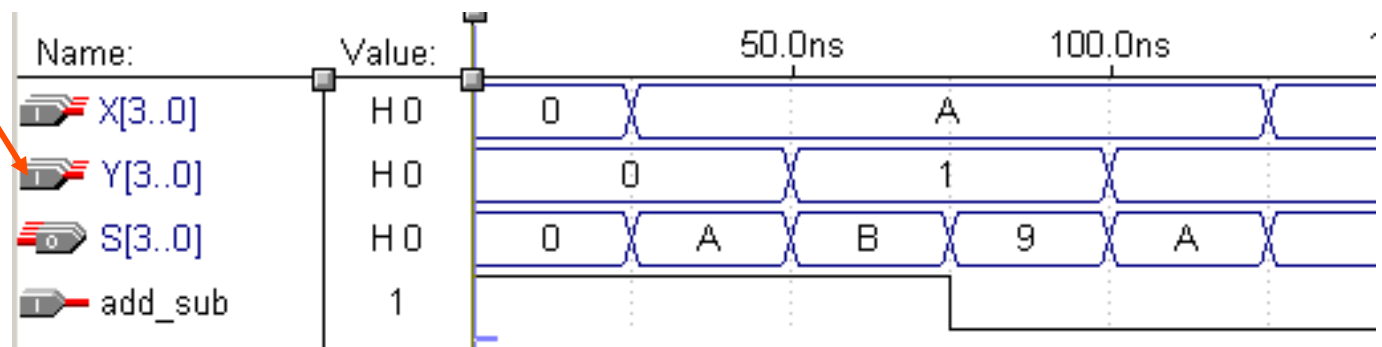
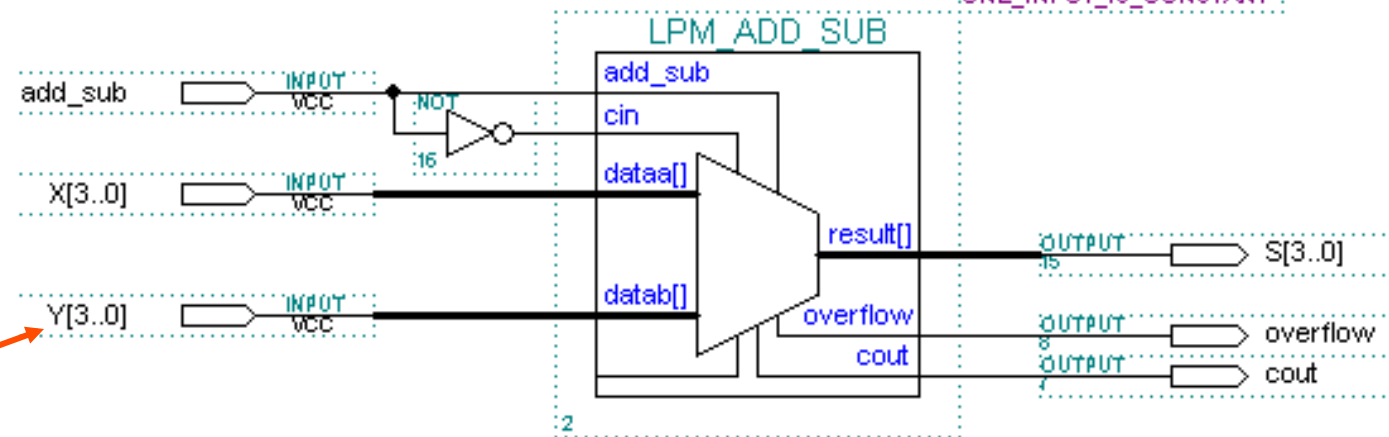
- Libraries of common circuits, such as adders, are available in most commercial CAD tools
  - Sometimes called macrofunctions or megafunctions
- Example: Quartus II provides a Library of Parametrized Modules (LPM)
  - Each module is parametrized, e.g. the user can set the number bits used in the module
  - LPM\_ADD\_SUB is an n-bit adder / subtractor where you can pick the n it will use
    - LPM\_WIDTH
    - Available in the “megafunctions arithmetic” library

# LPM\_ADD\_SUB Module

- $\text{add\_sub} = 1 \rightarrow \text{dataa} + \text{datab} + \text{cin}$
- $\text{add\_sub} = 0 \rightarrow \text{dataa} - \text{datab} + (\text{cin}-1)$

LPM\_DIRECTION=  
 LPM\_PIPELINE=  
 LPM\_REPRESENTATION=  
 LPM\_WIDTH=4  
 MAXIMIZE\_SPEED=  
 ONE\_INPUT\_IS\_CONSTANT=

Multi-bit signals must be named with [MSB..LSB]



## ■ Sample Application: BCD Adder

- Build a 4 bit BCD adder with carry-out

- Examples:

X= 0010  
Y= 0110  
Binary Sum= 1000

BCD Sum= 1000

Cout= 0

X= 1000  
Y= 1001  
Binary Sum= 0001

BCD Sum= 0111

Cout= 1

X= 0110  
Y= 0101  
Binary Sum= 1011

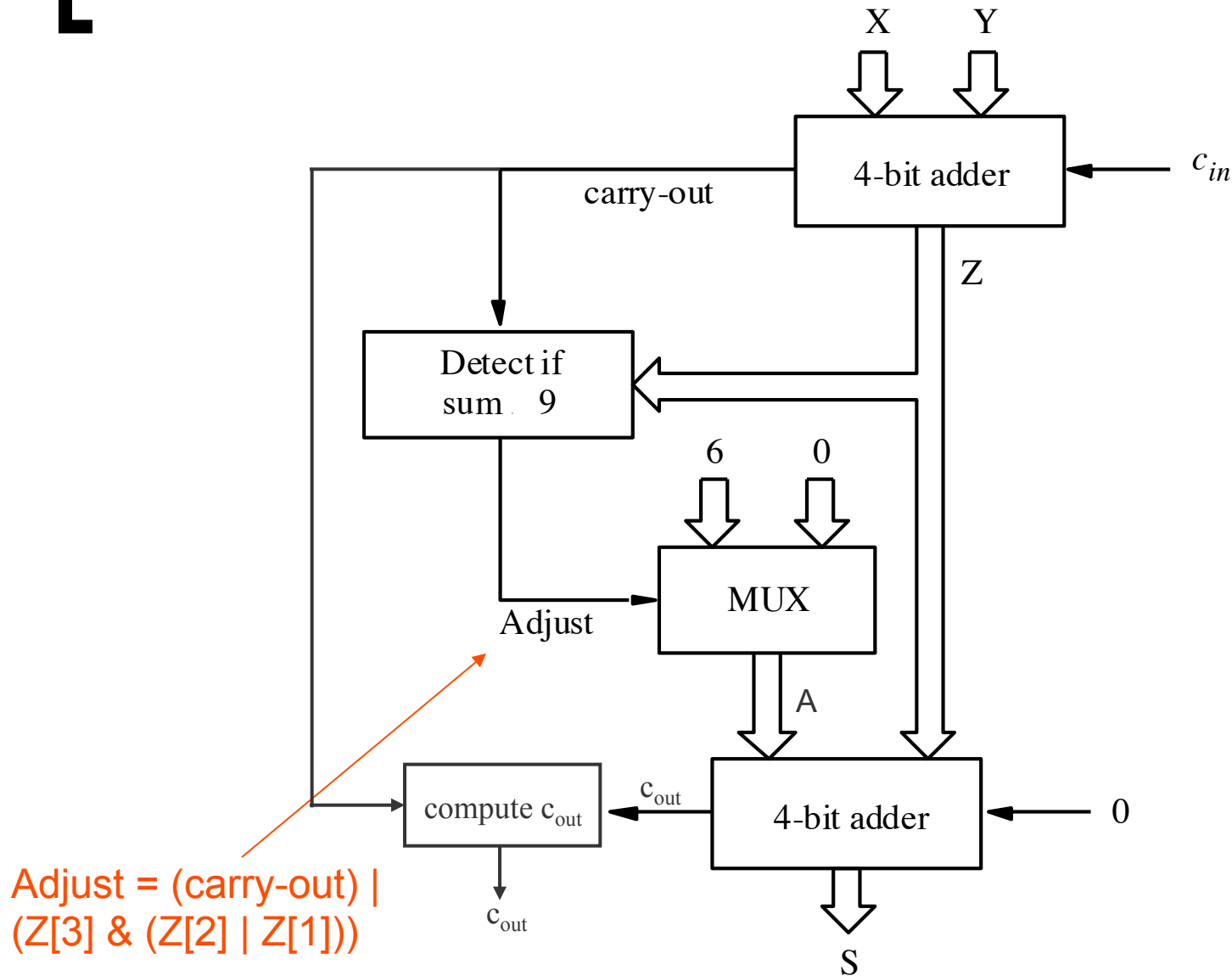
BCD Sum= 0001

Cout= 1

- What's the rule???

- If the binary sum exceeds 9 (including the carry out), then add an additional 6 to it to get the BCD sum
- cout = 1 if cout from binary add, or if cout from +6 add

## Block Diagram for BCD Adder



# Structural Verilog for BCD Adder

```
module BCDadder(S,cout,X,Y,cin);
    input [3:0] X, Y;
    input cin;
    output [3:0] S;
    output cout;
    wire [3:0] Z, A;
    wire adjust, adder1_cout, adder2_cout;

    lpm_add_sub adder1(.result(Z),.cout(adder1_cout),.dataa(X),.datab(Y),.cin(cin));
    defparam adder1.LPM_WIDTH = 4;

    lpm_add_sub adder2(.result(S),.cout(adder2_cout),.dataa(A),.datab(Z),.cin(0));
    defparam adder2.LPM_WIDTH = 4;

    assign adjust = (adder1_cout) | (Z[3] & (Z[2] | Z[1])); // the adjust logic
    assign A = (adjust) ? (4'b0110) : (4'b0000); // the 2x4 MUX
    assign cout = adder1_cout | adder2_cout;
endmodule
```

Using positional notation for arguments  
Note: add\_sub = 1 is the default and so need not be specified

Specifying the size of the adders using defparam

## ■ Behavioral Verilog for BCD Adder

```
module BCDadder(S,cout,X,Y,cin);
    input [3:0] X, Y;
    input cin;
    output [3:0] S;
    output cout;
    reg [3:0] S;
    reg cout;
    reg [4:0] Z;

    always @(X or Y or cin) begin
        Z = X + Y + cin;
        if (Z > 9)
            {cout,S} = Z + 6;
        else
            {cout,S} = Z;
        end
    end
endmodule
```