ELEC0010
Digital Design

Prof Robert Killey

r.killey@ucl.ac.uk

Room 810, Roberts Building

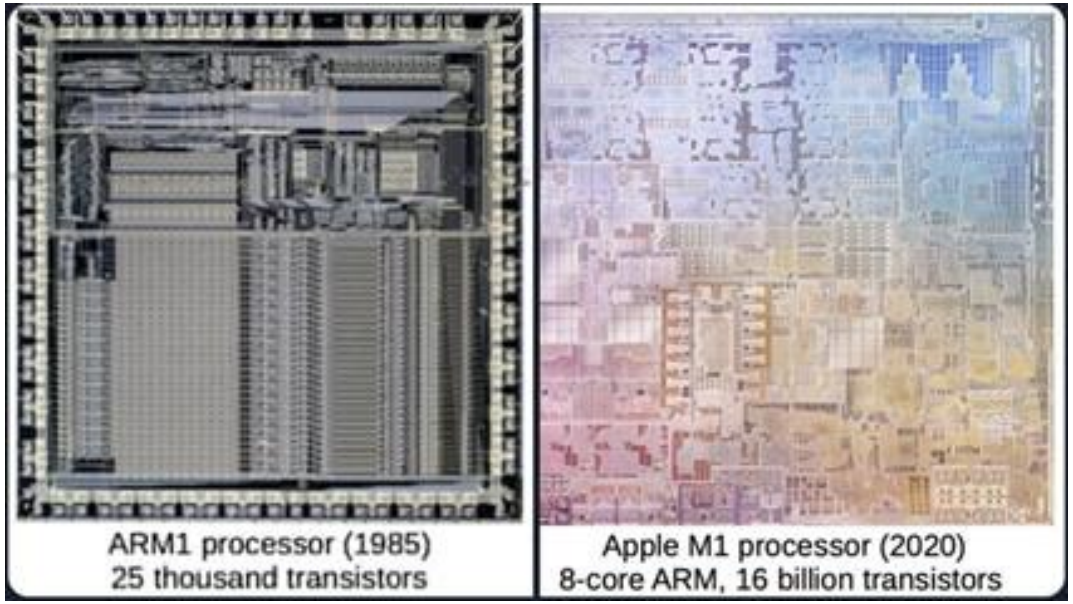
Part 1

Introduction to Logic Design with SystemVerilog

Introduction

From the introduction of electronic integrated circuits in the 1960s until the 1980's, manual methods were largely used in the design of digital circuits, e.g. the ARM1 microprocessor which has 25,000 transistors**.**

Today, high performance integrated circuits contain billions of transistors, and the process of digital hardware design is highly automated.



ARM1 processor (1985)
25 thousand transistors

Apple M1 processor (2020)
8-core ARM, 16 billion transistors

Picture credit: K. Shirriff

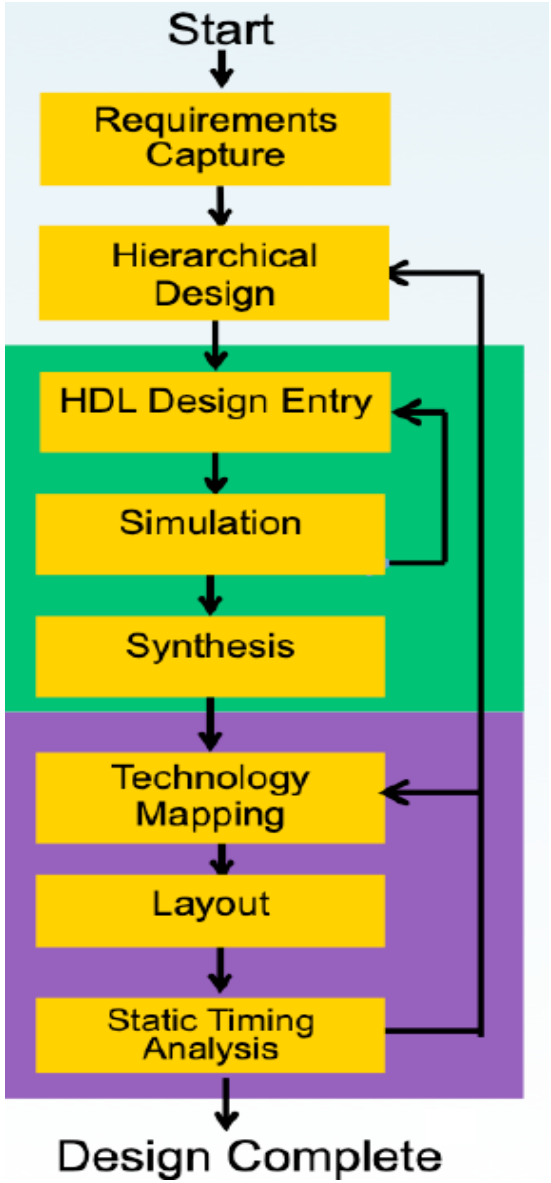
Introduction

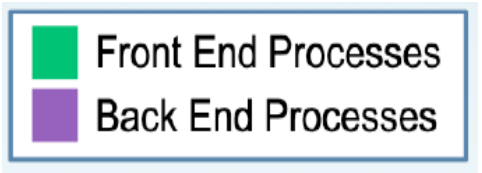
The digital system designer needs to:

- understand the CMOS design process
- create models of digital circuits in a hardware description language (HDL), and test them through simulations
- understand the process of circuit synthesis from HDL models
- understand how technology choices affect cost
- carry out timing and power analysis
- design circuits for testability

In this course, we will study the design process and techniques which are scalable to very complex systems, and are used in the development of all these systems.

1. The design process





Front-end processes –
deal with abstract logic

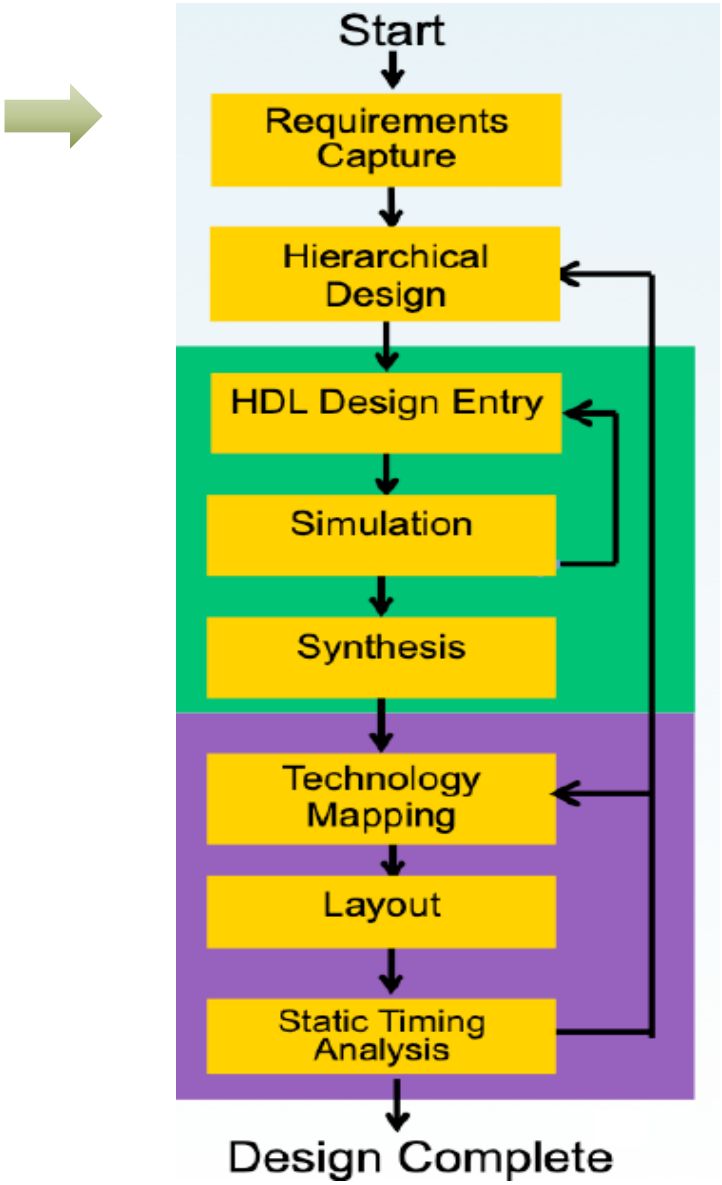
Back-end processes –
deal with circuit implementation with specific technology

1. The design process

Requirements capture and *hierarchical design* are common to all good structured design processes.

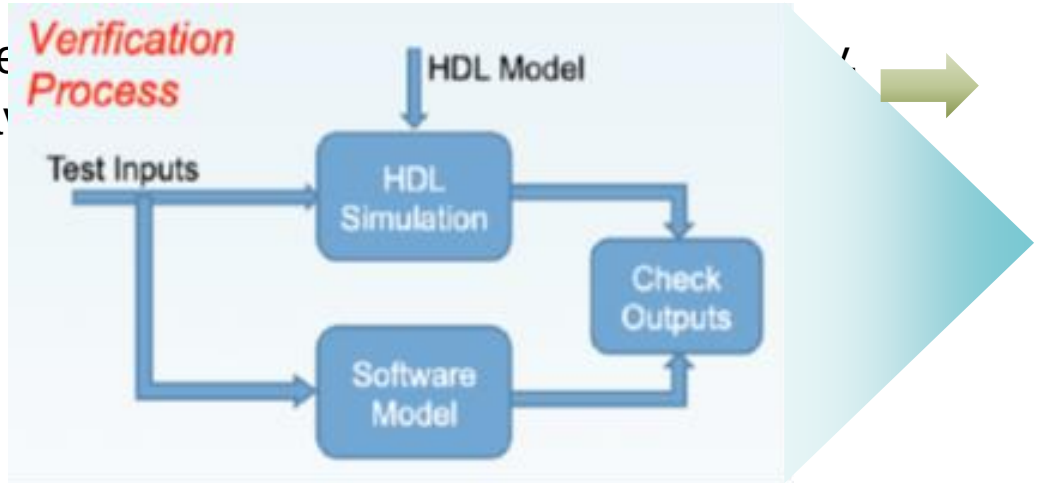
In digital design, requirements capture leads to constraints which are used to assess the success of a design during the timing analysis.

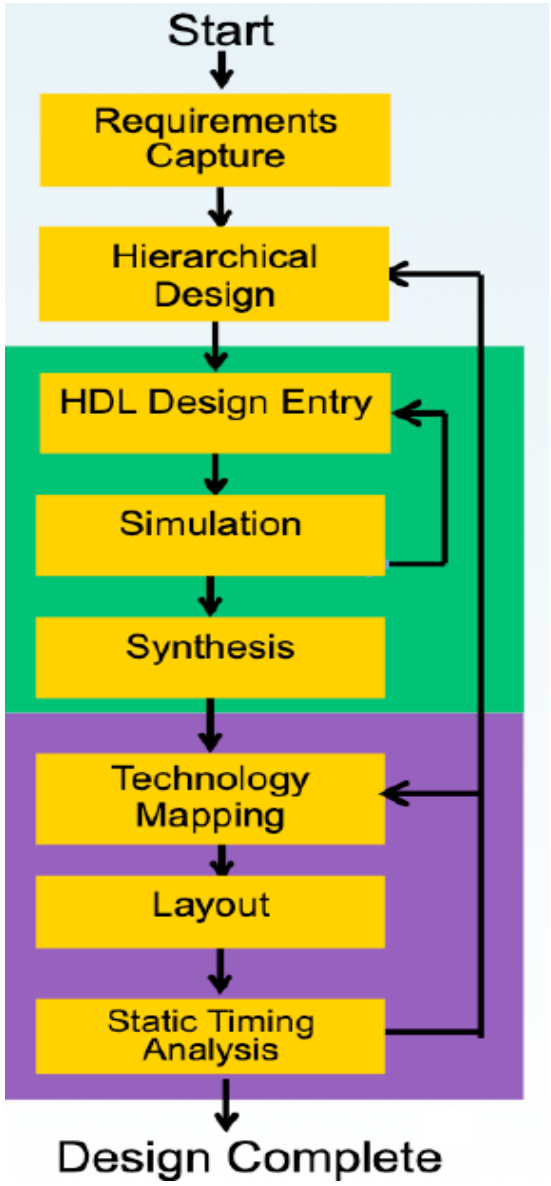
After requirements capture, partition design into sub-blocks



1. The design process

Next, hardware de
followed by beha
model.

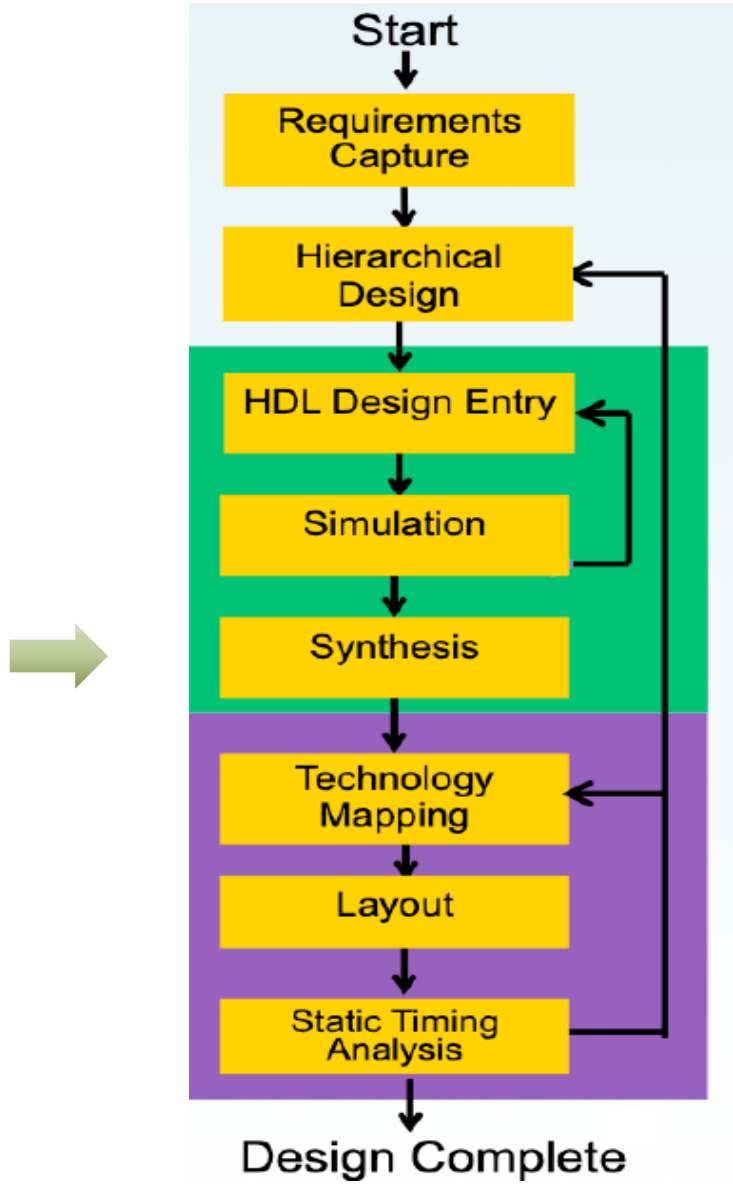




1. The design process

HDL code is input to synthesis software.

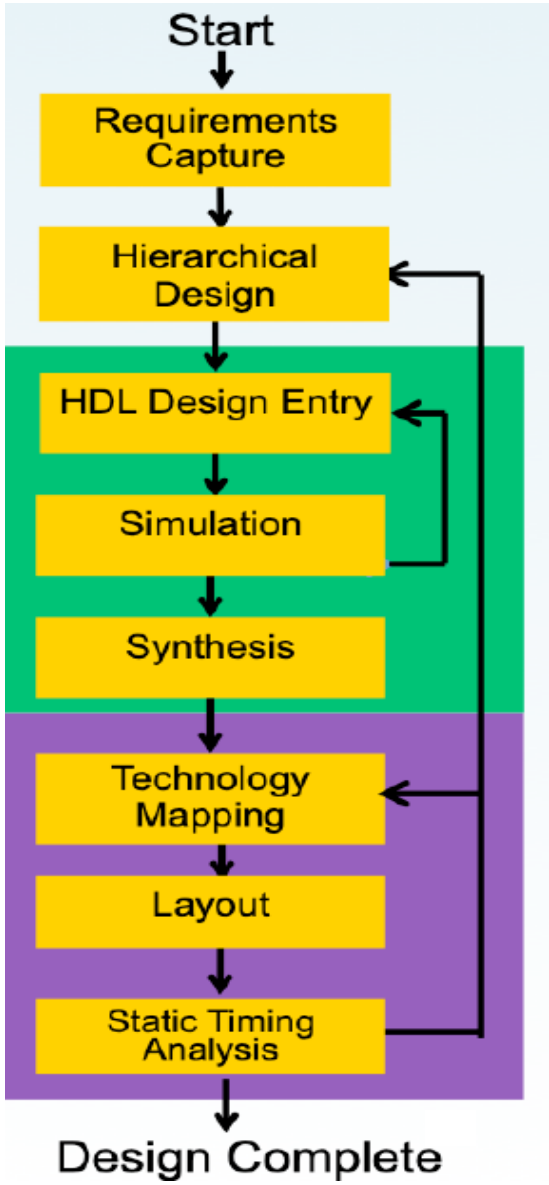
In this process, the behavioural HDL description is converted into a *netlist* of basic logic elements



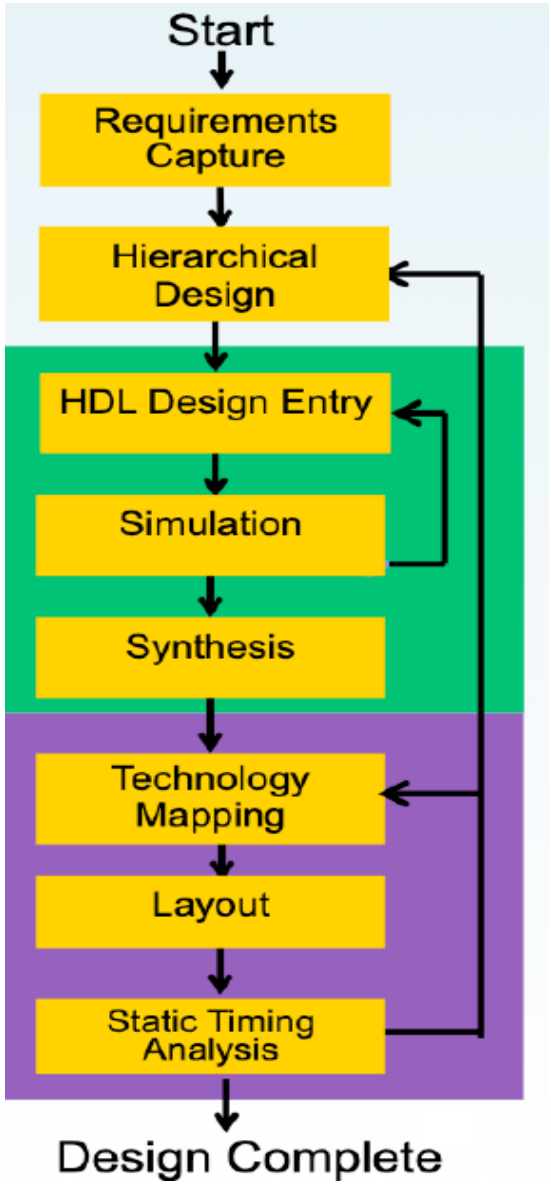
1. The design process

The back end processes deal with implementation of the circuits on a specific technology,

e.g. 14 nm standard cell CMOS or a specific field programmable gate array



The design process



Verify analog operation of the circuits meets requirements (e.g. timing constraints, power consumption)

2. Overview of Hardware Description Languages

- Classical design methods relied on schematics and manual methods to design a circuit.
- Today, computer-base languages are widely used to design circuits of enormous size and complexity.
- Using *hardware description languages* (HDLs), designers easily manage the complexity of large designs.
- Language-based designs are portable and independent of technology, allowing design teams to modify and re-use designs.
- A significant advantage from using an HDL is that a working circuit can be *synthesised* automatically from a language-based description.
- HDL-based synthesis is now the dominant design method used by industry. Designers build a software model of the design, verify its functionality, and then use a synthesis tool to automatically optimise the circuit and create a *netlist* in a physical technology.

2. Overview of Hardware Description Languages

Most commonly used languages: Verilog, SystemVerilog and VHDL.

All three languages are IEEE standards, and are supported by synthesis tools for application specific integrated circuits (ASICs) and field-programmable gate arrays (FPGAs).

Languages for analogue circuit design, such as Spice, cannot support abstract styles of design, and become impractical when used on a large scale.

Hybrid languages (e.g. Verilog-A) are used in designing mixed-signal circuits, which have both digital and analogue circuitry.

Systems-level design languages, such as SystemVerilog and SystemC can support a higher level of design abstraction than can be supported by Verilog or VHDL

2. Overview of Hardware Description Languages

Verilog started at the company Gateway Design Automation in 1984 and is now used extensively in the design of integrated circuits and digital systems.

Verilog has been designed to be intuitive and simple to learn

Verilog can closely simulate real circuits using built-in primitives, user-defined primitives, timing checks, delay simulation and the ability to apply external stimulus to designs enabling testing before synthesis.

The extension which made Verilog really take off was the Synthesis technology introduced in 1987. This, coupled with Verilog's ability to extensively verify a digital design, enables quick hardware design.

SystemVerilog was developed to support a higher level of design abstraction than Verilog, and is a superset of Verilog.

3. Structural Models of Combinational Logic

A SystemVerilog model of a circuit describes its functionality either with a *structural* or a *behavioural* view of its input-output (I/O) relationship

A structural view could be
• a *netlist* of gates
• a high-level architectural partitioning of the circuit into major functional blocks, such as arithmetic and logic unit (ALU)

A behavioural view could be
• a Boolean equation model
• a *register transfer level* (*RTL*) model
• an algorithm

3.1 Four-Value Logic

SystemVerilog uses a four-valued logic systems with the symbols: 0, 1, *x*, and *z*.

• 0 and 1 correspond to assertion (True) and de-assertion (False) respectively.

• X represents a condition of ambiguity, in which the simulator cannot determine where the value of the signal is 0 or 1.

• Z denotes a three-state condition, in which a wire is disconnected from its driver.

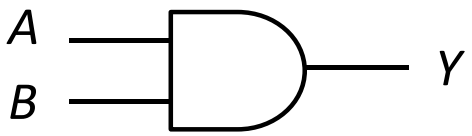
The symbol Z indicates that a node is being driven neither HIGH nor LOW.

The node is said to be floating, high impedance, or high Z.

A typical misconception is that a floating or undriven node is the same as a logic 0. In reality, a floating node might be 0, might be 1, or might be at some voltage in between

3.1 Four-Value Logic

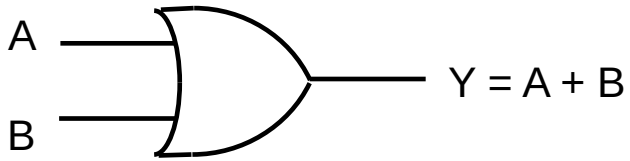
Example Write the truth table for the AND gate, including all four logic values.



A	B	Y
0	0	0
0	1	0
0	X	0
0	Z	0
1	0	0
1	1	1
1	X	X
1	Z	X
X	0	0
X	1	X
X	X	X
X	Z	X
Z	0	0
Z	1	X
Z	X	X
Z	Z	X

3.1 Four-Value Logic

Exercise Write the truth table for the **OR** gate, including all four logic values.



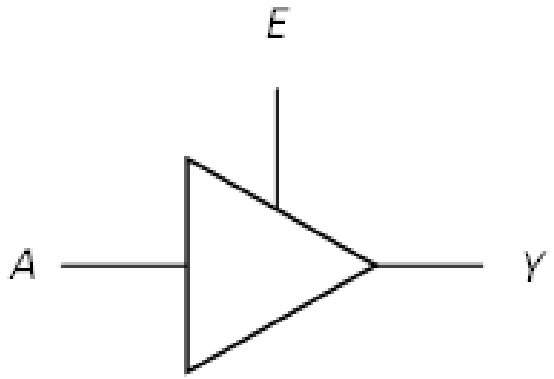
A	B	Y
0	0	0
0	1	1
0	X	X
0	Z	X
1	0	1
1	1	1
1	X	1
1	Z	1
X	0	X
X	1	1
X	X	X
X	Z	X
Z	0	X
Z	1	1
Z	X	X
Z	Z	X

3.1 Four-Value Logic

The *tristate buffer* has three possible output states: HIGH (1), LOW (0), and floating (Z).

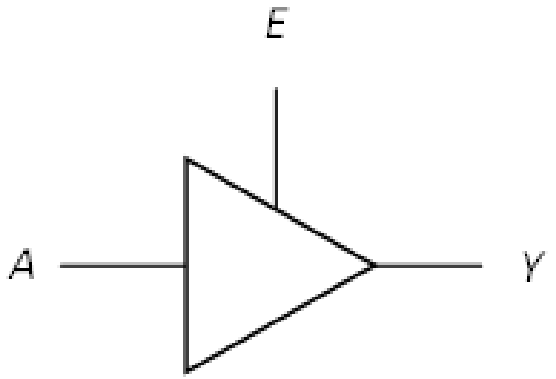
The tristate buffer has an input, A, an output, Y, and an enable, E.

- When the enable is TRUE, the tristate buffer acts as a simple buffer, transferring the input value to the output.
- When the enable is FALSE, the output is allowed to float (Z).



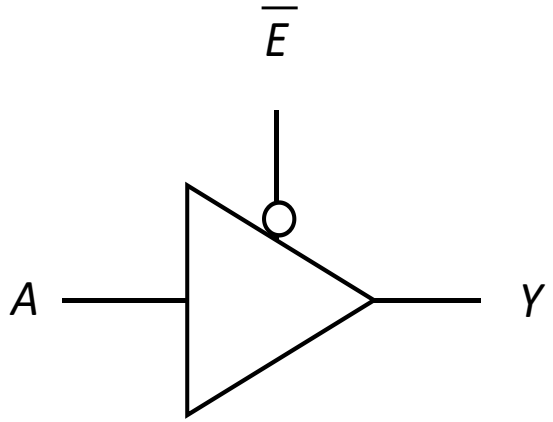
E	A	Y
0	0	Z
0	1	Z
1	0	0
1	1	1

3.1 Four-Value Logic



E	A	Y
0	0	Z
0	1	Z
1	0	0
1	1	1

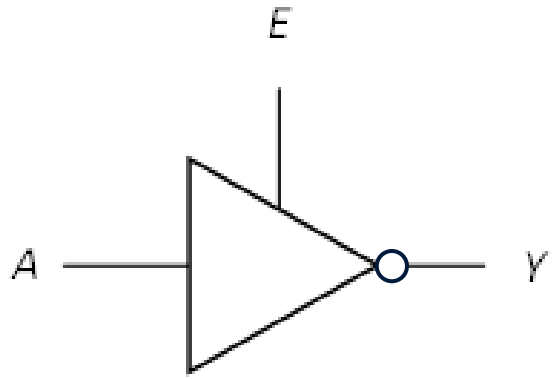
Tristate buffer with active high enable



$\overline{E}$	A	Y
0	0	0
0	1	1
1	0	Z
1	1	Z

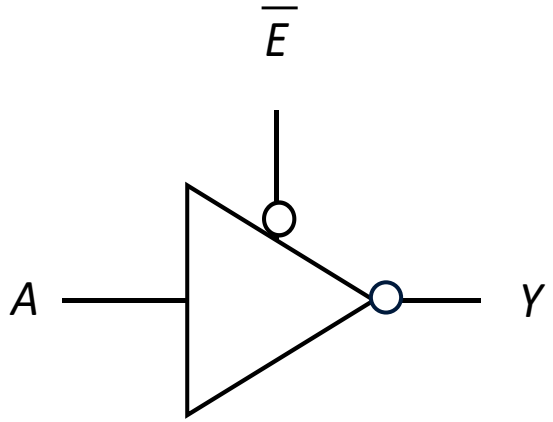
Tristate buffer with active low enable

3.1 Four-Value Logic



E	A	Y
0	0	Z
0	1	Z
1	0	1
1	1	0

Tristate inverter with active high enable



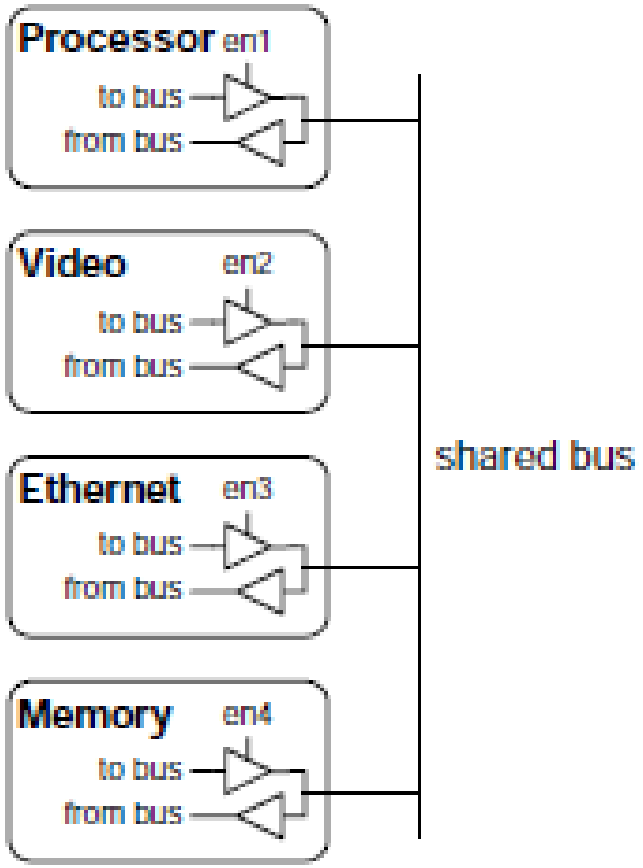
$\overline{E}$	A	Y
0	0	1
0	1	0
1	0	Z
1	1	Z

Tristate inverter with active low enable

3.1 Four-Value Logic

Tristate buffers are commonly used on *busses* that connect multiple chips. For example, a microprocessor, a video controller, and an Ethernet controller might all need to communicate with the memory system in a personal computer.

Each chip can connect to a shared memory bus using tristate buffers



Only one chip at a time is allowed to assert its enable signal to drive a value onto the bus.

The other chips must produce floating outputs so that they do not cause contention with the chip talking to the memory.

3.2 Some Language Rules

SystemVerilog, like any high level language has a number of tokens.

Tokens can be comments, delimiters, numbers, strings, identifiers and keywords. All keywords are in lower case.

White Space
The white space characters are space (\b), tabs (\t), newlines (\n). These are ignored except in strings

Comments
Two types of comments are supported, single line comments starting with ***//*** and multiple line comments delimited by ***/*** ... ****/.*** Comments cannot be nested.

It is usually a good idea to use single line comments to comment code and multiple lines comments to comment out sections of code when debugging.

3.3 Gate Types (SystemVerilog primitives for modelling combinational logic gates)

Keywords: **and**, **nand**, **or** , **nor**, **xor**, **xnor**, **buf**, **not**, **bufif0**, **bufif1**, **notif0**, **notif1**.

Some of the gates supplied by SystemVerilog are given above. A name can be given to the gate but this is optional.

For example an **and** gate,

and <name><list of arguments>
and myand (out, in1, in2, in3); // and gate with three inputs
and (out, in1, in2); // legal gate with no name.

Note the convention of putting the output at the start of the argument list.

The **buf** and **not** gates each have one input and one or more outputs. The convention is the same, the outputs come first and the last argument in the list is the input.

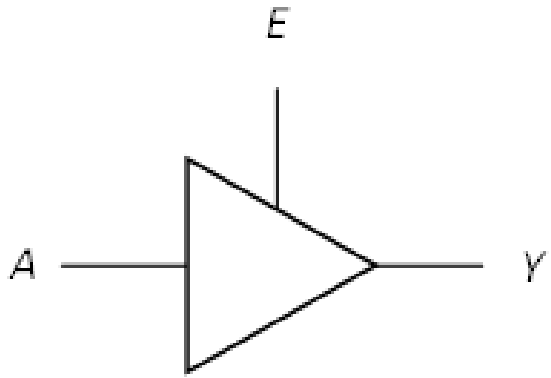
buf mybuf(out1, out2, out3, in);
not (out, in);

3.3 Gate Types

The **bufif0, bufif1** primitives refer to tristate buffers with active low enable and active high enable, respectively.

The order of the arguments is shown in this example:

bufif1 (Y, A, E);



E	A	Y
0	0	Z
0	1	Z
1	0	0
1	1	1

The **notif0, notif1** primitives refer to tristate inverters with active low enable and active high enable, respectively.

3.4 Structural Connectivity

Keyword **wire**

In Verilog, the logic gates (primitives) can be interconnected by variables of type **wire**.

The logic value of a **wire** is determined dynamically during simulation by what is connected to the wire.

If a wire is attached to the output of a primitive, it is said to be driven by the primitive, and the primitive is said to be its driver.

In Verilog, any identifier that is referenced without having a type declaration is by default of type **wire**.

3.4 Structural Connectivity

In Verilog, another variable type is **reg** (short for register) which is used for values which are evaluated at particular points. They act like variables in ordinary procedural languages. They store information while the program executes.

In SystemVerilog, we can use variable type **logic** for any logic signal (rather than **wire** or **reg**).

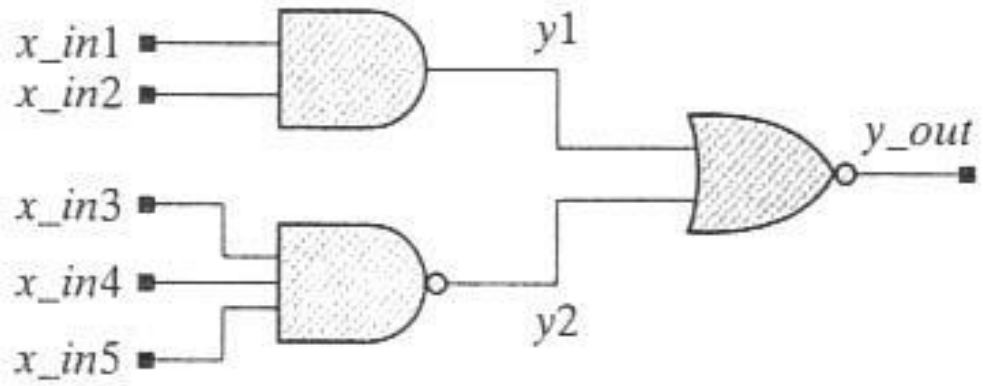
Any identifier that is referenced without having a type declaration is by default of type **logic**.

3.4 Structural Connectivity

Example

Draw a schematic of the circuit described by the netlist of primitives below.

Solution



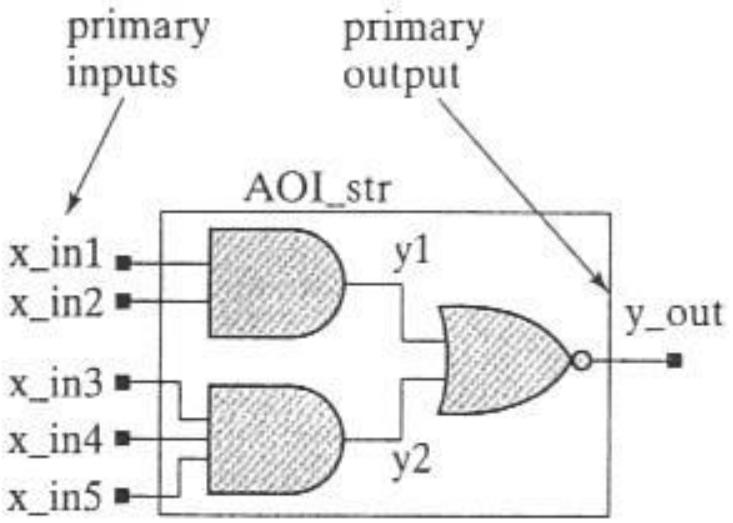

```
wire    y1, y2;
nor     (y_out, y1, y2);
and     (y1, x_in1, x_in2);
nand    (y2, x_in3, x_in4, x_in5);
```

3.5 SystemVerilog Structural Models

A structural model of a logic circuit is declared and encapsulated as a named Verilog module, consisting of:

(1) a module name accompanied by its ports
(2) a list of operational modes of the ports (e.g. input)
(3) an optional list of internal wires and/or other variables used by the model
(4) A list of interconnected primitives and/or other modules

A complete SystemVerilog structural model of the circuit below illustrates some terminology.




```
module AOI_str(output logic y_out,
                        input logic   x_in1, x_in2, x_in3, x_in4, x_in5);
logic y1;
logic y2;

    nor     (y_out, y1, y2);
    and     (y1, x_in1, x_in2);
    and     (y2, x_in3, x_in4, x_in5);

endmodule
```

The keywords **module** and **endmodule** enclose (encapsulate) the description.

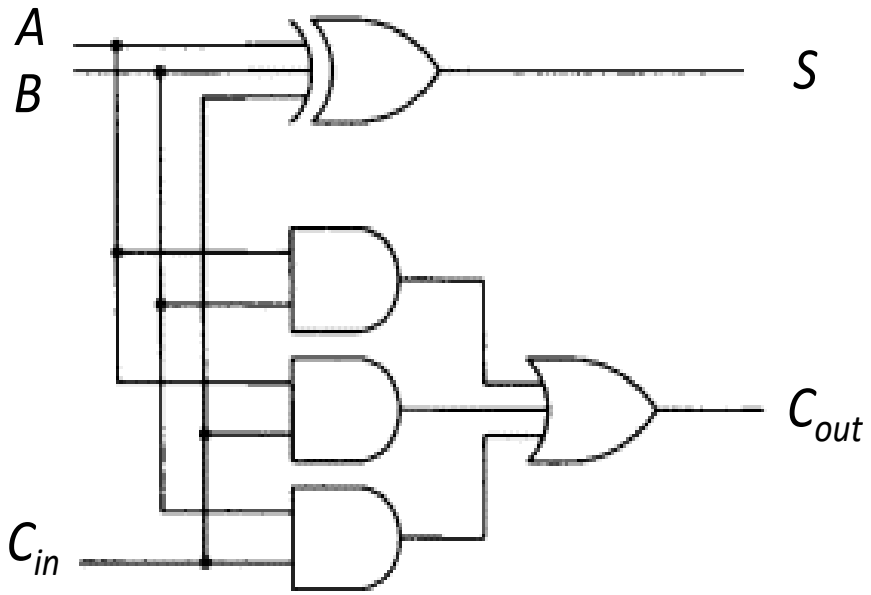
Internal variable y1 and y2 are declared, which are used to interconnect the logic gates.

The ports of the module define its interface to the environment in which it is used. The *mode* of a port determines the direction that information may flow through the port: **input**, **output** or **inout**.

The environment of a module interacts with its ports, but does not have access to the internal description of the module's functionality.

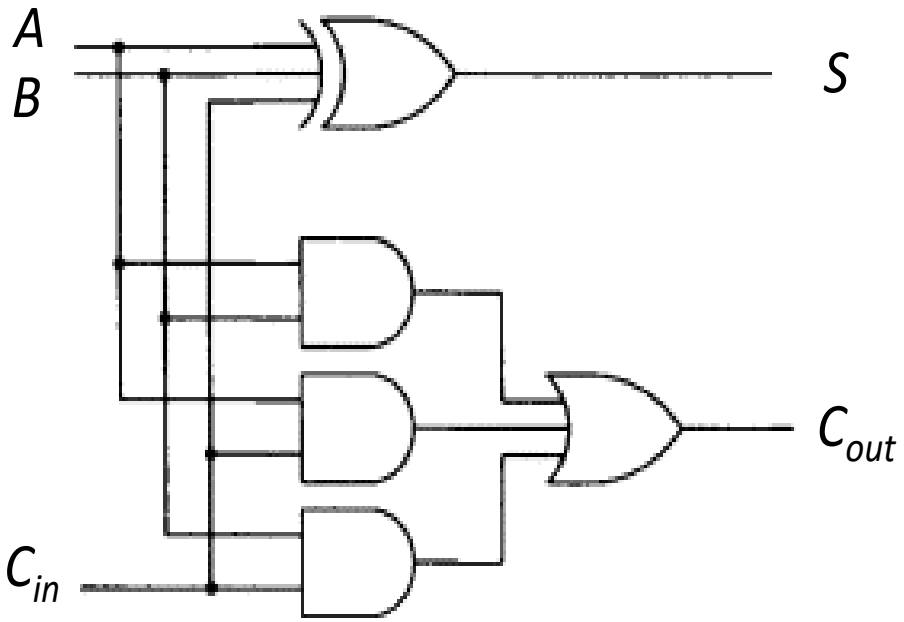
3.5 SystemVerilog Structural Models

Exercise: Write a SystemVerilog module, with the name full_adder, with a netlist for the full adder circuit:

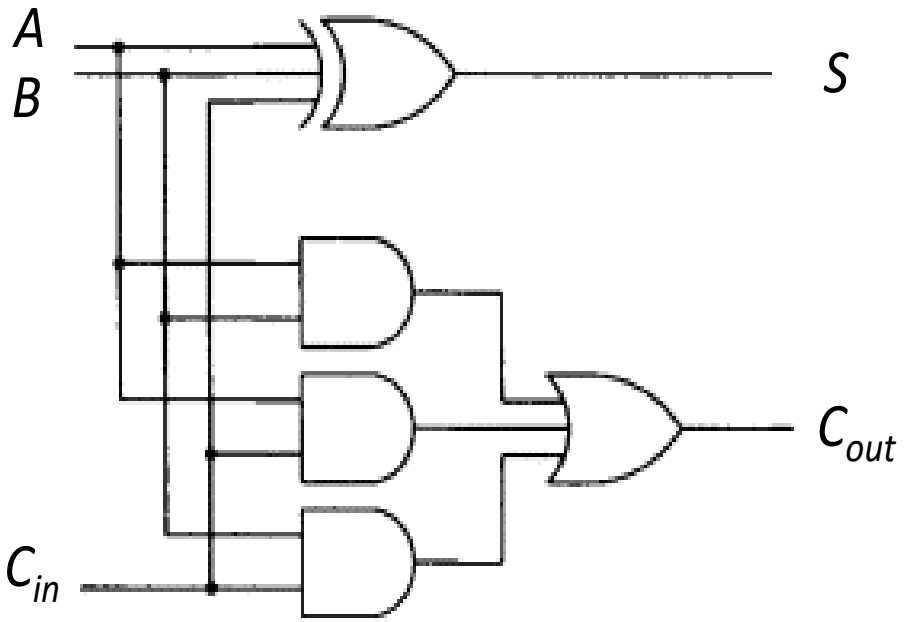


A	B	C_{in}	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

3.5 SystemVerilog Structural Models



3.5 SystemVerilog Structural Models




```
module full_adder(output logic s, c_out,
                  input logic a, b, c_in);

  logic x1, x2, x3;
  xor (s, a, b, c_in);
  and (x1, a, b);
  and (x2, a, c_in);
  and x3, b, c_in);
  or (c_out, x1, x2, x3);
endmodule
```

3.6 Vectors in SystemVerilog

A vector in SystemVerilog is denoted by square brackets, enclosing a continuous range of bits.

The leftmost index in the bit range is the most significant, the rightmost is the least significant bit.

Example

If an 8-bit word vect_word[7:0] has a stored value of decimal 4, what are the values of vect_word[2], vect_word[3:0], and vect_word[5:1]?

- vect_word [2] has a value of 1. 00000100
- vect_word [3:0] has a value of 4. 00000100
- vect_word [5:1] has a value of 2. 00000100

3.6 Vectors in SystemVerilog

It is possible to declare a variable which is a set of vectors, for example used to describe a memory array.

Example Declare a 32-element memory 8 bits wide with variable name mem:

logic [7:0] mem [0:31]

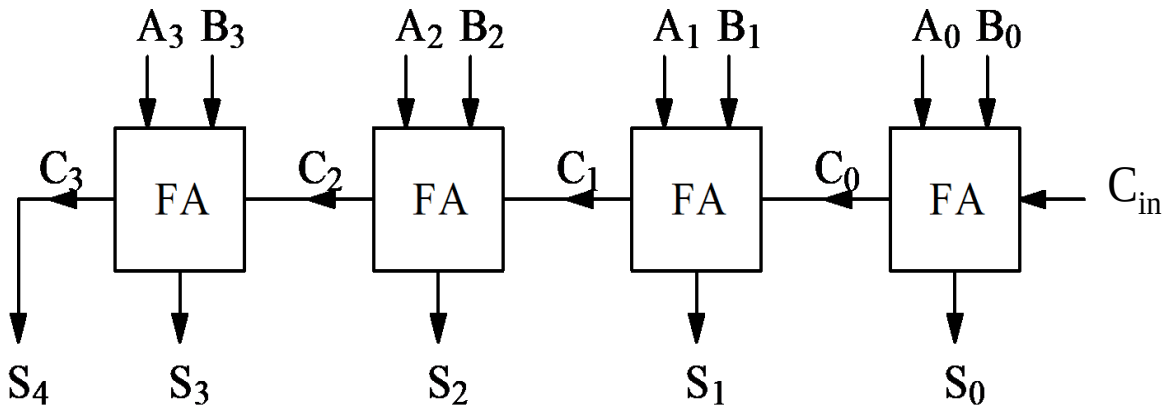
This variable could be used, for example, to store the contents of a RAM or ROM memory array with addresses 0 to 31, with each word 8 bits wide.

3.7 Top-Down Design and Nested Modules

Complex systems are designed by systematically and repeatedly partitioning them into simpler functional units. A high level partition and organisation of the design is sometimes referred to as an *architecture*.

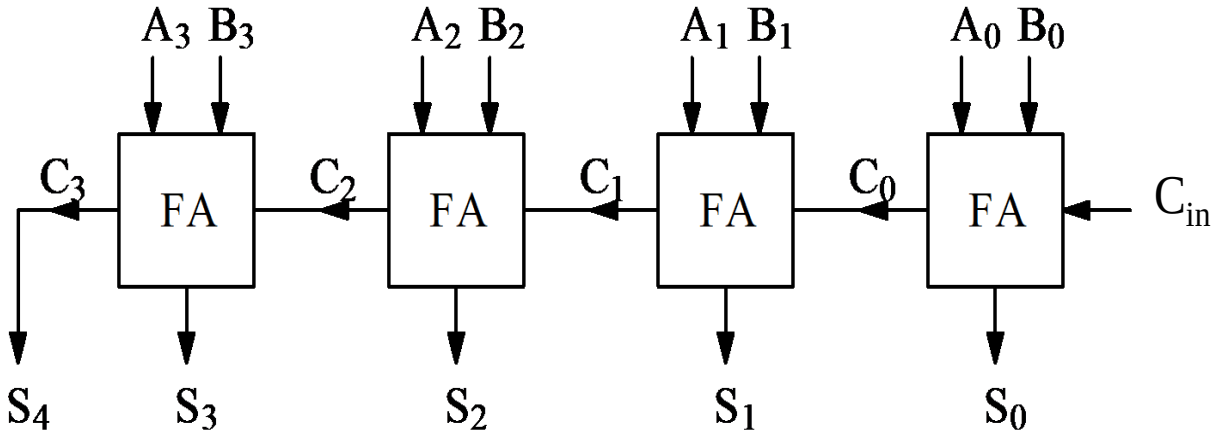
Example

Construct a 4-bit adder by combining four instances of the full adder cell module we wrote previously



3.7 Top-Down Design and Nested Modules

```
module full_adder(output logic s, c_out,
                  input logic a, b, c_in);

  logic x1, x2, x3;

  xor (s, a, b, c_in);
  and (x1, a, b);
  and (x2, a, c_in);
  and x3, b, c_in);
  or (c_out, x1, x2, x3);
endmodule
```




```
module adder_4bit (output [4:0] s,
                   input [3:0] a, b,
                   input c_in);

  logic c0, c1, c2;       // Intermediate carries

  full_adder (s[0], c0, a[0], b[0], c_in);
  full_adder (s[1], c1, a[1], b[1], c0);
  full_adder (s[2], c2, a[2], b[2], c1);
  full_adder (s[3], s[4], a[3], b[3], c2);

endmodule
```

3.8 Truth table models of combinational and sequential logic with SystemVerilog

SystemVerilog supports truth-table models of combinational and sequential logic.

This is done by building user-defined primitives (UDPs), which use truth tables.

UDPs are declared in a source file in the same way that a module is declared, but with the encapsulating keyword pair **primitive** and **endprimitive**

The truth table for a UDP consists of a section of columns, one for each input, followed by a colon (:) and a final column that specifies the output. The order of the input columns must be the same as the order in which the input ports are listed in the declaration. There can be only one output variable.

Input and output ports of a UDP *must be scalars*.

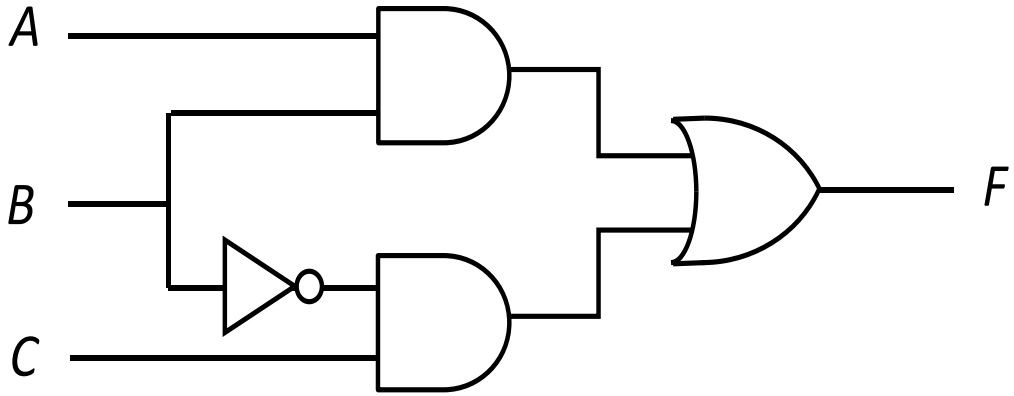
Note that a simulator will automatically assign the default value X to the output of a UDP if its inputs have values that do not match a row of the table.

An input value of Z is treated as X by the simulator.

3.8 Truth table models of combinational and sequential logic with SystemVerilog

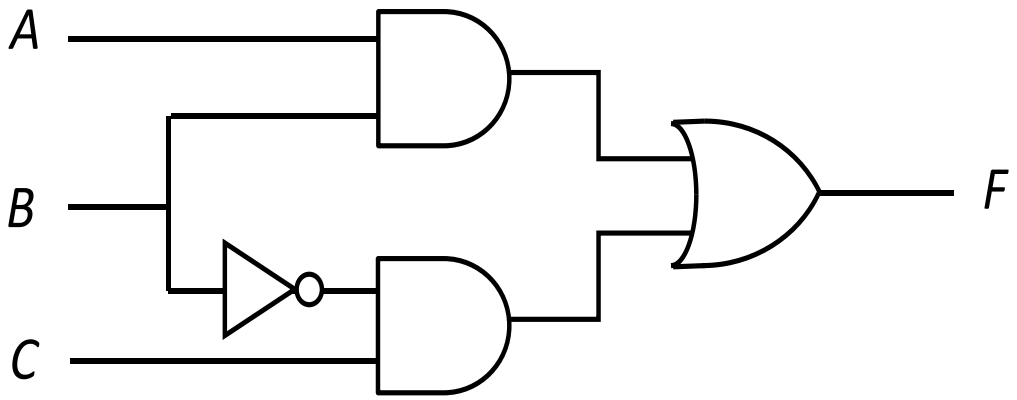
Example

Write a SystemVerilog user defined primitive with the name *UDP_SOP_circuit* that has the functionality of the following circuit:



3.8 Truth table models of combinational and sequential logic with SystemVerilog

```
primitive UDP_SOP_circuit(f, a, b, c);
output f;
input a, b, c;

 table
 // a b c : f
    0 0 0 : 0;
    0 0 1 : 1;
    0 1 0 : 0;
    0 1 1 : 0;
    1 0 0 : 0;
    1 0 1 : 1;
    1 1 0 : 1;
    1 1 1 : 1;
 endtable

endprimitive
```



4. Behavioural Modelling

SystemVerilog supports structural and behavioural modelling.

Structural modelling connects primitive gates and/or functional units to create a specified functionality.

not the most convenient or understandable models of a circuit, especially for large, complex designs.

Behavioural modelling is the most commonly used descriptive style used by industry, enabling the design of massive chips.

Behavioural modelling describes the functionality of the design – i.e. *what* the designed circuit will do, not *how* to build it.

allows the designer to rapidly explore alternatives and tradeoffs, rather than spending time on individual logic gates and interconnections

4.1 Operators

Operator type	Operator symbols	Operation performed
Bitwise	~	1's complement
	&	Bitwise AND
	|	Bitwise OR
	^	Bitwise XOR
	~^ or ^~	Bitwise XNOR
Logical	!	NOT
	&&	AND
	||	OR
Reduction	&	Reduction AND
	~&	Reduction NAND
	|	Reduction OR
	~|	Reduction NOR
	^	Reduction XOR
	~^ or ^~	Reduction XNOR

4.1 Operators

Operator type	Operator symbols	Operation performed
Arithmetic	+	Addition
	-	Subtraction
	-	2's complement
	*	Multiplication
	/	Division
	**	exponent
Relational	>	Greater than
	<	Less than
	>=	Greater than or equal to
	<=	Less than or equal to
	==	logical equality
	!=	Logical inequality
	===	4-state logical equality
	!==	4-state Logical inequality
Shift	>>	Logical Right shift
	<<	Logical Left shift
Concatenation	{ , }	Concatenation
Replication	{n{m}}	Replicate value m for n times
Conditional	? :	Conditional

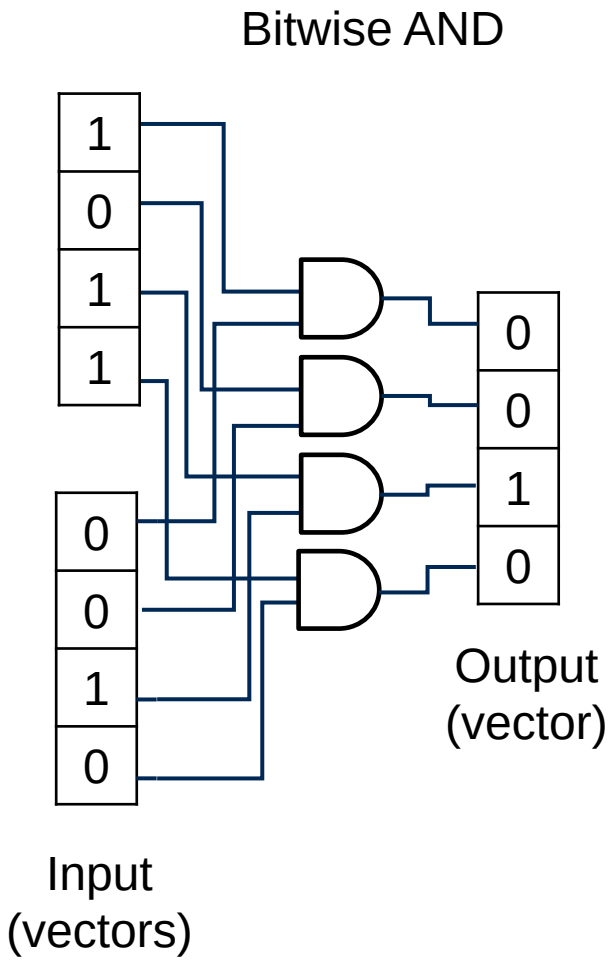
4.1 Operators

It is necessary to distinguish between arithmetic and logical operations, so different symbols are used for each.

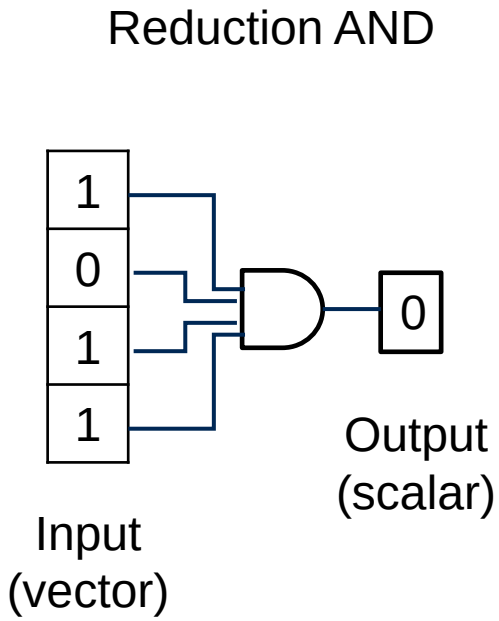
The plus symbol (+) indicates the arithmetic operation of addition; the bitwise logic AND operation uses the symbol &.

4.1 Operators

Bitwise logical operations operate bit-by-bit on a pair of vector operands, to produce a vector result.



Reduction logical operations operate on a single vector operand to produce a scalar result.



4.1 Operators

The *equality symbol*, ==, for checking the equality between to different values (a relational operator type), uses two equals signs (without a space between them), to distinguish it from the equals sign = used to assign a value to an operand.

Operator type	Operator symbols	Operation performed
Relational	>	Greater than
	<	Less than
	>=	Greater than or equal to
	<=	Less than or equal to
	==	logical equality
	!=	Logical inequality
	===	4-state logical equality
	!==	4-state Logical inequality

4.1 Operators

Operator type	Operator symbols	Operation performed
Concatenation	{ , }	Concatenation
Replication	{n{m}}	Replicate value m for n times

The *concatenation symbol* { , } allows two vectors to be concatenated (joined together) to form a large vector.

For example, if x = 0011 and y = 1010, then w = {x, y} = 00111010.

The *replication symbol* {n{m}} allows a vector to be created which consists of multiple concatenated copies of the input operand, e.g. if x = 0011, then w = {3,{x}} = 001100110011.

4.1 Operators

The *conditional operator* (? :) acts like a multiplexer, and will be explained in the next section.

Operator type	Operator symbols	Operation performed
Conditional	? :	Conditional

4.2 Definition of Constants

The definition of constants in SystemVerilog supports the addition of a width parameter.

The basic syntax is:

<Width in bits>'<base letter><number>

Examples:

4'b1010 - Binary 1010 (using 4 bits)

20'd44 - Decimal 44 (using 20 bits - 0 extension is automatic)

12'h123 - Hexadecimal 123 (using 12 bits)

6'o77 - Octal 77 (using 6 bits)

8’bx – Binary xxxxxxxx (using 8 bits)

4.2 Definition of Constants

Constants are declared using the keyword **parameter**, e.g.

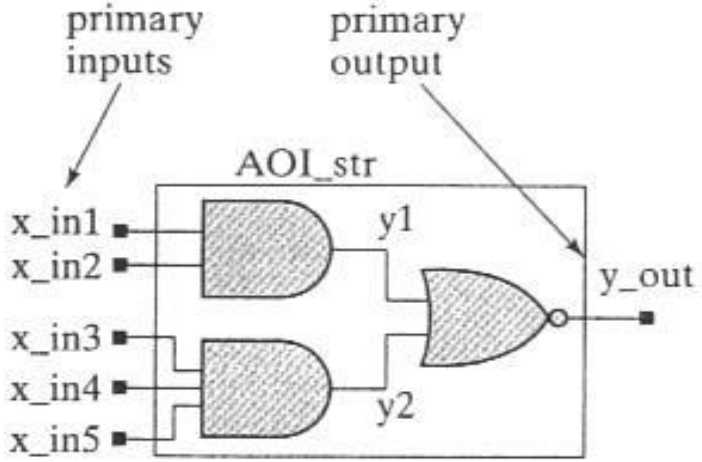
parameter a = 4’b1010

4.3 Boolean Equation-Based Behavioural Models of Combinational Logic

A Boolean equation describes combinational logic by an expression of operations on variables. Its counterpart in SystemVerilog is the *continuous assignment statement.*

Keyword: **assign**

Example
Write a SystemVerilog model, with module name *AOI_CA*, which uses a continuous assignment statement to form the output of the circuit on the right




```
module AOI_CA (
input logic    x_in1, x_in2, x_in3, x_in4, x_in5,
output logic  y_out);

  assign  y_out = ~((x_in1 & x_in2) | (x_in3 & x_in4 & x_in5));

endmodule
```

4.4 Conditional operators

The conditional operator (? :) provides the functionality of a multiplexer or a tri-state buffer. The conditional operator (? :) takes three operands:

<condition> ? <true-expression> : <false-expression>

The condition is evaluated. If the result is logic 1, the true expression is evaluated and used to assign a value to the left-hand side of an assignment statement.

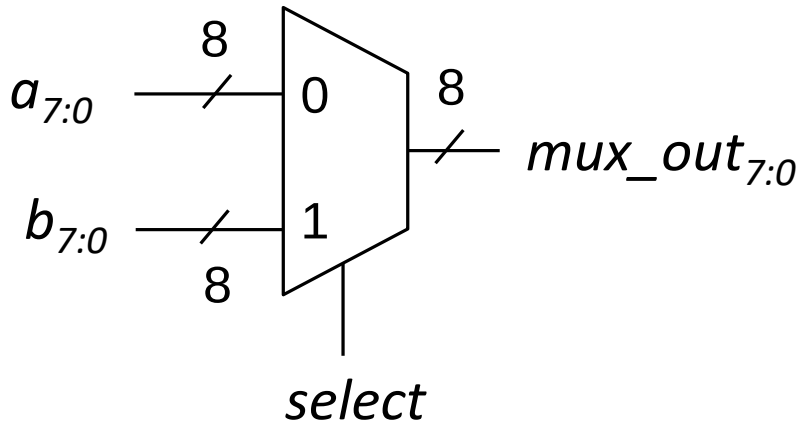
If the result is logic 0, the false expression is evaluated.

The two conditions together are equivalent to an if-else condition.

4.4 Conditional operators

Example Write a module which describes the behaviour of the 2-to1 multiplexer

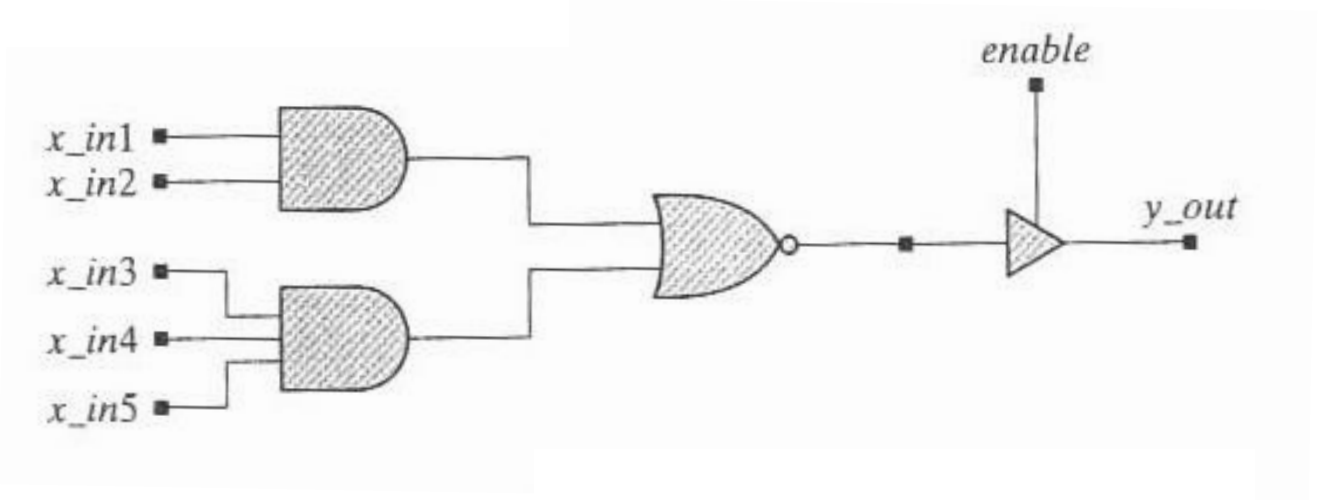
```
// Behavioural description of a 2-to-1 multiplexer

module mux_2_to_1 (output logic [7:0] mux_out,
                   input logic [7:0] a, b,
                   input logic select);

assign mux_out = (select) ? b : a ;
endmodule
```



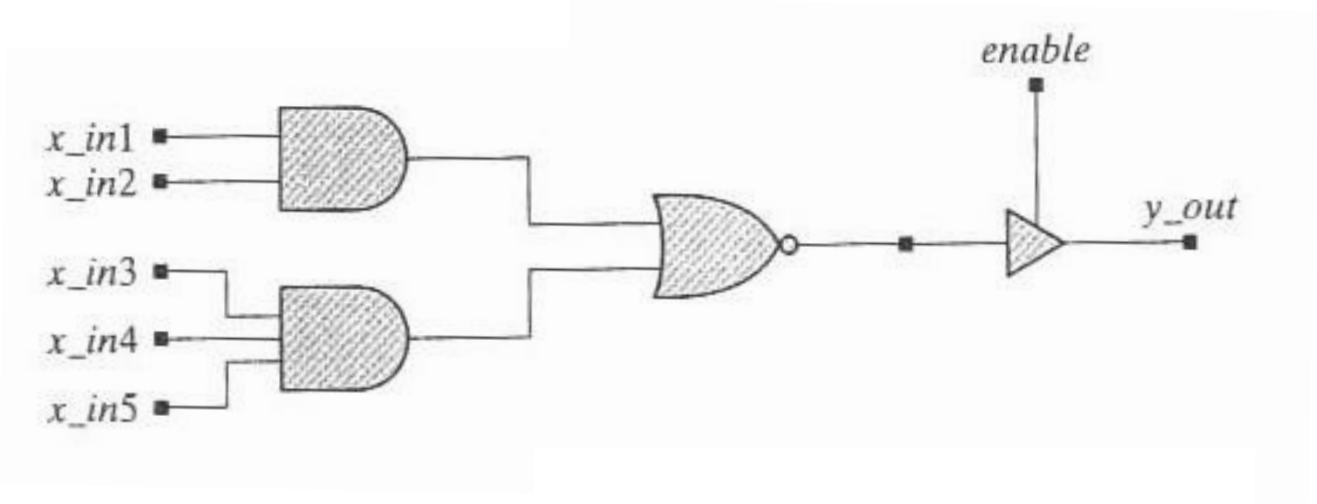
4.4 Conditional operators

Example

Write a Verilog model of this circuit with the module name *AOI_CA_3state*.



4.4 Conditional operators



Solution

```
module AOI_CA_3state (
input logic     x_in1, x_in2, x_in3, x_in4, x_in5, enable,
output logic y_out);

  assign  y_out = (enable) ? ~((x_in1 & x_in2) | (x_in3 & x_in4 & x_in5)) : 1'bz;

endmodule
```

Note that the conditional operator (**? :**) acts like a software if-then-else switch that selects between two expressions.

In the example of *AOI_3state*, if the value of *enable* is true, then the expression to the right of the **?** is evaluated and used to assign value to *y_out*, otherwise, the expression to the right of the **:** is used.

5. Test benches

A test bench is an HDL program used for describing and applying a stimulus to an HDL model of a circuit (the device under test, or DUT) in order to test it and observe its response during simulation.

5. Test benches

Test benches use the **initial** statement to provide a stimulus to the circuit being tested.

The **initial** statement executes only once, starting from simulation time 0, and may continue with any operations that are delayed by a given number of time units, as specified by the symbol #. For example, consider the **initial** block

```
initial
begin
a = 0; b = 0;          //  At time 0, a and b are set to 0
#10 a = 1;             //  10 time units later, a is changed to 1
#20 a = 0; b = 1;      //  20 time units after that (i.e., at t = 30), a is changed to 0 and b to 1.
end
```

5. Test benches

A test bench has the following form:

module test_module_name;

// Declare local identifiers.

// Instantiate the design module under test.

// Specify a stopwatch, using $finish to terminate the simulation.

// Generate stimulus, using **initial** and **always** statements.

// Display the output response (text or graphics (or both)).

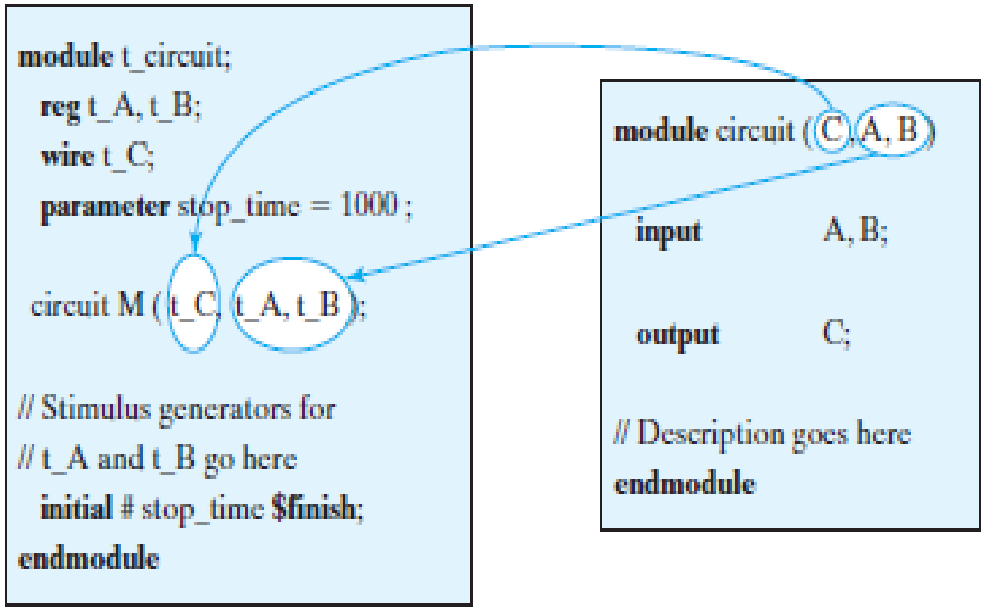
endmodule

5. Test benches

A test bench is written like any other SystemVerilog module, but it typically has no inputs or outputs.

The signals that are applied as inputs to the design module for simulation, and the outputs of the design module that are displayed for testing, are declared in the test bench module as local **logic** data type.

The interaction between a test bench and a design module is shown



5. Test benches

The response to the stimulus generated by the **initial** block will appear in text format as standard output and as waveforms (timing diagrams) in simulators having graphical output capability.

Numerical outputs are displayed by using SystemVerilog *system tasks* .

These are built-in system functions that are recognized by keywords that begin with the symbol **$**. Some of the system tasks that are useful for display are:

$display — display a one-time value of variables or strings with an end-of-line return,

$write — same as **$display ,** but without going to next line,

$monitor — display variables whenever a value changes during a simulation run,

$time — display the simulation time,

$finish — terminate the simulation.

5. Test benches

The syntax for **$display, $write,** and **$monitor** is of the form

<Task-name> (<format specification>, <argument list>);

Example

$display (“C = %d, A = %b B = %b”, C, A, B);

%d – present value in decimal

%b – present value in binary

%h – present value in hexadecimal

5. Test benches

Example – Testbench to test a 2-to-1 multiplexer module

```
// Behavioural description of a 2-to-1 multiplexer

module mux_2_to_1 (output logic [7:0] mux_out,
                   input logic [7:0] a, b,
                   input logic select);

assign mux_out = (select) ? b : a ;
endmodule
```

```
`timescale 1ns/1ps
module mux_2_to_1_tb;
logic [7:0] t_mux_out, t_a, t_b;
logic t_select;

mux_2_to_1    M1 (t_mux_out, t_a, t_b, t_select);

initial begin                    // Stimulus generator
t_select = 1; t_a = 3; t_b = 5;
#10 t_a = 9; t_b = 14; #10 t_select = 0;
#10 t_a = 48; t_b = 27;
end

 initial begin                   // Response monitor

$monitor ("t_select = %d t_a = %d t_b = %d t_mux_out = %d",
          t_select, t_a, t_b, t_mux_out);
end
endmodule
```

5. Test benches

Running the simulation will produce the following result:

Simulation log:

```
t_select = 1        t_a =  3  t_b =  5  t_mux_out =  5
t_select = 1        t_a =  9  t_b = 14  t_mux_out = 14
t_select = 0        t_a =  9  t_b = 14  t_mux_out =  9
t_select = 0        t_a = 48  t_b = 27  t_mux_out = 48
```