Verilog – a hardware description language (HDL)

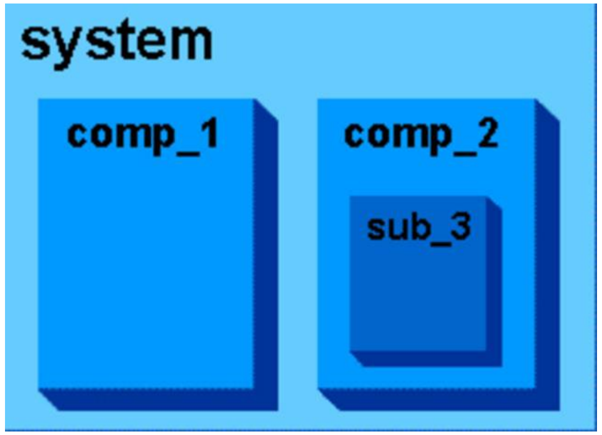
Two ways to describe a circuit in Verilog:

1. Structural–using circuit elements eg. logic gates.
2. Behavioural –using logic expressions & programming constructs

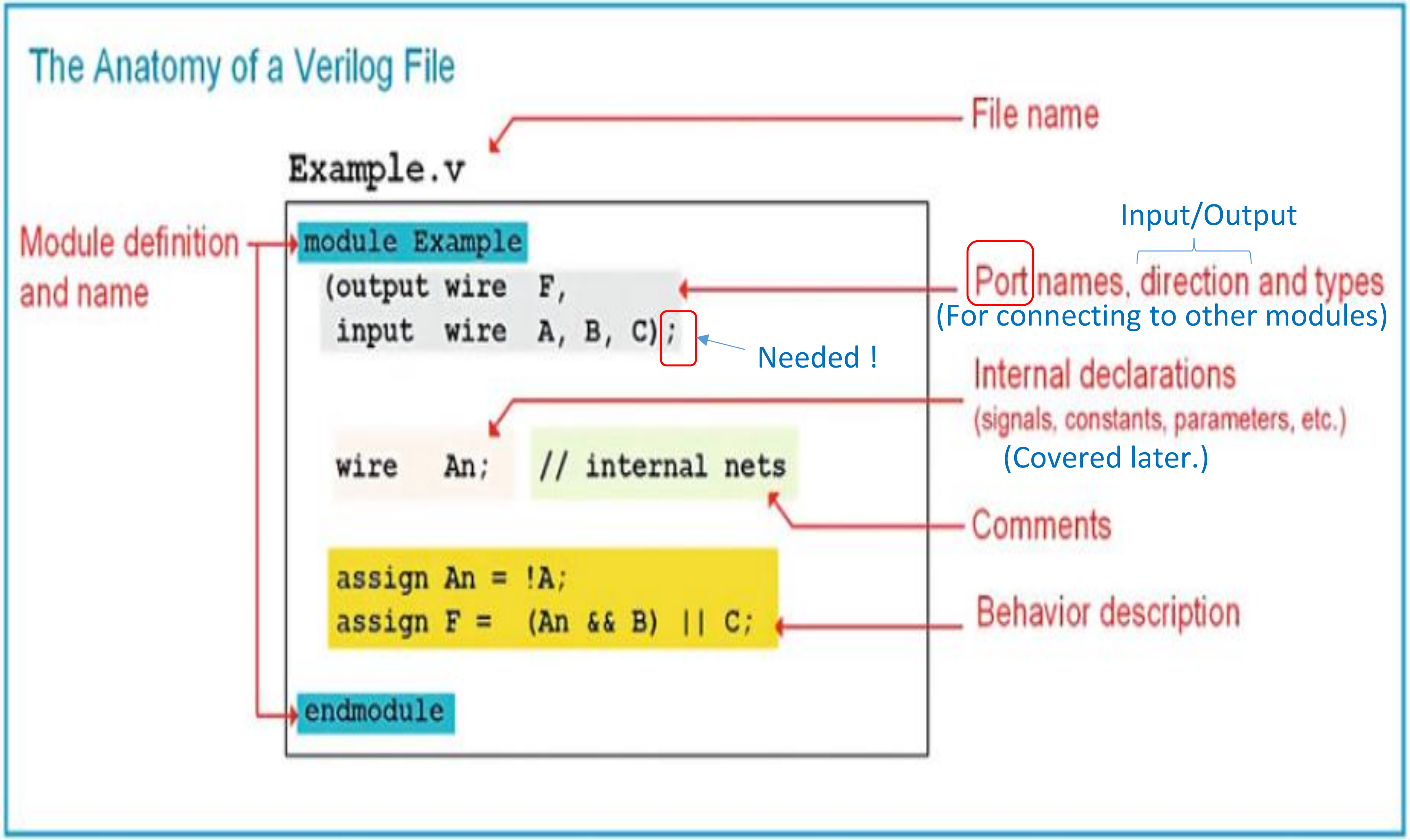
A Verilog **model** (project) is composed of one or more **modules**.
Example:

- Module "system" is the parent of "comp_1" and "comp_2".
- Module "comp_2" is the parent of "sub_3".

Ch.2
p.2



(From: “Quick Start Guide to Verilog” p.18)

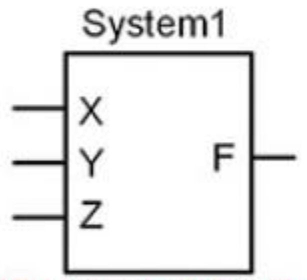

The Anatomy of a Verilog File
File name
Example.v
Module definition and name
Input/Output
Port names, direction and types
(For connecting to other modules)
Needed !
Internal declarations
(signals, constants, parameters, etc.)
(Covered later.)
Comments
Behavior description

```
module Example
  (output wire  F,
   input  wire  A, B, C);

   wire   An;   // internal nets

   assign An =  !A;
   assign F =  (An && B) || C;

endmodule
```

(From: “Quick Start Guide to Verilog” p.19)

Example: Verilog Port Declarations

Our notes uses this approach



All ports are type wire

or

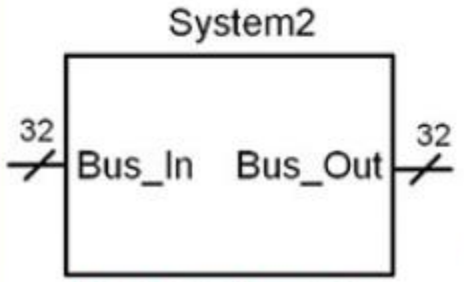
Pre Verilog-2001 Approach: Port names are listed after the module name with directions and types listed separately within the module.

```
module System1 (F, X, Y, Z);

   output  F;           // Port directions
   input   X, Y, Z;

   wire    F;           // Port types
   wire    X, Y, Z;

  //-- module items go here...

endmodule
```

Post Verilog-2001 Approach: Port names, directions, and types are listed after the module name.

```
module System1 (output wire F,
                 input  wire X, Y, Z);

  //-- module items go here...

endmodule
```

(Scalar – 1-bit)



Inputs are type wire,
outputs are type reg

Post Verilog-2001 Approach:

```
module System2 (output reg  Bus_Out[31:0],
                 input  wire Bus_In[31:0]);

  //-- module items go here...

endmodule
```

Array

General syntax for module and port

Ch.2
p.3

form

module module_name [(port_name{, portname })]; Don't forget !

Means can optionally have more like this.

[**parameter** declarations]

Direction

[**input** declarations]

[**output** declarations]

[**inout** declarations]

Type

[**wire** or **tri** declarations]

[**reg** or **integer** declarations]

[Means optional]

(Will learn these keywords later.)

Execution

[**assign** continuous assignments] For simple combinational logic

[**initial** block] Used in test-bench module only

[**always** blocks] [gate instantiations] [module instantiations]

For more complex combinational logic **or** sequential logic

endmodule

No need to have "**;**" here!

Verilog Signals:

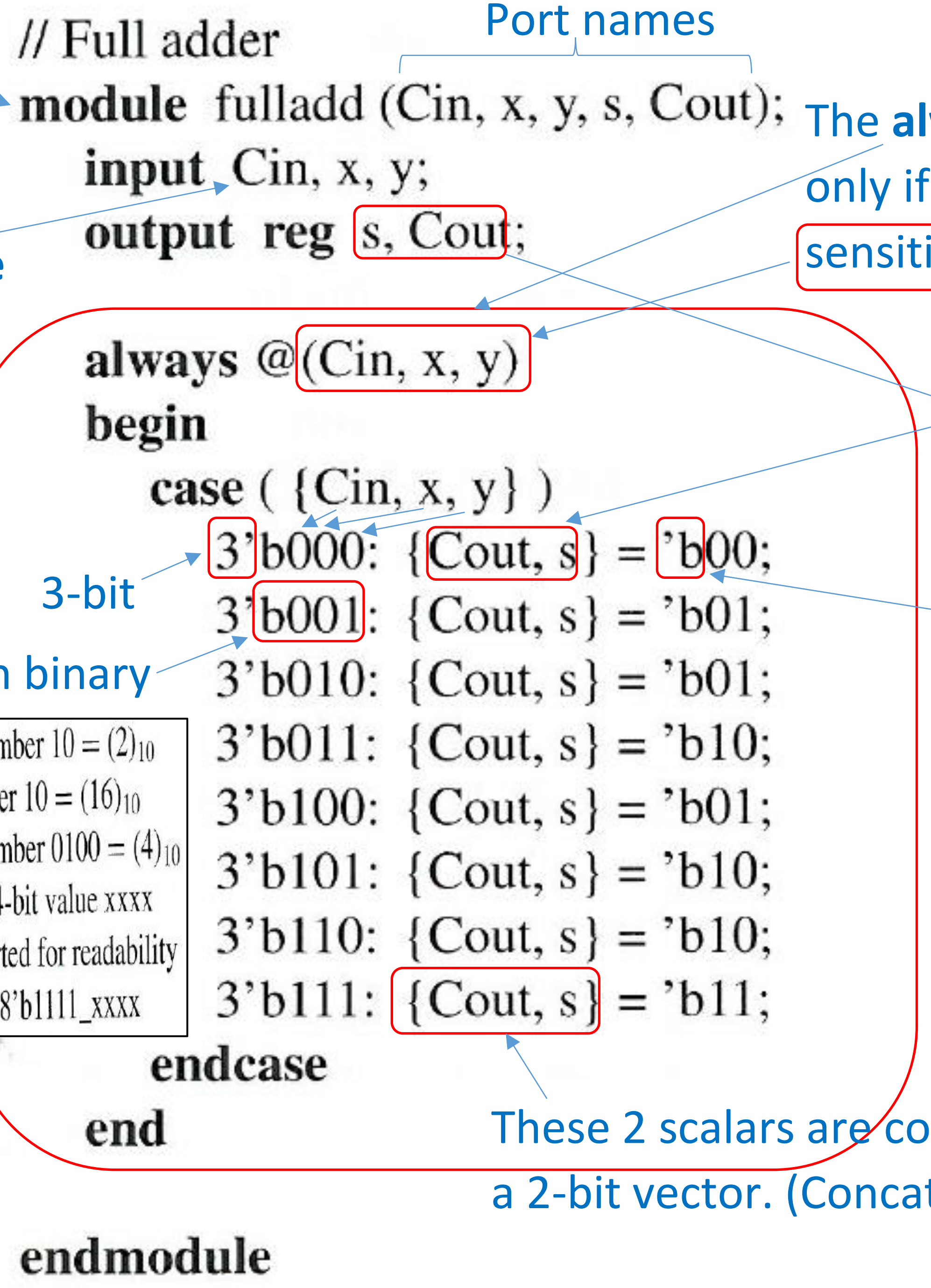
1. Net – represents a circuit node, for combinational logic circuits <u>only</u>.
   - (a) **wire** - a simple connection between components.
   - (b) **tri** - as **wire** but for a net driven by multiple sources.
2. Variable
   - (a) **reg** - models logic storage. Sequential logic circuits <u>must</u> use it.
   - (b) **integer** – 32-bit signed number, usually for loop control.

Verilog Signal values:

- 0 - logic zero (false)
- 1 - logic one (true)
- X or x - unknown or don't care value
- Z or z - high-impedance value

A **scalar** is 1-bit; while **vector** is multi-bit e.g. **wire** [31:0] bus;
A **parameter** is for representing a constant value, usually to be used multiple times in the file. For example: **parameter** BUS_WIDTH = 64;

Example: Ch.2 p.4



```
// Full adder
module  fulladd (Cin, x, y, s, Cout);
   input  Cin, x, y;
   output  reg  s, Cout;

   always @(Cin, x, y)
   begin
      case ( {Cin, x, y} )
         3'b000:  {Cout, s} = 'b00;
         3'b001:  {Cout, s} = 'b01;
         3'b010:  {Cout, s} = 'b01;
         3'b011:  {Cout, s} = 'b10;
         3'b100:  {Cout, s} = 'b01;
         3'b101:  {Cout, s} = 'b10;
         3'b110:  {Cout, s} = 'b10;
         3'b111:  {Cout, s} = 'b11;
      endcase
   end

endmodule
```

Port names

The **always** block will be executed only if any variable in its sensitivity list has been changed.

Default to **wire** if type is not specified.

reg can be assiged value in **always** block, but not for **wire**.

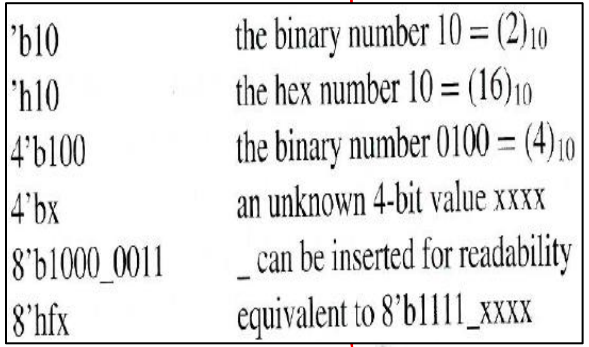
3-bit

In binary

If not specified, the compiler will decide how many bits from the data

These 2 scalars are combined to form a 2-bit vector. (Concatentation.)

Other examples on p.5:



'b10	the binary number $10 = (2)_{10}$
'h10	the hex number $10 = (16)_{10}$
4'b100	the binary number $0100 = (4)_{10}$
4'bx	an unknown 4-bit value xxxx
8'b1000_0011	_ can be inserted for readability
8'hfx	equivalent to 8'b1111_xxxx

The above example implements the Full Adder's truth-table using a **case** structure in an **always** block. (The case structure must be inside an **always** block.)

The following example shows how the Full Adder can also be implemented from its boolean equations through two **assign** statements.

```
module fulladd (Cin, x, y, s, Cout); // Pre-2001 style
    input Cin, x, y; // They are type wire by default
    output s, Cout; // Note that they are also type wire in this example

    assign s = x ^ y ^ Cin; // ^ is XOR
    assign Cout = (x & y) | (y & Cin) | (x & Cin); // & is AND, | is OR
endmodule
```

Notes:

1. Only **wire** can be assigned a value (on the LHS) in an **assign** statement.
2. Only **reg** can be assigned a value (on the LHS) within an **always** block.
3. The **assign** statements are considered to be in parallel, their sequence is not important.

The Full Adder can also be implemented by connecting logic gates together as in Fig 2-11. This is said to be in *structural* description; while the above two examples are in *behaviour* description.

Verilog Operators (Notes Ch.2 p.8 and “Quick Start Guide to Verilog” p.23)

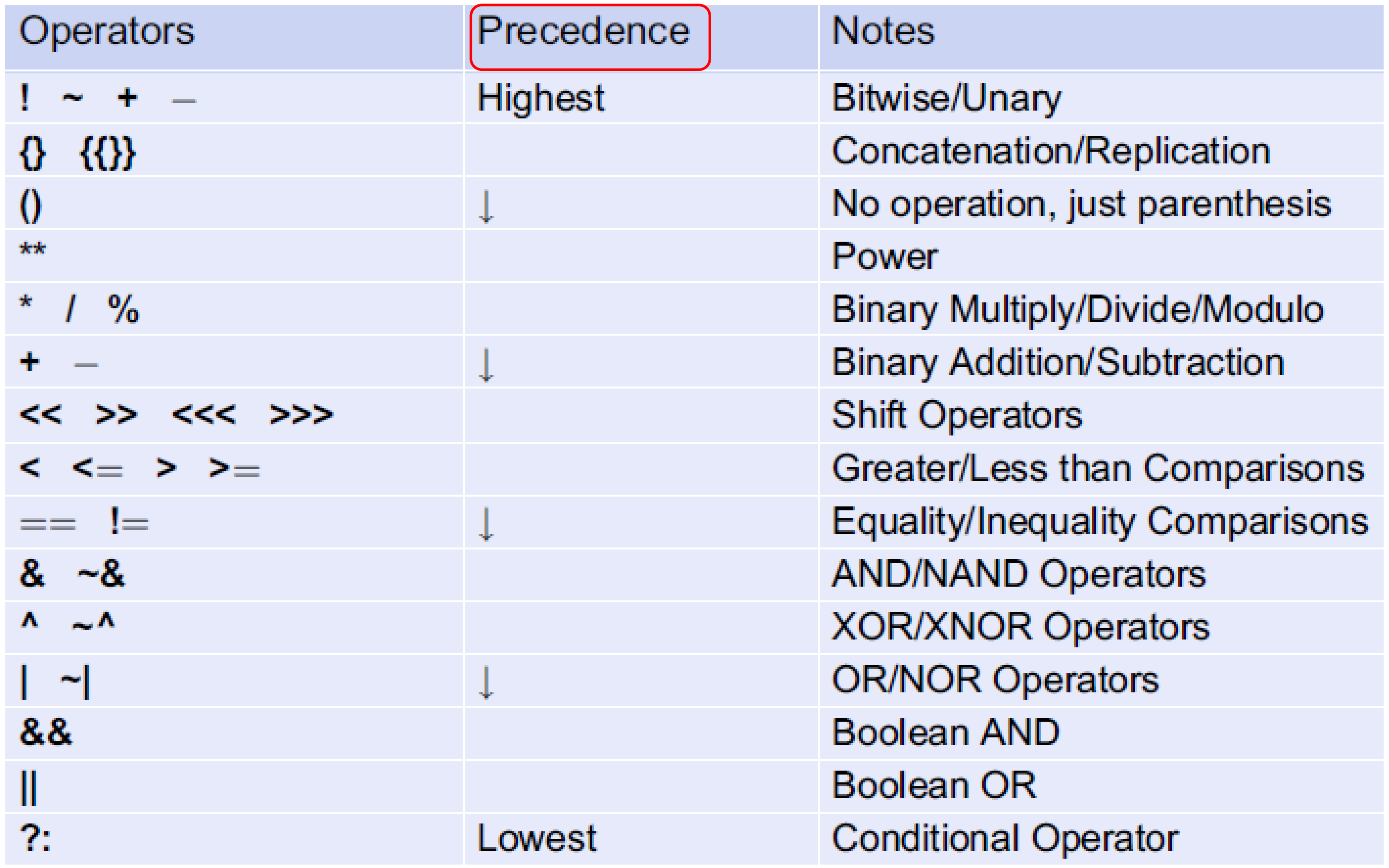
1. **Bitwise** – result length is same as the longest operand.
   - Z = ~X; // invert each bit in X.

     OR XOR XNOR
   - Z = X & Y; // AND each bit of X with each bit of Y. (Others: |, ^, ~^)
2. **Reduction** – result is 1-bit long.

     NAND NOR
   - bit = &X; // AND all bits in vector X together. (Others: |, ~&, ~|, ^, ~^)
3. **Logical & Relational** – result is 1-bit: 1 for true, 0 for false.
   - If(!X)… // If X is true (non-0), result=0 (false). If X is false (0), result=1.
   - if (X == Y)… // If X is equal to Y, result=1 (true), else result=0 (false).
   - if (X != Y)… // not equal. (Others: >, <, >=, <=)
4. **Arithmetic (Numerical)** – result length is same as the longest operand.
   - Z = A % B; // Modulo, ie. remainder. (Others: +, -, *, /, **)
5. **Shift** – result length is same as the operand.
   - Z = X << 4; // Shift X left 4 times and fill empty LSB location with 0s.
   - Z = X >> 4; // Shift X right 4 times and fill empty MSB location with 0s.

6. **Concatenation** – result length is the sum of all operands’ lengths.
- Z = {A, B, C} ; // If A is 2’b11, B is 1’b0, C is 1’b1 then Z will be 4’b1101.

7. **Conditional**
- Z = (A) ? 10 : 20; // If A is true (non-0) then Z = 10, else Z = 20.



Operators	Precedence	Notes
! ~ + –	Highest	Bitwise/Unary
{} {{}}		Concatenation/Replication
()	↓	No operation, just parenthesis
**		Power
* / %		Binary Multiply/Divide/Modulo
+ –	↓	Binary Addition/Subtraction
<< >> <<< >>>		Shift Operators
< <= > >=		Greater/Less than Comparisons
== !=	↓	Equality/Inequality Comparisons
& ~&		AND/NAND Operators
^ ~^		XOR/XNOR Operators
| ~|	↓	OR/NOR Operators
&&		Boolean AND
||		Boolean OR
?:	Lowest	Conditional Operator

From: “Quick Start Guide to Verilog” p.28

See also notes Ch.2, p.9

Concurrent statements (Notes Ch.2 p.7-9)

- In any HDL (hardware description language) including Verilog, the statements represent different parts of the circuit and the signals in them may change concurrently (i.e. at the same time).
- Concurrent statements are considered in parallel and the sequence of statements in the code does not matter.
- In Verilog, concurrency can be achieved by:
  - **Continuous assignment** – with keyword **assign**.
    - Can only assign to net type such as **wire**.
    - Models <u>combinational logic</u>.
    - Recommended for <u>simple circuits</u>.

<u>Example:</u>

```
        wire a, b, c;
        reg x, y;
        assign a = b & c; // OK, a is of type wire.
        assign b = a & x; // OK, b is of type wire.
        assign y = a & b; // Error, as y is of type reg.
```

*The order of these two **continuous assign** statements is not important.*

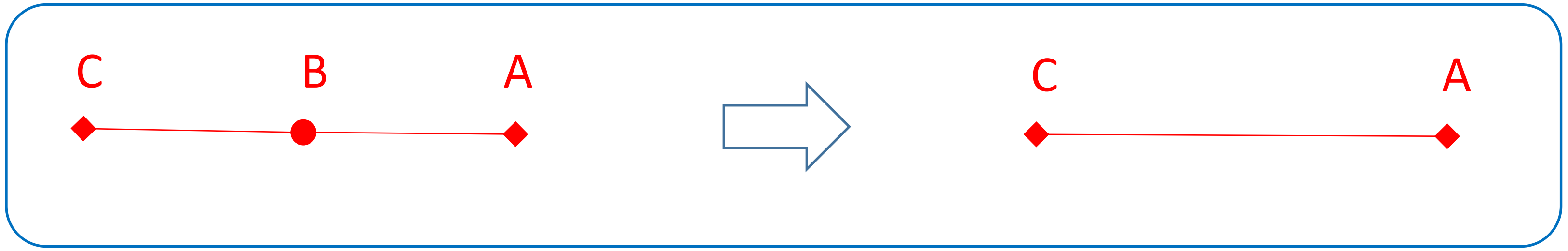
Example: Concurrent statements using continuous assignment:

```
wire A, B, C;

assign A = B;
assign B = C;
```

Showing the input/output waveforms.

Generating the circuit diagram.

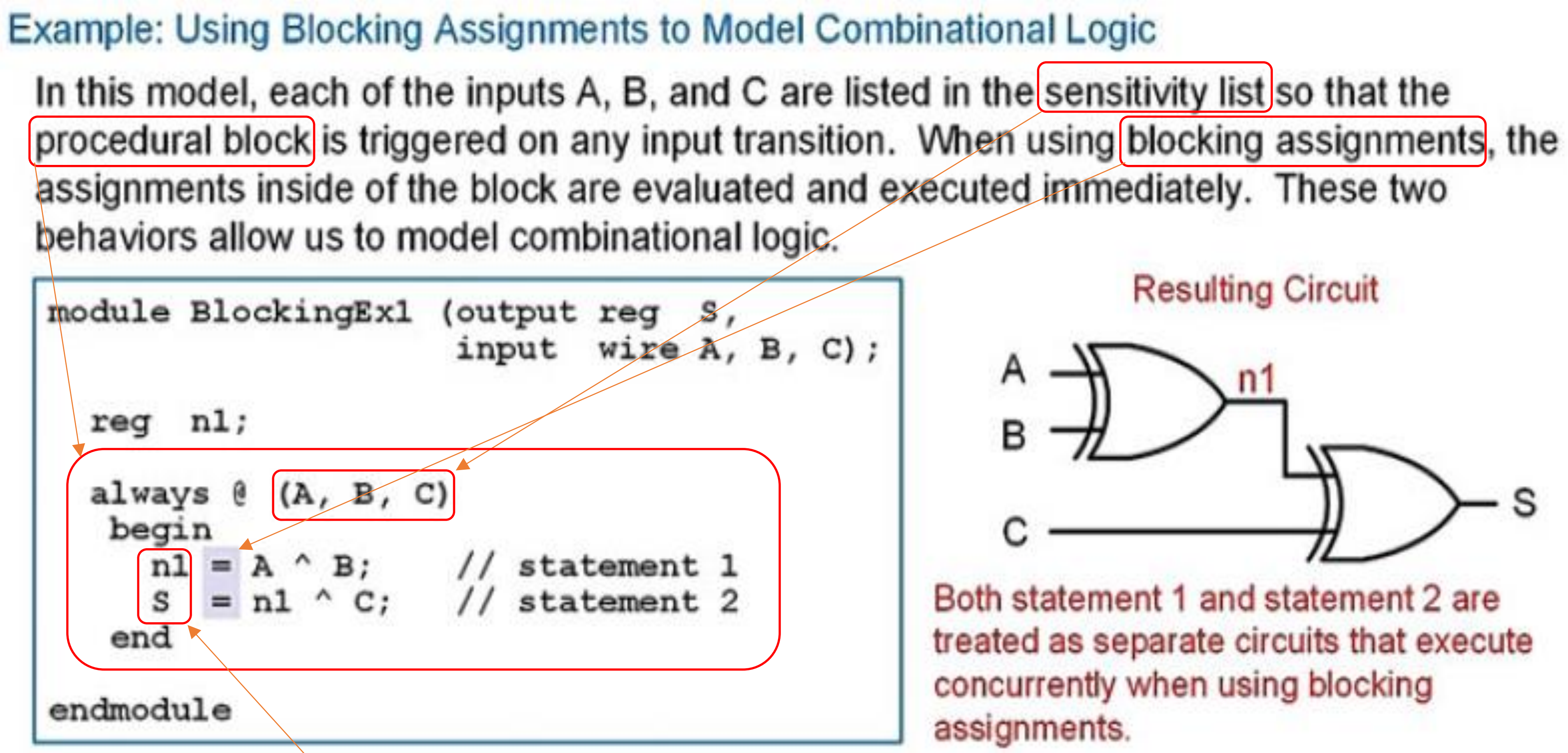
- In Verilog simulation, signal assignments of C to B, and B to A will take place at the same time (i.e. concurrently).
- During synthesis, signal B will be eliminated since it describes two wires in series.
- Automated synthesis tools will eliminate unnecessary signals.
- Different results from a sequentially executed computer program (where A will hold the original value of B, while the new value of B is copied from C).



Procedural assignments (Notes Ch.2 p.9-11)

- Must be enclosed within procedural blocks:
  - **initial** block - execute one time at the beginning of the simulation.
  - **always** block - execute throughout the duration of the simulation.
  - Executed statements in the order they are listed in the code.
  - Can use programming constructs (while **assign** cannot).
  - Continuous assignments (**assign**) not allowed in these blocks.
- Can only assign to variable types (e.g. **reg**).
- Blocking assignment e.g. S **=** X + Y;
  - Suitable for modeling <u>combinational</u> logic circuits.
  - Assign values to LHS immediately.
- Non-blocking assignment e.g. S **<=** X + Y;
  - Suitable for modeling <u>sequential</u> logic circuits.
  - Assign values to LHS only at the end of the **initial** or **always** block.

From: “Quick Start Guide to Verilog” p.78



Example: Using Blocking Assignments to Model Combinational Logic

In this model, each of the inputs A, B, and C are listed in the sensitivity list so that the procedural block is triggered on any input transition. When using blocking assignments, the assignments inside of the block are evaluated and executed immediately. These two behaviors allow us to model combinational logic.

```
module BlockingEx1 (output reg  S,
                    input  wire A, B, C);

  reg  n1;

  always @ (A, B, C)
   begin
     n1 = A ^ B;      // statement 1
     S  = n1 ^ C;     // statement 2
   end

endmodule
```

Resulting Circuit

Both statement 1 and statement 2 are treated as separate circuits that execute concurrently when using blocking assignments.

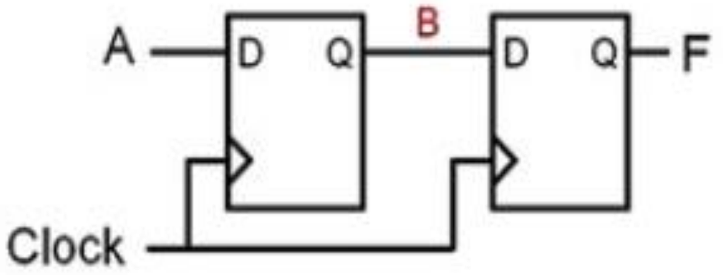
LHS of statements in an **initial**/**always** block must be variable type (e.g. **reg**).

Example: Using Non-Blocking Assignments to Model Sequential Logic

In this model, the always block will only trigger on the rising edge of a clock. When using non-blocking assignments, the assignments inside of the block are only executed at the end of the block. These two behaviors allow us to model sequential logic.

```
module NonBlockingEx1 (output reg   F,
                       input  wire A,
                       input  wire Clock);
  reg  B;

  always @ (posedge Clock)
    begin
      B <= A;    // statement 1
      F <= B;    // statement 2
    end

endmodule
```

Resulting Circuit



Notice that the value of B in statement 2 is not immediately updated with the assignment made in statement 1 due to the nature of non-blocking assignments.

LHS of statements in an **initial**/**always** block must be of variable type (e.g. **reg**).

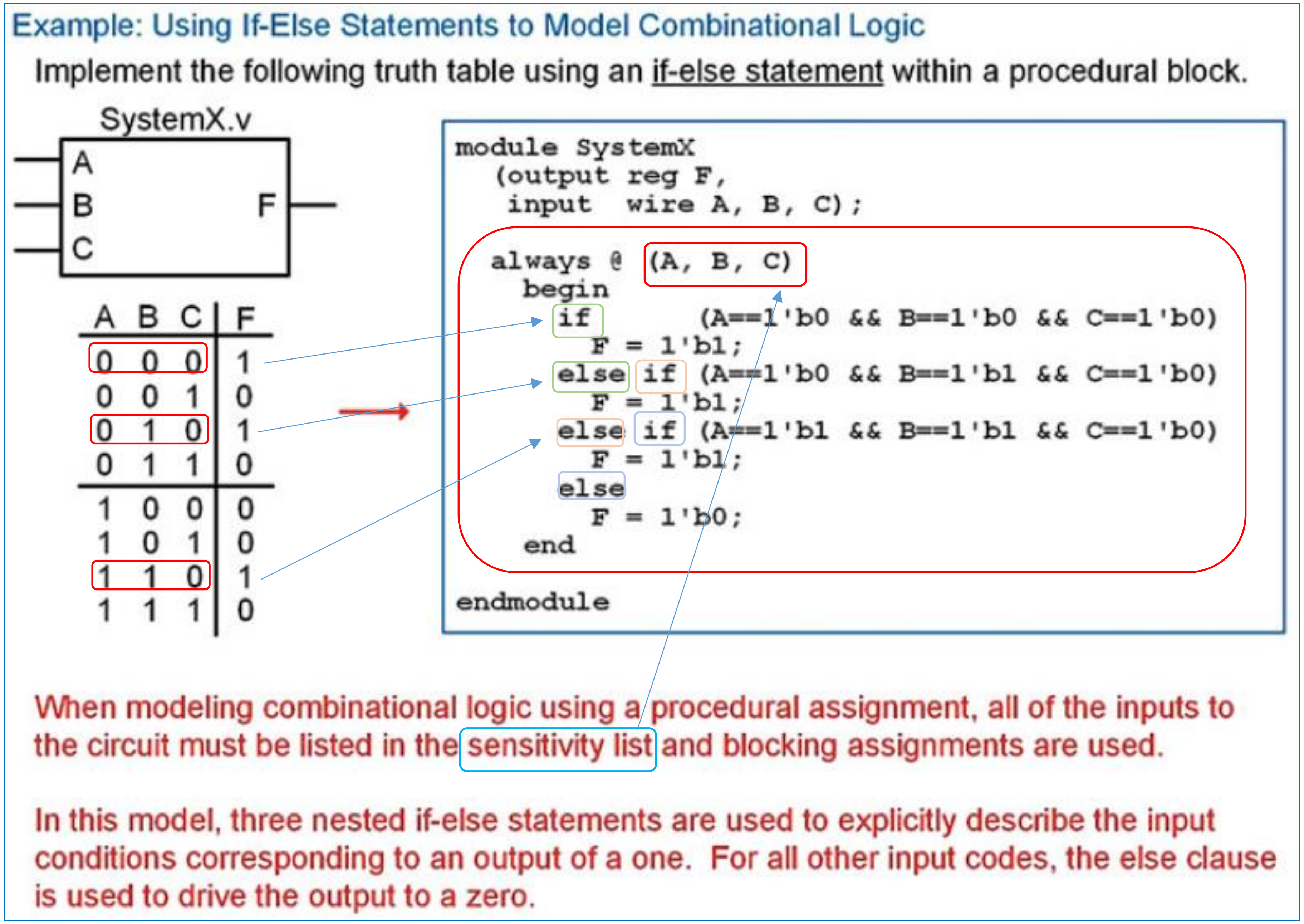
Programming Constructs - only allowed in procedural blocks (**init** or **always**).

- **if-else** Statements
- **case** Statements
- **casez** and **casex** Statements – allows x or z in input conditions.
- **forever** Loops
- **while** Loops
- **repeat** Loops
- **for** Loops

Notes Ch.2 p.11-14

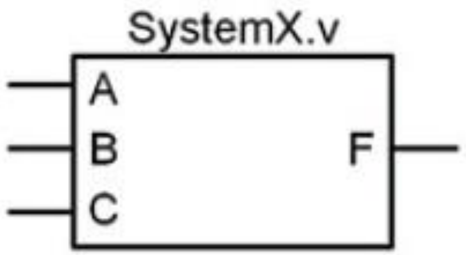
$\left.\begin{array}{l}\text{forever Loops}\\\text{while Loops}\\\text{repeat Loops}\end{array}\right\}$ Not covered in notes.

From: “Quick Start Guide to Verilog” p.75



Example: Using If-Else Statements to Model Combinational Logic

Implement the following truth table using an if-else statement within a procedural block.



A	B	C	F
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

```
module SystemX
   (output reg F,
    input  wire A, B, C);

  always @ (A, B, C)
    begin
      if         (A==1'b0 && B==1'b0 && C==1'b0)
        F = 1'b1;
      else if (A==1'b0 && B==1'b1 && C==1'b0)
        F = 1'b1;
      else if (A==1'b1 && B==1'b1 && C==1'b0)
        F = 1'b1;
      else
        F = 1'b0;
    end

endmodule
```

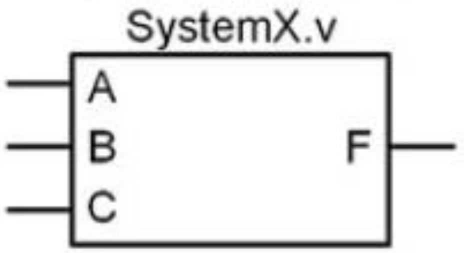
When modeling combinational logic using a procedural assignment, all of the inputs to the circuit must be listed in the sensitivity list and blocking assignments are used.

In this model, three nested if-else statements are used to explicitly describe the input conditions corresponding to an output of a one. For all other input codes, the else clause is used to drive the output to a zero.

From: “Quick Start Guide to Verilog” p.76

Example: Using Case Statements to Model Combinational Logic

Implement the following truth table using a <u>case statement</u> within a procedural block.



A	B	C	F
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

$\longrightarrow$

```
module SystemX
   (output reg F,
    input  wire A, B, C);

   always @ (A, B, C)
     begin
       case ( {A,B,C} )
           3'b000   : F = 1'b1;
           3'b001   : F = 1'b0;
           3'b010   : F = 1'b1;
           3'b011   : F = 1'b0;
           3'b100   : F = 1'b0;
           3'b101   : F = 1'b0;
           3'b110   : F = 1'b1;
           3'b111   : F = 1'b0;
           default  : F = 1'bX;
       endcase
     end

endmodule
```

Concatentate the 3 scalers into a 3-bit vector.

In this model, each binary input code is explicitly listed to create a model that mirrors the truth table format. Note that since the inputs are scalars, they must be concatenated so that 3-bit vectors can be listed as the input conditions. A default condition is needed to provide the output assignment for input codes containing X or Z.

Below are two alternative approaches of using a case statement that are more compact.

```
case ( {A,B,C} )
   3'b000   : F = 1'b1;
   3'b010   : F = 1'b1;
   3'b110   : F = 1'b1;
   default  : F = 1'b0;
endcase
```

In this approach, only the input codes corresponding to an output of one are explicitly listed. The default clause is used to handle all other input codes corresponding to an output of zero.

```
case ( {A,B,C} )
    3'b000, 3'b010, 3'b111  : F = 1'b1;
    default                 : F = 1'b0;
endcase
```

In this model, the input codes corresponding to the output assignment of one are listed on the same line, comma-delimited.

Example: Using **casex** statement to model a priority encoder

Notes Ch.2 p.13

The 2-bit output (Y1, Y0) indicates which input is 1.

If two or more inputs are 1, take the highest of them.

Output Z = 1 if any input is 1, else Z = 0.

w_3	w_2	w_1	w_0	y_1	y_0	z
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	x	0	1	1
0	1	x	x	1	0	1
1	x	x	x	1	1	1

X's are allowed as input conditions in **casex** but not in **case** statements.

```
module priority (w, y, z);
  input [3:0] w;
  output reg [1:0] y;
  output z;
  assign z=(w!=0);
  always @(w)
  begin
    casex (w)
      4'b1xxx: y = 3;
      4'b01xx: y = 2;
      4'b001x: y = 1;
      default: y = 0;
    endcase
  end
endmodule
```

4-bit

If w is not 0000 return 1 to z, else return 0 to z.

Two concurrent statements running at the same time.

If any changes in w then run the **always** block.

Can assign to **reg** but not **wire** in an **always** block.

Example: Using **for** statement to model a 4-bit adder

Notes Ch.2 p.14

```
module ripple (carryin, X, Y, S, carryout);
  parameter n = 4;
  input carryin;
  input [n-1:0] X, Y;
  output reg [n-1:0] S;
  output reg carryout;
  reg [n:0] C;
  always @( X, Y, carryin)
  begin
     C[0] = carryin;
     for (k = 0; k<= n-1; k=k+1)
     begin
        S[k] = X[k] ^ Y[k] ^ C[k];
        C[k+1] = (X[k] & Y[k]) | (C[k] & X[k]) | (C[k] & Y[k]);
     end
     carryout=C[n];
  end
endmodule
```

Any occurrence of "n" in the module will be replaced by 4.

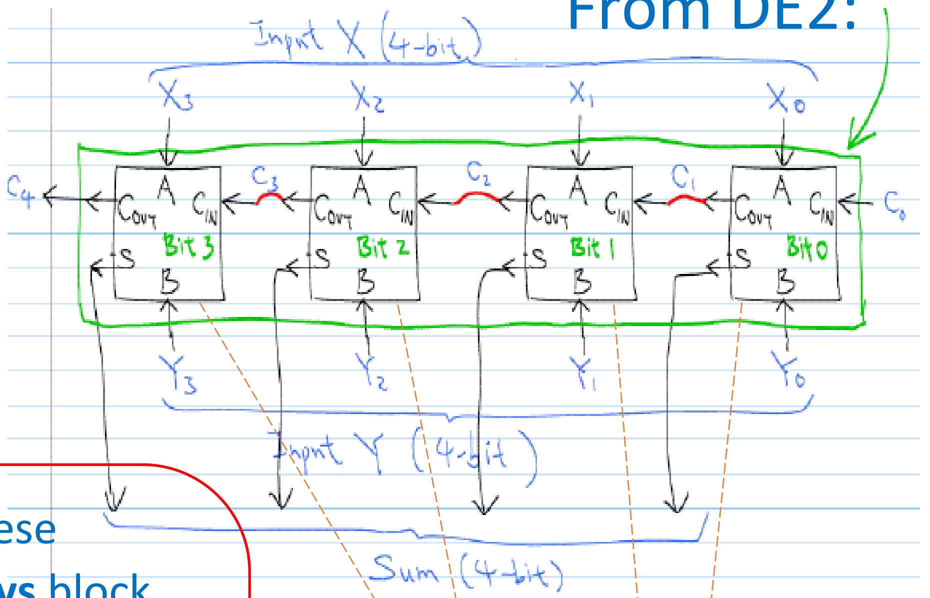
X, Y and S: 4-bit

C: 5-bit

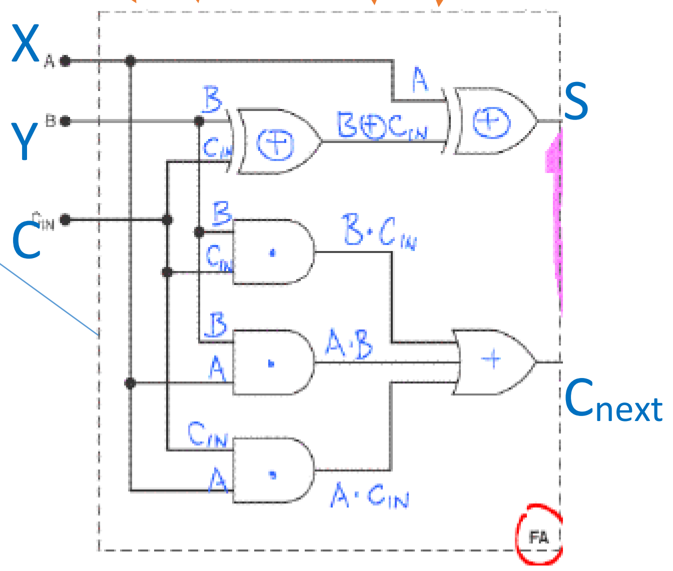
Only if any changes at these inputs then run the **always** block.

Duplicate the following circuit 4 times …

From DE2:



Input X (4-bit)
X3
X2
X1
X0
C4
C3
C2
C1
C0
COUT
A
CIN
S
B
Bit 3
Bit 2
Bit 1
Bit 0
Y3
Y2
Y1
Y0
Input Y (4-bit)
Sum (4-bit)



X
Y
C
A
B
CIN
B⊕CIN
B·CIN
A·B
A·CIN
S
Cnext
FA