

Unit-1

Verilog as HDL

Verilog has a variety of constructs as part of it. All are aimed at providing a functionally tested and a verified design description for the target FPGA or ASIC.

The language has a dual function – one fulfilling the need for a design description and the other fulfilling the need for verifying the design for functionality and timing constraints like propagation delay, critical path delay, slack, setup, and hold times.

Levels of Design Description

The components of the target design can be described at different levels with the help of the constructs in Verilog.

In Verilog HDL a module can be defined using various levels of abstraction. There are four levels of abstraction in verilog.

They are: 1. Circuit Level 2. Gate Level 3. Data Flow Level 4. Behavioral Level

Circuit Level

At the circuit level, a switch is the basic element with which digital circuits are built. Switches can be combined to form inverters and other gates at the next higher level of abstraction. Verilog has the basic MOS switches built into its constructs, which can be used to build basic circuits like inverters, basic logic gates, simple 1-bit dynamic and static memories. They can be used to build up larger designs to simulate at the circuit level, to design performance critical circuits.

The below Figure1 shows the circuit of an inverter suitable for description with the switch level constructs of Verilog.

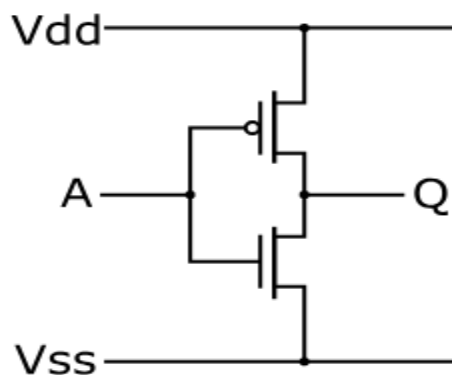
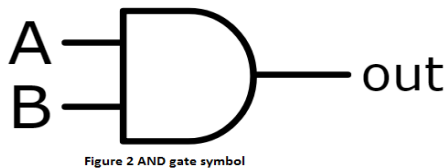


Figure 1 CMOS inverter

Gate Level

At the next higher level of abstraction, design is carried out in terms of basic gates. All the basic gates are available as ready modules called “Primitives.” Each such primitive is defined in terms of its inputs and outputs. Primitives can be incorporated into design descriptions directly. Just as full physical hardware can be built using gates, the primitives can be used repeatedly and judiciously to build larger systems.

Figure 2 shows an AND gate suitable for description using the gate primitive of Verilog.



The gate level modeling or structural modeling as it is sometimes called is akin to building a digital circuit on a bread board, or on a PCB. One should know the structure of the design to build the model here. One can also build hierarchical circuits at this level. However, beyond 20 to 30 of such gate primitives in a circuit, the design description becomes unwieldy; testing and debugging become laborious.

Data Flow

Data flow is the next higher level of abstraction. All possible operations on signals and variables are represented here in terms of assignments. All logic and algebraic operations are accommodated. The assignments define the continuous functioning of the concerned block. At the data flow level, signals are assigned through the data manipulating equations. All such assignments are concurrent in nature. The design descriptions are more compact than those at the gate level.

Figure 3 shows an A-O-I relationship suitable for description with the Verilog constructs at the data flow level.

$$e = \overline{a.b + c.d}$$

Figure : 3 An A-O-I gate represented as a data flow type of relationship.

Behavioral Level

Behavioral level constitutes the highest level of design description; it is essentially at the system level itself. With the assignment possibilities, looping constructs and conditional branching possible, the design description essentially looks like a “C” program.

A module can be implemented in terms of the design algorithm. The designer no need to have any knowledge of hardware implementation.

The statements involved are “dense” in function. Compactness and the comprehensive nature of the design description make the development process fast and efficient.

Figure 4 shows an A-O-I gate expressed in pseudo code suitable for description with the behavioral level constructs of Verilog.

<p>If (a, b, c or d changes) Compute e as $e = a.b + c.d$</p>

Figure . 4 An A-O-I gate in pseudo code at behavioral level.

The Overall Design Structure in Verilog

The possibilities of design description statements and assignments at different levels necessitate their accommodation in a mixed mode. In fact the design statements coexisting in a seamless manner within a design module is a significant characteristic of Verilog. Thus Verilog facilitates the mixing of the above-mentioned levels of design. A design built at data flow level can be instantiated to form a structural mode design. Data flow assignments can be incorporated in designs which are basically at behavioral level.

Concurrency

In an electronic circuit all the units are to be active and functioning concurrently. The voltages and currents in the different elements in the circuit can change simultaneously. In turn the logic levels too can change. Simulation of such a circuit in an HDL calls for concurrency of operation.

A number of activities – may be spread over different modules – are to be run concurrently here. Verilog simulators are built to simulate concurrency. (This is in contrast to programs in the normal languages like C where execution is sequential.)

Concurrency is achieved by proceeding with simulation in equal time steps. The time step is kept small enough to be negligible compared with the propagation delay values. All the activities scheduled at one time step are completed and then the simulator advances to the next time step and so on. The time step values refer to simulation time and not real time. One can redefine timescales to suit technology as and when necessary and carry out test runs.

In some cases the circuit itself may demand sequential operation as with data transfer and memory-based operations. Only in such cases sequential operation is ensured by the appropriate usage of sequential constructs from Verilog HDL.

Simulation and Synthesis

The design that is specified and entered as described earlier is simulated for functionality and fully debugged. Translation of the debugged design into the corresponding hardware circuit (using an FPGA or an ASIC) is called “synthesis.”

The tools available for synthesis relate more easily with the gate level and data flow level modules [Smith MJ]. The circuits realized from them are essentially direct translations of functions into circuit elements.

In contrast many of the behavioral level constructs are not directly synthesizable; even if synthesized they are likely to yield relatively redundant or wrong hardware. The way out is to take the behavioral level modules and redo each of them at lower levels. The process is carried out successively with each of the behavioral level modules until practically the full design is available as a pack of modules at gate and data flow levels (more commonly called the “RTL level”).

Programming Language Interface (PLI)

PLI provides an active interface to a compiled Verilog module. The interface adds a new dimension to working with Verilog routines from a C platform. The key functions of the interface are as follows:

- One can read data from a file and pass it to a Verilog module as input. Such data can be test vectors or other input data to the module. Similarly, variables in Verilog modules can be accessed and their values written to output devices.
- Delay values, logic values, etc., within a module can be accessed and altered.
- Blocks written in C language can be linked to Verilog modules.

MODULE

Any Verilog program begins with a keyword – called a “module.” A module is the name given to any system considering it as a black box with input and output terminals as shown in Figure 1. The terminals of the module are referred to as ‘ports’. The ports attached to a module can be of three types:

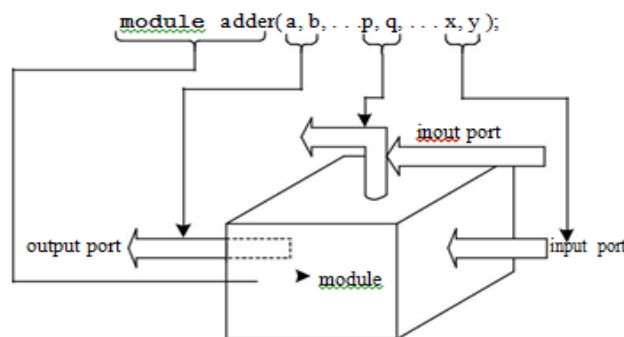


Figure 1 Representation of a module as black box with its ports.

- input ports through which one gets entry into the module; they signify the input signal terminals of the module.
- output ports through which one exits the module; these signify the output signal terminals of the module.
- inout ports: These represent ports through which one gets entry into the module or exits the module; These are terminals through which signals are input to the module sometimes; at some other times signals are output from the module through these.

Whether a module has any of the above ports and how many of each type are present depend solely on the functional nature of the module. Thus one module may not have any port at all; another may have only input ports, while a third may have only output ports, and so on.

All the constructs in Verilog are centered on the module. They define ways of building up, accessing, and using modules. The structure of modules and the mode of invoking them in a design are discussed here.

A module comprises a number of “lexical tokens” arranged according to some predefined order. The possible tokens are of seven categories:

- White spaces
- Comments
- Operators
- Numbers
- Strings
- Identifiers
- Keywords

The rules constraining the tokens and their sequencing will be dealt with as we progress. For the present let us consider modules. In Verilog any program which forms a design description is a “module.” Any program written to test a design description is also a “module.” The latter are often called as “stimulus modules” or “test benches.” A module used to do simulation has the form shown in Figure 2. Verilog takes the active statements appearing between the “module” statement and the “endmodule” statement and interprets all of them together as forming the body of the module. Whenever a module is invoked for testing or for incorporation into a bigger design module, the name of the module (“test” here) is used to identify it for the purpose.

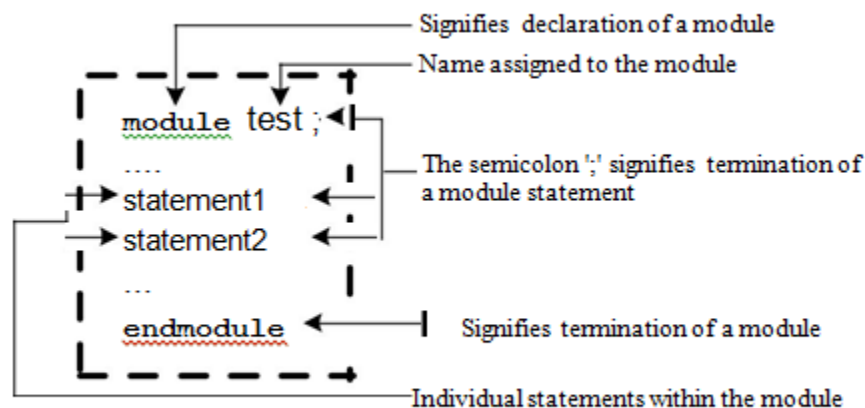


Figure 2 Structure of a typical simulation module.

LANGUAGE CONSTRUCTS AND CONVENTIONS IN VERILOG

Introduction

The constructs and conventions make up a software language. A clear understanding and familiarity of these is essential for the mastery of the language. Verilog has its own constructs and conventions [IEEE, Sutherland]. In many respects they resemble those of C language [Gottfried].

Any source file in Verilog (as with any file in any other programming language) is made up of a number of ASCII characters. The characters are grouped into sets — referred to as “lexical tokens.” A lexical token in Verilog can be a single character or a group of characters. Verilog has 7 types of lexical tokens- operators, keywords, identifiers, white spaces, comments, numbers, and strings.

Case Sensitivity

Verilog is a case-sensitive language like C. Thus sense, Sense, SENSE, sENse,... etc., are all related as different entities / quantities in Verilog.

Keywords

The keywords define the language constructs. A keyword signifies an activity to be carried out, initiated, or terminated. As such, a programmer cannot use a keyword for any purpose other than that it is intended for. All keywords in Verilog are in small letters and require to be used as such (since Verilog is a case-sensitive language). All keywords appear in the text in New Courier Bold-type letters.

Examples

module --	signifies the beginning of a module definition.
endmodule --	signifies the end of a module definition.
begin --	signifies the beginning of a block of statements.
end --	signifies the end of a block of statements.
if --	signifies a conditional activity to be checked
while --	signifies a conditional activity to be carried out.

Identifiers

Any program requires blocks of statements, signals, etc., to be identified with an attached nametag. Such nametags are identifiers. It is good practice for us to use identifiers, closely related to the significance of variable, signal, block, etc., concerned. This eases understanding and debugging of any program.

e.g., clock, enable, gate_1, . . .

There are some restrictions in assigning identifier names. All characters of the alphabet or an underscore can be used as the first character. Subsequent characters can be of alphanumeric type, or the underscore (`_`), or the dollar (`$`) sign – for example

`name`, `_name`. `Name`, `name1`, `name_$`, . . . -- all these are allowed as identifiers

`name aa` -- not allowed as an identifier because of the blank ("`name`" and "`aa`" are interpreted as two different identifiers)

`$name` -- not allowed as an identifier because of the presence of "`$`" as the first character.

`1_name` -- not allowed as an identifier, since the numeral "`1`" is the first character

`@name` -- not allowed as an identifier because of the presence of the character "`@`".

`A+b m` not allowed as an identifier because of the presence of the character "`+`".

White Space Characters

Blanks (`\b`), tabs (`\t`), newlines (`\n`), and formfeed form the white space characters in Verilog. In any design description the white space characters are included to improve readability. Functionally, they separate legal tokens. They are introduced between keywords, keyword and an identifier, between two identifiers, between identifiers and operator symbols, and so on. White space characters have significance only when they appear inside strings.

Comments

Comments can be inserted in the code for readability and documentation. There are two ways to write comments. A one-line comment starts with `"//"`. Verilog skips from that point to the end of line. A multiple-line comment starts with `"/*"` and ends with `"*/"`. Multiple-line comments cannot be nested. However, one-line comments can be embedded in multiple-line comments.

```
a = b && c; // This is a one-line comment
```

```
/* This is a multiple line
```

```
comment */
```

```
/* This is /* an illegal */ comment */
```

```
/* This is //a legal comment */
```

Operators

Operators are of three types: unary, binary, and ternary. Unary operators precede the operand. Binary operators appear between two operands. Ternary operators have two separate operators that separate three operands.

`a = ~ b;` // ~ is a unary operator. b is the operand

`a = b && c;` // && is a binary operator. b and c are operands

`a = b ? c : d;` // ?: is a ternary operator. b, c and d are operands

Number Specification

There are two types of number specification in Verilog: sized and unsized.

Sized numbers

Sized numbers are represented as `<size> '<base format> <number>`.

`<size>` is written only in decimal and specifies the number of bits in the number. Legal base formats are decimal ('d or 'D), hexadecimal ('h or 'H), binary ('b or 'B) and octal ('o or 'O). The number is specified as consecutive digits from 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f. Only a subset of these digits is legal for a particular base. Uppercase letters are legal for number specification.

<code>4'b1111</code>	//	This is a 4-bit	binary number
<code>12'habc</code>	//	This is a	12-bit hexadecimal number
<code>16'd255</code>	//	This is a	16-bit decimal number.

Unsized numbers

Numbers that are specified without a `<base format>` specification are decimal numbers by default. Numbers that are written without a `<size>` specification have a default number of bits that is simulator- and machine-specific (must be at least 32).

`23456` // This is a 32-bit 'hc3 // This is a 32-bit 'o21 // This is a 32-bit

decimal number by default hexadecimal number octal number

X or Z values

Verilog has two symbols for unknown and high impedance values. These values are very important for modeling real circuits. An unknown value is denoted by an x. A high impedance value is denoted by z.

12'h13x // This is a 12-bit hex number; 4 least significant bits unknown

6'hx // This is a 6-bit hex number

32'bz // This is a 32-bit high impedance number

An x or z sets four bits for a number in the hexadecimal base, three bits for a number in the octal base, and one bit for a number in the binary base. If the most significant bit of a number is 0, x, or z, the number is automatically extended to fill the most significant bits, respectively, with 0, x, or z. This makes it easy to assign x or z to whole vector. If the most significant digit is 1, then it is also zero extended.

Negative numbers

Negative numbers can be specified by putting a minus sign before the size for a constant number. Size constants are always positive. It is illegal to have a minus sign between <base format> and <number>. An optional signed specifier can be added for signed arithmetic.

-6'd3 // 8-bit negative number stored as 2's complement of 3 -6'sd3 // Used for performing signed integer math 4'd-2 // Illegal specification

Underscore characters and question marks

An underscore character "_" is allowed anywhere in a number except the first character. Underscore characters are allowed only to improve readability of numbers and are ignored by Verilog.

A question mark "?" is the Verilog HDL alternative for z in the context of numbers.

12'b1111_0000_1010 // Use of underline characters for readability

4'b10?? // Equivalent of a 4'b10zz

Strings

A string is a sequence of characters that are enclosed by double quotes. The restriction on a string is that it must be contained on a single line, that is, without a carriage return. It cannot be on multiple lines. Strings are treated as a sequence of one-byte ASCII values.

"Hello Verilog World" // is a string

"a / b" // is a string

Value Set or Logic Values

Verilog supports four values and eight strengths to model the functionality of real hardware. The four value levels are listed in Table below.

Value Level	Condition in Hardware Circuits
0	Logic zero, false condition
1	Logic one, true condition
x	Unknown logic value
Z	High impedance, floating state

Strengths

The logic levels are also associated with strengths. In many digital circuits, multiple assignments are often combined to reduce silicon area or to reduce pin-outs. To facilitate this, one can assign strengths to logic levels. Verilog has eight strength levels – four of these are of the driving type, three are of capacitive type and one of the hi-Z type.

In addition to logic values, strength levels are often used to resolve conflicts between drivers of different strengths in digital circuits. Value levels 0 and 1 can have the strength levels listed in Table below

Strength Level	Type	Degree
supply	Driving	strongest
strong	Driving	↑
pull	riving	
large	Storage	
weak	Driving	
medium	Storage	
small	Storage	weakest
highz	High Impedance	

If two signals of unequal strengths are driven on a wire, the stronger signal prevails.

For example, if two signals of strength strong1 and weak0 contend, the result is resolved as a strong1. If two signals of equal strengths are driven on a wire, the result is unknown. If two signals of strength strong1 and strong0 conflict, the result is an x. Strength levels are particularly useful for accurate modeling of signal contention, MOS devices, dynamic MOS, and other low-level devices.

Data Types

The data handled in Verilog fall into two categories:

- (i) Net data type
- (ii) Variable data type

The two types differ in the way they are used as well as with regard to their respective hardware structures. Data type of each variable or signal has to be declared prior to its use. The same is valid within the concerned block or module.

Nets

A net signifies a connection from one circuit unit to another. Such a net carries the value of the signal it is connected to and transmits to the circuit blocks connected to it. If the driving end of a net is left floating, the net goes to the high impedance state. A net can be specified in different ways.

wire: It represents a simple wire doing an interconnection. Only one output is connected to a wire and is driven by that.

tri: It represents a simple signal line as a wire. Unlike the wire, a tri can be driven by more than one signal outputs.

Nets are one-bit values by default unless they are declared explicitly as vectors. The terms wire and net are often used interchangeably.

Variable Data Type

A variable is an abstraction for a storage device. It can be declared through the keyword reg and stores the value of a logic level: 0, 1, x, or z. A net or wire connected to a reg takes on the value stored in the reg and can be used as input to other circuit elements. But the output of a circuit cannot be connected to a reg. The value stored in a reg is changed through a fresh assignment in the program.

time, integer, real, and realtime are the other variable types of data; these are dealt with later.

Time

Verilog simulation is done with respect to simulation time. A special time register data type is used in Verilog to store simulation time. A time variable is declared with the keyword time. The width for time register data types is implementation-specific but is at least 64 bits. The system function \$time is invoked to get the current simulation time.

```
time save_sim_time; // Define a time variable save_sim_time initial
```

```
save_sim_time = $time; // Save the current simulation time
```


Scalars and Vectors

Entities representing single bits — whether the bit is stored, changed, or transferred — are called “scalars.” Often multiple lines carry signals in a cluster — like data bus, address bus, and so on. Similarly, a group of regs stores a value, which may be assigned, changed, and handled together. The collection here is treated as a “vector.”

Figure below illustrates the difference between a scalar and a vector. `wr` and `rd` are two scalar nets connecting two circuit blocks `circuit1` and `circuit2`. `b` is a 4-bit-wide vector net connecting the same two blocks. `b[0]`, `b[1]`, `b[2]`, and `b[3]` are the individual bits of vector `b`. They are “part vectors.”

A vector reg or net is declared at the outset in a Verilog program and hence treated as such. The range of a vector is specified by a set of 2 digits (or expressions evaluating to a digit) with a colon in between the two. The combination is enclosed within square brackets.

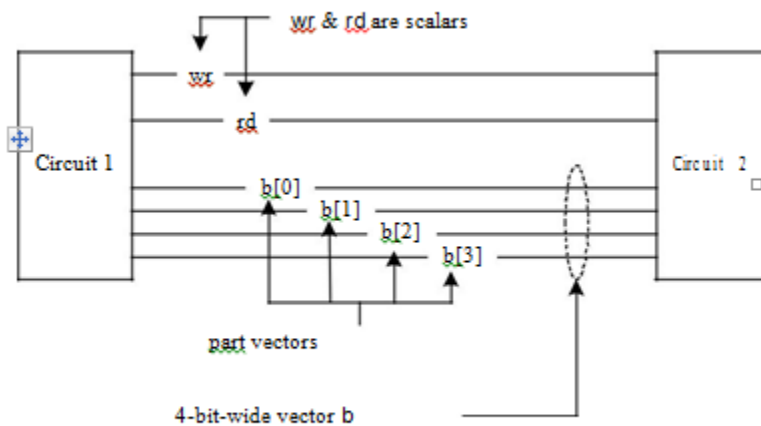


Figure Illustration of scalars and vectors.

Examples:

```
wire[3:0] a;    /* a is a four bit vector of net type; the bits are designated as a[3], a[2], a[1] and a[0]. */
```

```
reg[2:0] b;     /* b is a three bit vector of reg type; the bits are designated as b[2], b[1] and b[0]. */
```

```
reg[4:2] c;     /* c is a three bit vector of reg type; the bits are designated as c[4], c[3] and c[2]. */
```

```
wire[-2:2] d;   /* d is a 5 bit vector with individual bits designated as d[-2], d[-1], d[0], d[1] and d[2]. */
```

Whenever a range is not specified for a net or a reg, the same is treated as a scalar – a single bit quantity. In the range specification of a vector the most significant bit and the least significant bit can be assigned specific integer values. These can also be expressions evaluating to integer constants – positive or negative.

Normally vectors – nets or regs – are treated as unsigned quantities. They have to be specifically declared as “signed” if so desired.

Examples

```
wire signed[4:0] num; // num is a vector in the range -16 to +15.
```

```
reg signed [3:0] num_1; // num_1 is a vector in the range -8 to +7.
```

SYSTEM TASKS, FUNCTIONS, AND COMPILER DIRECTIVES

A number of facilities in Verilog relate to the management of simulation; starting and stopping of simulation, selectively monitoring the activities, testing the design for timing constraints, etc., are amongst them. Although a variety of such constructs is available in Verilog.

PARAMETERS

Verilog defines parameter as a constant value that is declared within structure of module. The constant value signifies timing values, range of variables, wires e.t.c.

The parameter values can be specified and changed to suit the design environment or test environment. Such changes are effected and frozen at instantiation.

The assigned values cannot change during testing or synthesis.

Two types of parameters are of use in modules: specparam and defparam.

Specparam : Parameters related to timings, time delays, rise and fall times, etc., are technology-specific and used during simulation. Parameter values can be assigned or overridden with the keyword “specparam” preceding the assignments.

Defparam: Parameters related to design, bus width, and register size are of a different category. They are related to the size or dimension of a specific design; they are technology-independent. Assignment or overriding is with assignments following the keyword “defparam”.

Timing-Related Parameters

The constructs associated with parameters are discussed here through specific design or test modules.

Example: Module of a half-adder with delays assigned to the output transitions; a test bench is also included in the figure.

```
module ha_1(s,ca,a,b);  
  input a,b; output s,ca;  
  xor #(1,2) (s,a,b);  
  and #(3,4) (ca,a,b);  
endmodule
```

```
//test-bench  
module tstha;  
  reg a,b; wire s,ca;  
  ha_1 hh(s,ca,a,b);  
  initial  
  begin
```

```

a=0;b=0;
end
always
begin
#5 a=1;b=0;
#5 a=0;b=1;
#5 a=1;b=1;
#5 a=0;b=0;
end
initial $monitor($time , " a = %b , b = %b ,out carry = %b , outsum = %b ",a,b,ca,s);
initial #30 $stop;
endmodule

```

Parameter Declarations and Assignments

Declaration of parameters in a design as well as assignments to them can be effected using the keyword “Parameter.” A declaration has the form

```
parameter alpha = a, beta = b;
```

Where

- parameter is the keyword,
- alpha and beta are the names assigned to two parameters and
- a, b are values assigned to alpha and beta, respectively.

In general a and b can be constant expressions. The parameter values can be overridden during instantiation but cannot be changed during the run-time. If a parameter assignment is made through the keyword “localparam,” its value cannot be overridden.

PATH DELAYS

The delay between source pin (input or inout) and destination pin (output or inout) of module is called module path delay.

Verilog has the provision to specify and check delays associated with total paths – from any input to any output of a module. Such paths and delays are at the chip or system level. They are referred to as “module path delays”.

Constructs available make room for specifying their paths and assigning delay values to them – separately or together.

Specify Blocks

Module paths are specified and values assigned to their delays through specify blocks. They are used to specify rise time, fall time, path delays pulse widths, and the like. A “specify” block can have the form shown in Figure

specify

```
specparam rise_time = 5, fall_time = 6;
```

```
(a => b) = (rise_time, fall_time);
```

```
(c => d) = (6, 7);
```

endspecify

The block starts with the keyword “specify” and ends with the keyword “endspecify”. Specify blocks can appear anywhere within a module.

Module Paths

Module Path delays are assigned in Verilog within the keywords specify and endspecify. The statements within these keywords constitute a specify block.

Module paths can be specified in different ways inside a specify block.

Parallel connection

Every path delay statement has a source field and a destination field.

A parallel connection is specified by the symbol => and is used as shown below.

Usage: (<source_field> => <destination_field>) = <delay_value>;

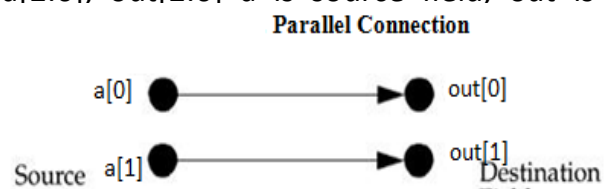
In a parallel connection, each bit in source field connects to its corresponding bit in the destination field.

If the source and the destination fields are vectors, they must have the same number of bits; otherwise, there is a mismatch. Thus, a parallel connection specifies delays from each bit in source to each bit in destination.

Example: Parallel Connection

(a => out) = 9; //bit-to-bit connection. Both a and out are single-bit

// vector connection. Both a and out are 4-bit vectors a[2:0], out[2:0] a is source field, out is destination field.



```
// for three bit-to-bit connection statements.
```

```
(a[0] => out[0]) = 9;
```

```
(a[1] => out[1]) = 9;
```

```
(a[2] => out[2]) = 9;
```

```
//illegal connection. a[4:0] is a 5-bit vector, out[3:0] is 4-bit.
```

```
//Mismatch between bit width of source and destination fields
```

```
(a => out) = 9; //bit width does not match.
```

Full connection

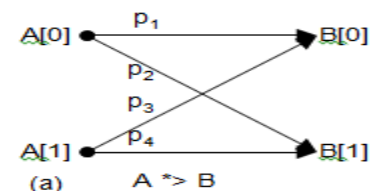
A full connection is specified by the symbol `*>` and is used as shown below.

Usage: (<source_field> `*>` <destination_field>) = <delay_value>;

In a full connection, each bit in the source field connects to every bit in the destination field. If the source and the destination are vectors, then they need not have the same number of bits. A full connection describes the delay between each bit of the source and every bit in the destination.

Example:

Figure below illustrates a case of all possible paths from a 2-bit vector A to another 2-bit vector B; the specification implies 4 pa



We can write the module M with pin-to-pin delays, using specify blocks as follows:

```
// Parallel connection
```

```
module M (out, a, b, c, d);
```

```
output out;
```

```
input a, b, c, d;
```

```
wire e, f;
```

```
//Specify block with path delay statements  
specify
```

```
(a => out) = 9;
```

```
(b => out) = 9;
```

```
(c => out) = 11;
```

```
(d => out) = 11;
```

```
endspecify
```

```
//gate instantiations
```

```
and a1(e, a, b);
```

```
and a2(f, c, d);
```

```
and a3(out, e, f);
```

```
endmodule
```

```
//Full Connection
```

```
module M (out, a, b, c, d);
```

```
output out;
```

```
input a, b, c, d;
```

```
wire e, f;
```

```
specify
```

```
(a,b *> out) = 9;
```

```
(c,d *> out) = 11;
```

```
endspecify
```

```
and a1(e, a, b);
```

```
and a2(f, c, d);
```

```
and a3(out, e, f);
```

```
endmodule
```

MODULE PARAMETERS

Module parameters are associated with size of bus, register, memory, ALU, and so on. They can be specified within the concerned module but their value can be altered during instantiation. The alterations can be brought about through assignments made with defparam. Such defparam assignments can appear anywhere in a module.

Example

The parameter msb specifies the ALU size — consistently in the input and the output vectors of the ALU. The size assignment has been made separately through the assignment statement

```
parameter msb = 3;
```

The ALU module with its size declared as a parameter.

```
module alu (d, co, a, b, f, cci);
  parameter msb=3;
  output [msb:0] d; output co;
  wire[msb:0]d;
  input cci;
  input [msb : 0 ] a, b;
  input [1 : 0] f;
  specify
    (a,b=>d)=(1,2);
    (a,b,cci*>co)=1;
  endspecify
  assign {co,d}= (f==2'b00)?(a+b+cci):((f==2'b01)?(a-b):((f==2'b10)?{1'bz,a^b}:{1'bz,~a}));
endmodule
```

SYSTEM TASKS AND FUNCTIONS

Verilog has a number of System Tasks and Functions defined in the LRM (language reference manual).

They are for taking output from simulation, control simulation, debugging design modules, testing modules for specifications, etc.

A “\$” sign preceding a word or a word group signifies a system task or a system function.

Output Tasks

A number of system tasks are available to output values of variables and selected messages, etc., on the monitor. Out of these \$monitor and \$display tasks have been extensively used.

Display Tasks

The **\$display** task, whenever encountered, displays the arguments in the desired format; and the display advances to a new line.

\$strobe Task:

When a variable or a set of variables is sampled and its value displayed, the \$strobe task can be used; it senses the value of the specified variables and displays them.

The \$strobe task is executed as the last activity in the concerned time step. It is useful to check for specific activities and debug modules.

Example:

```
initial #9 $strobe ("at time %t, di=%b, do=%b", $time, di, do);
```

\$monitor Task:

\$monitor task is activated and displays the arguments specified whenever any of the arguments changes.

\$stop and \$finish Tasks:

The \$stop task suspends simulation. The compiled design remains active; simulation can be resumed through commands available in the simulator.

In contrast \$finish stops simulation, closes the simulation environment, and reverts to the operating system.

\$random Function:

A set of random number generator functions are available as system functions.

One can start with a seed number (optional) and generate a random number repeatedly. Such random number sequences can be fruitfully used for testing.

Compiler directives

Compiler directives are special commands, beginning with ```, that affect the operation of the Verilog simulator.

Time Scale

``timescale` specifies the time unit and time precision. A time unit of 10 ns means a time expressed as say #2.3 will have a delay of 23.0 ns. Time precision specifies how delay values are to be rounded off during simulation. Valid time units include s, ms, us (μ s), ns, ps, fs.

Only 1, 10 or 100 are valid integers for specifying time units or precision. It also determines the displayed time units in display commands like `$display`.

Syntax

```
`timescale time_unit / time_precision;
```

Examples

```
`timescale 1 ns/1 ps // unit = 1ns, precision = 1/1000ns
```

```
`timescale 1 ns /100 ps // time unit = 1ns; precision = 1/10ns;
```

``define`

A macro is an identifier that represents a string of text. Macros are defined with the directive ``define`, and are invoked with the quoted macro name as shown in the example. Verilog compilers will substitute the string for the macro name before starting compilation. Many people prefer to use macros instead of parameters.

The define directive in Verilog is similar to `#define` in c-language.

Syntax

```
`define macro_name text_string;
... `macro_name ...
```

Example

```
`define add_lsb a[7:0] + b[7:0]
`define N 8 // Word length
wire [N-1:0] S;
assign S = 'add_lsb; // assign S = a[7:0] + b[7:0];
```

Include Directive

Include is used to include the contents of a text file at the point in the current file where the include directive is. The include directive is similar to the C/C++ include directive.

Syntax

```
`include file_name;
```

Example

```
module x;
`include "dclr.v"; // contents of file "dclr.v" are put here
```

USER-DEFINED PRIMITIVES (UDP):

The primitives available in Verilog are all of the gate or switch types. Verilog has the provision for the user to define primitives –called “user defined primitive (UDP)” and use them.

The designers occasionally like to use their own custom-built primitives when developing a design. Verilog provides the ability to define User- Defined Primitives (UDP). These primitives are self-contained and do not instantiate other modules or primitives. UDPs are instantiated exactly like gate-level primitives.

UDPs are basically of two types –combinational and sequential. A combinational UDP is used to define a combinational scalar function and a sequential UDP for a sequential function.

Combinational UDPs:

A combinational UDP accepts a set of scalar inputs and gives a scalar output. An inout declaration is not supported by a UDP. The UDP definition is on par with that of a module; that is, it is defined independently like a module and can be used in any other module.

```
primitive udp_and(out, a, b);
output out;
input a, b;
table
    // a b: Out;
    0 0: 0;
    0 1: 0;
    1 0: 0;
    1 1: 1;
endtable
endprimitive
```

Sequential UDPs:

Any sequential circuit has a set of possible states. When it is in one of the specified states, the next state to be taken is described as a function of the input logic variables and the present state. A sequential UDP can accommodate all these.

```
primitive latch(q, d, clock, clear); // d-latch
output q;
reg q; //q declared as reg to create internal storage input d, clock, clear;
initial q = 0; //initialize output to value 0
table //state table
//d clock clear: q : q+ ;
?? 1 : ? : 0 ; //clear condition;
1 1 0 : ? : 1; //latchq =data=1
0 1 0 : ? : 0; //latchq =data=0
? 0 0 : ? : - ; //retain original state if clock = 0
endtable
endprimitive
```

Operators

Operators act on the operands to produce desired results. Verilog provides various types of operators.

d1 && d2 // && is an operator on operands d1 and d2
!a[0]
// ! is an operator on operand a[0]

B >> 1 // >> is an operator on operands B and 1

Operator Types

Verilog provides many different operator types. Operators can be arithmetic, logical, relational, equality, bitwise, reduction, shift, concatenation, or conditional. Some of these operators are similar to the operators used in the C programming language. Each operator type is denoted by a symbol. The following table shows the complete listing of operator symbols classified by category.

Table: Operator Types and Symbols

Operator Type	Operator Symbol	Operation Performed	Number of Operands
Arithmetic	*	multiply	two
	/	divide	two
	+	add	two
	-	subtract	two
	%	modulus	two
	**	power (exponent)	two
Logical	!	logical negation	one
	&&	logical and	two
		logical or	two
	>	greater than	two

Relational	<	less than	two
	>=	greater than or equal	two
	<=	less than or equal	two
Equality	==	equality	two
	!=	inequality	two
	===	case equality	two
	!==	case inequality	two

Bitwise	~	bitwise negation	one
	&	bitwise and	two
		bitwise or	two
	^	bitwise xor	two
	^~ or ~^	bitwise xnor	two
Reduction	&	reduction and	one
	~&	reduction nand	one
		reduction or	one
	~	reduction nor	one
	^	reduction xor	one
	^~ or ~^	reduction xnor	one

Shift	>>	Right shift	Two
	<<	Left shift	Two
	>>>	Arithmetic right shift	Two
	<<<	Arithmetic left shift	Two
Concatenation	{ }	Concatenation	Any number
Replication	{ { } }	Replication	Any number
Conditional	?:	Conditional	Three

Let us now discuss each operator type in detail.

Arithmetic Operators

There are two types of arithmetic operators: binary and unary.

Binary operators

Binary arithmetic operators are multiply (*), divide (/), add (+), subtract (-), power (**), and modulus (%). Binary operators take two operands.

A = 4'b0011; B = 4'b0100; // A and B are register vectors
D = 6; E = 4; F=2// D and E are integers

A * B // Multiply A and B. Evaluates to 4'b1100

D / E // Divide D by E. Evaluates to 1. Truncates any fractional part.
A + B // Add A and B. Evaluates to 4'b0111

`B - A // Subtract A from B. Evaluates to 4'b0001`
`F = E **`
`F; //E to the power F, yields 16`

If any operand bit has a value x, then the result of the entire expression is x. This seems intuitive because if an operand value is not known precisely, the result should be an unknown.

`in1 = 4'b101x;`

`in2 = 4'b1010;`

`sum = in1 + in2; // sum will be evaluated to the value 4'bx`

Modulus operators produce the remainder from the division of two numbers. They operate similarly to the modulus operator in the C programming language.

`13 % 3 // Evaluates to 1`

`16 % 4 // Evaluates to 0`

`-7 % 2 // Evaluates to -1, takes sign of the first operand`

`7 % -2 // Evaluates to +1, takes sign of the first operand`

Unary operators

The operators + and - can also work as unary operators. They are used to specify the positive or negative sign of the operand. Unary + or ? operators have higher precedence than the binary + or ? operators.

`-4 // Negative 4`

`+5 // Positive 5`

Negative numbers are represented as 2's complement internally in Verilog. It is advisable to use negative numbers only of the type integer or real in expressions. Designers should avoid negative numbers of the type <sss> '<base> <nnn> in expressions because they are converted to unsigned 2's complement numbers and hence yield unexpected results.

//Advisable to use integer or real numbers -10 /

5// Evaluates to -2

//Do not use numbers of type <sss> '<base> <nnn>

-'d10 / 5// Is equivalent $(2^{\text{word width}} - 10)/5$

where 32 is the default machine word width.

This evaluates to an incorrect and unexpected result

Logical Operators

Logical operators are logical-and (&&), logical-or (||) and logical- not (!). Operators && and || are binary operators. Operator ! is a unary operator. Logical operators follow these conditions:

Logical operators always evaluate to a 1-bit value, 0 (false), 1 (true), or x (ambiguous). If an operand is not equal to zero, it is equivalent to a logical 1 (true condition). If it is equal to zero, it is equivalent to a logical 0 (false condition). If any operand bit is x or z, it is equivalent to x (ambiguous condition) and is normally treated by simulators as a false condition. Logical operators take variables or expressions as operands. Use of parentheses to group logical operations is highly recommended to improve readability. Also, the user does not have to remember the precedence of operators.

Logical operations A = 3;

B = 0;

A && B // Evaluates to 0. Equivalent to (logical-1 && logical-0) A || B //

Evaluates to 1. Equivalent to (logical-1 || logical-0) !A// Evaluates to 0.

Equivalent to not(logical-1)

!B// Evaluates to 1. Equivalent to not(logical-0)

Unknowns

A = 2'b0x; B = 2'b10;

A && B // Evaluates to x. Equivalent to (x && logical 1)

// Expressions

(a == 2) && (b == 3) // Evaluates to 1 if both a == 2 and b == 3 are true.

// Evaluates to 0 if either is false.

Relational Operators

Relational operators are greater-than (>), less-than (<), greater-than-or-equal-to (>=), and less-than-or-equal-to (<=). If relational operators are used in an expression, the expression returns a logical value of 1 if the expression is true and 0 if the expression is false. If there are any unknown or z bits in the operands, the expression takes a value x. These operators function exactly as the corresponding operators in the C programming language.

A = 4, B = 3

X = 4'b1010, Y = 4'b1101, Z = 4'b1xxx

A <= B // Evaluates to a logical 0

A > B // Evaluates to a logical 1

Y >= X // Evaluates to a logical 1

Y < Z // Evaluates to an x

Equality Operators

Equality operators are logical equality (==), logical inequality (!=), case equality (===), and case inequality (!==). When used in an expression, equality operators return logical value 1 if true, 0 if false. These operators compare the two operands bit by bit, with zero filling if the operands are of unequal length. Table below lists the operators.

It is important to note the difference between the logical equality operators (==, !=) and case equality operators (===, !==). The logical equality operators (==, !=) will yield an x if either operand has x or z in its bits. However, the case equality operators (===, !==) compare both operands bit by bit and compare all bits, including x and z. The result is 1 if the operands match exactly, including x and z bits. The result is 0 if the operands do not match exactly. Case equality operators never result in an x.

Table: Equality Operators

Expression	Description	Possible Logical Value
<code>a == b</code>	a equal to b, result unknown if x or z in a or b	0, 1, x
<code>a != b</code>	a not equal to b, result unknown if x or z in a or b	0, 1, x
<code>a === b</code>	a equal to b, including x and z	0, 1
<code>a !== b</code>	a not equal to b, including x and z	0, 1

`A = 4, B = 3`

`X = 4'b1010, Y = 4'b1101`

`Z = 4'b1xxz, M = 4'b1xxz, N = 4'b1xxx`

`A == B` // Results in logical 0

`X != Y` // Results in logical 1

`X == Z` // Results in x

`Z === M` // Results in logical 1 (all bits match, including x and z)

`Z === N` // Results in logical 0 (least significant bit does not match) `M !== N` // Results in logical 1

Bitwise Operators

Bitwise operators are negation (~), and(&), or (|), xor (^), xnor (^~, ~^). Bitwise operators perform a bit-by-bit operation on two operands. They take each bit in one operand and perform the operation with the corresponding bit in the other operand. If one operand is shorter than the other, it will be bit-extended with zeros to match the length of the longer operand. Logic tables for the bit-by-bit computation are shown in Table. A z is treated as an x in a bitwise operation. The exception is the unary negation operator (~), which takes only one operand and operates on the bits of the single operand.

Table: Truth Tables for Bitwise Operators

bitwise and	0	1	x
0	0	0	0
1	0	1	x
x	0	x	x

bitwise or	0	1	x
0	0	1	x
1	1	1	1
x	x	1	x

bitwise xor	0	1	x
0	0	1	x
1	1	0	x
x	x	x	x

bitwise xnor	0	1	x
0	1	0	x
1	0	1	x
x	x	x	x

bitwise negation	result
0	1
1	0
x	x

Examples of bitwise operators are shown below.

```
X = 4'b1010, Y = 4'b1101
```

```
Z = 4'b10x1
```

```
~X    // Negation. Result is 4'b0101
```

```
X & Y  // Bitwise and. Result is 4'b1000
```

```
X | Y  // Bitwise or. Result is 4'b1111
```

```
X ^ Y // Bitwise xor. Result is 4'b0111
```

```
X ^^ Y // Bitwise xnor. Result is 4'b1000
```

```
X & Z // Result is 4'b10x0
```

It is important to distinguish bitwise operators \sim , $\&$, and $|$ from logical operators $!$, $\&\&$, $||$. Logical operators always yield a logical value 0, 1, x, whereas bitwise operators yield a bit-by-bit value. Logical operators perform a logical operation, not a bit-by-bit operation.

```
// X = 4'b1010, Y = 4'b0000
```

```
X | Y // bitwise operation. Result is 4'b1010
```

```
X || Y // logical operation. Equivalent to 1 || 0. Result is 1.
```

Reduction Operators

Reduction operators are and ($\&$), nand ($\sim\&$), or ($|$), nor ($\sim|$), xor (\wedge), and xnor ($\sim\wedge$, $\wedge\sim$). Reduction operators take only one operand. Reduction operators perform a bitwise operation on a single vector operand and yield a 1-bit result. The difference is that bitwise operations are on bits from two different operands, whereas reduction operations are on the bits of the same operand. Reduction operators work bit by bit from right to left. Reduction nand, reduction nor, and reduction xnor are computed by inverting the result of the reduction and, reduction or, and reduction xor, respectively.

```
// X = 4'b1010
```

```
&X //Equivalent to 1 & 0 & 1 & 0. Results in 1'b0
```

```
|X //Equivalent to 1 | 0 | 1 | 0. Results in 1'b1
```

```
^X //Equivalent to 1 ^ 0 ^ 1 ^ 0. Results in 1'b0
```

```
//A reduction xor or xnor can be used for even or odd parity
```

```
//generation of a vector.
```

The use of a similar set of symbols for logical (!, &&, ||), bitwise (~, &, |, ^), and reduction operators (&, |, ^) is somewhat confusing initially. The difference lies in the number of operands each operator takes and also the value of results computed.

Shift Operators

Shift operators are right shift (>>), left shift (<<), arithmetic right shift (>>>), and arithmetic left shift (<<<). Regular shift operators shift a vector operand to the right or the left by a specified number of bits. The operands are the vector and the number of bits to shift. When the bits are shifted, the vacant bit positions are filled with zeros. Shift operations do not wrap around. Arithmetic shift operators use the context of the expression to determine the value with which to fill the vacated bits.

```
// X = 4'b1100
```

```
Y = X >> 1; //Y is 4'b0110. Shift right 1 bit. 0 filled in MSB position.
```

```
Y = X << 1; //Y is 4'b1000. Shift left 1 bit. 0 filled in LSB position.
```

```
Y = X << 2; //Y is 4'b0000. Shift left 2 bits.
```

```
integer a, b, c; //Signed data types
```

```
a = 0;
```

```
b = -10; // 00111...10110 binary
```

```
c = a + (b >>> 3); //Results in -2 decimal, due to arithmetic shift
```

Shift operators are useful because they allow the designer to model shift operations, shift-and-add algorithms for multiplication, and other useful operations.

Concatenation Operator

The concatenation operator ({, }) provides a mechanism to append multiple operands. The operands must be sized. Unsized operands are not allowed because the size of each operand must be known for computation of the size of the result. Concatenations are expressed as operands within braces, with commas separating the operands. Operands can be scalar nets or registers, vector nets or registers, bit-select, part-select, or sized constants.

```
// A = 1'b1, B = 2'b00, C = 2'b10, D = 3'b110
Y = {B , C} // Result Y is 4'b0010
Y = {A , B , C , D , 3'b001} // Result Y is 11'b10010110001
Y = {A , B[0], C[1]} // Result Y is 3'b101
```

Replication Operator

Repetitive concatenation of the same number can be expressed by using a replication constant. A replication constant specifies how many times to replicate the number inside the brackets ({ }).

```
reg A;
```

```
reg [1:0] B, C;
```

```
reg [2:0] D;
```

```
A = 1'b1; B = 2'b00; C = 2'b10; D = 3'b110;
```

```
Y = { 4{A} } // Result Y is 4'b1111
```

```
Y = { 4{A} , 2{B} } // Result Y is 8'b11110000
```

```
Y = { 4{A} , 2{B} , C } // Result Y is 8'b1111000010
```

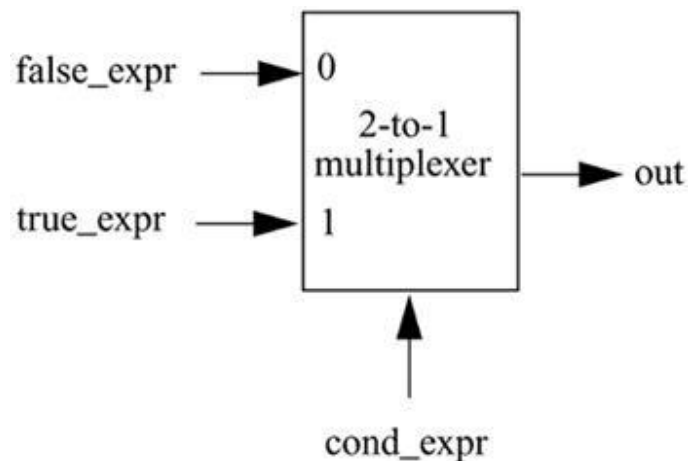
Conditional Operator

The conditional operator(?:) takes three operands.

Usage: condition_expr ? true_expr : false_expr ;

The condition expression (condition_expr) is first evaluated. If the result is true (logical 1), then the true_expr is evaluated. If the result is false (logical 0), then the false_expr is evaluated. If the result is x (ambiguous), then both true_expr and false_expr are evaluated and their results are compared, bit by bit, to return for each bit position an x if the bits are different and the value of the bits if they are the same.

The action of a conditional operator is similar to a multiplexer. Alternately, it can be compared to the if-else expression.



Conditional operators are frequently used in dataflow modeling to model conditional assignments. The conditional expression acts as a switching control.

```
//model functionality of a tristate buffer
```

```
assign addr_bus = drive_enable ? addr_out : 36'bz;
```

```
//model functionality of a 2-to-1 mux
```

```
assign out = control ? in1 : in0;
```

Conditional operations can be nested. Each `true_expr` or `false_expr` can itself be a conditional operation. In the example that follows, convince yourself that $(A==3)$ and `control` are the two select signals of 4-to-1 multiplexer with `n`, `m`, `y`, `x` as the inputs and `out` as the output signal.

```
assign out = (A == 3) ? ( control ? x : y ) : ( control ? m : n );
```

Operator Precedence

Having discussed the operators, it is now important to discuss operator precedence. If no parentheses are used to separate parts of expressions, Verilog enforces the following precedence. Operators listed in Table are in order from highest precedence to lowest precedence. It is recommended that parentheses be used to separate expressions except in case of unary operators or when there is no ambiguity.

Table: Operator Precedence

Operators	Operator Symbols	Precedence
Unary	+ - ! ~	Highest precedence
Multiply, Divide, Modulus	* / %	
Add, Subtract	+ -	
Shift	<< >>	
Relational	< <= > >=	
Equality	== != === !==	
Reduction	&, ~& ^ ^~ , ~	
Logical	&& 	
Conditional	?:	Lowest precedence

Testbench

- Test benches are used to simulate your design without the need of any physical hardware.
- A test bench is actually just another Verilog file! However, the Verilog you write in a test bench is not quite the same as the Verilog you write in your designs
- If the number of input signals are very large and/or we have to perform simulation several times, then this process can be quite complex, time consuming and irritating.
- with the help of testbenches, we can generate results in the form of csv (comma separated file), which can be used by other softwares for further analysis e.g. Python, Excel and Matlab etc.

Procedure

- Testbenches are written in separate Verilog files
- A test bench starts off with a module declaration
- A testbench with name 'half_adder_tb
- Ports of the testbench is always empty i.e. no inputs or outputs are defined
- After we declare our variables, we instantiate the module we will be testing
- 'Initial block' is used , which is executed only once, and terminated when the last line of the block executed
- DUT is a very common name for the module to be tested in a test bench

Half adder

```
Module half_adder( input wire a, b,  
Output wire sum, carry);
```

```
assign sum = a ^ b;  
assign carry = a & b;
```

```
endmodule
```

Half adder test bench

```
module half_adder_tb;
```

```
  reg a, b;
```

```
  wire sum, carry;
```

```
  localparam period = 20;
```

```
  half_adder UUT (.a(a), .b(b), .sum(sum), .carry(carry));
```

```
  initial // initial block executes only once
```

```
  begin // values for a and b
```

```
    a = 0; b = 0;
```

```
    #period; // wait for period
```

```
    a = 0; b = 1;
```

```
    #period;
```

```
    a = 1; b = 0;
```

```
    #period;
```

```
    a = 1; b = 1;
```

```
    #period;
```

```
  end
```

```
endmodule
```

Jk flipflop

```
module jkff_behave(clk,j,knq,qbar);  
input clk,j,k;  
output reg q,qbar;  
always@(posedge clk)  
begin  
    if(k = 0)  
        begin  
            q <= 0;  
            qbar <= 1;  
        end  
end
```

```
always@(posedge clk)
```

```
begin
```

```
  if(k = 0)
```

```
    begin
```

```
      q <= 0;
```

```
      qbar <= 1;
```

```
    end
```

```
  else if(j = 1)
```

```
    begin
```

```
      q <= 0;
```

```
      qbar <= 0;
```

```
    end
```

```
Else if(j = 0 & k = 0)
  begin
    q <= q;
    qbar <= qbar;
  end
else if(j = 1 & k = 1)
  begin
    q <= ~q;
    qbar <= ~qbar;
  end
end
endmodule
```

Using case statement

```
module JKFlipFlop( input J,input K,input clk,output Q, output Qbar );
reg Q,Qbar;
always@(posedge clk)
begin
    case({J,K})
        2'b00:Q<=Q;
        2'b01:Q<=1'b0;
        2'b10:Q<=1'b1;
        2'b11:Q<=Qbar;
    endcase
end
endmodule
```


Test Bench

```
module JK_FlipFlop_TB;  
    // Inputs  
    reg J;  
    reg K;  
    // Outputs  
    wire Q;  
    wire Qbar;  
    // Instantiate the Unit Under Test (UUT)  
    JKFlipFlop uut ( .J(J), .K(K), .Q(Q),.Qbar(Qbar) );
```

```
initial begin
    // Initialize Inputs
    clk=0;
    forever #5 clk=~clk
        #100 J = 0; K = 0;
        #100 J=0; K=1;
        #100 J=1;k=0;
        #100 J=1; K=1;
    end
endmodule
```

Up counter design

```
Module up_counter(input clk, reset, output[3:0] counter
);
reg [3:0] counter_up;
```

```
// up counter
always @(posedge clk or posedge reset)
begin
if(reset)
    counter_up <= 4'd0;
else
    counter_up <= counter_up + 4'd1;
end
assign counter = counter_up;
endmodule
```

Test bench

```
Module upcounter_testbench();  
reg clk, reset;  
wire [3:0] counter;  
up_counter dut(clk, reset, counter);  
initial begin  
    clk=0;  
    forever #5 clk=~clk;  
end  
initial begin  
    reset=1;  
    #20;  
    reset=0;  
end  
endmodule
```

Unit-2

Gate Level Modeling

Introduction

Digital designers are normally familiar with all the common logic gates, their symbols, and their working. Flip-flops are built from the logic gates. All other functionally complex and more involved circuits can also be built using the basic gates. All the basic gates are available as “Primitives” in Verilog. Primitives are generalized modules that already exist in Verilog [IEEE]. They can be instantiated directly in other modules.

And Gate Primitive

The AND gate primitive in Verilog is instantiated with the following statement:

and g1 (O, I1, I2, . . . , In);

Here ‘and’ is the keyword signifying an AND gate. g1 is the name assigned to the specific instantiation. O is the gate output; I1, I2, etc., are the gate inputs. The following are noteworthy:

- The AND module has only one output. The first port in the argument list is the output port.
- An AND gate instantiation can take any number of inputs — the upper limit is compiler-specific.
- A name need not be necessarily assigned to the AND gate instantiation; this is true of all the gate primitives available in Verilog.

Truth Table of AND Gate Primitive

The truth table for a two-input AND gate is shown in Table below. It can be directly extended to AND gate instantiations with multiple inputs. The following observations are in order here:

Truth table of AND gate primitive

		Input 1			
		0	1	X	z
Input 2	0	0	0	0	0
	1	0	1	X	x
	x	0	x	X	x
	z	0	x	X	x

- If any one of the inputs to the AND gate instantiation is in the 0 state, its output is also in the 0 state. It is irrespective of whether the other inputs are at the 0, 1, x or z state.
- The output is at 1 state if and only if every one of the inputs is at 1 state.
- For all other cases the output is at the x state.
- Note that the output is never at the z state – the high impedance state. This is true of all other gate primitives as well.

Module Structure

In a general case a module can be more elaborate. A lot of flexibility is available in the definition of the body of the module. However, a few rules need to be followed:

- The first statement of a module starts with the keyword module; it may be followed by the name of the module and the port list if any.
- All the variables in the ports-list are to be identified as inputs, outputs, or inout. The corresponding declarations have the form shown below:

```
? Input a1, a2;
? Output b1, b2;
? Inout c1, c2;
```

The port-type declarations here follow the module declaration mentioned above.

- The ports and the other variables used within the body of the module are to be identified as nets or registers with specific types in each case. The respective declaration statements follow the port-type declaration statements.

Examples:

```
wire a1, a2, c;
reg b1, b2;
```

The type declaration must necessarily precede the first use of any variable or signal in the module.

- The executable body of the module follows the declaration indicated above.
- The last statement in any module definition is the keyword “endmodule”.
- Comments can appear anywhere in the module definition.

Other Gate Primitives

All other basic gates are also available as primitives in Verilog. Details of the facilities and instantiations in each case are given in Table below. The following points are noteworthy here:

- In all cases of instantiations, one need not necessarily assign a name to the instantiation. It need be done only when felt necessary – say for clarity of circuit description.
- In all the cases the output port(s) is (are) declared first and the input port(s) is (are) declared subsequently.
- The buffer and the inverter have only one input each. They can have any number of outputs; the upper limit is compiler-specific. All other gates have one output each but can have any number of inputs; the upper limit is again compiler-specific.

Table for Basic gate primitives in Verilog with details

Gate	Mode of instantiation	Output port(s)	Input port(s)
AND	and ga (o, i1, i2, . . . i8);	o	i1, i2, . .
OR	or gr (o, i1, i2, . . . i8);	o	i1, i2, . .
NAND	nand gna (o, i1, i2, . . . i8);	o	i1, i2, . .
NOR	nor gnr (o, i1, i2, . . . i8);	o	i1, i2, . .
XOR	xor gxr (o, i1, i2, . . . i8);	o	i1, i2, . .
XNOR	xnor gxn (o, i1, i2, . . . i8);	o	i1, i2, . .
BUF	buf gb (o1, o2, i);	o1, o2, o3, . .	i
NOT	not gn (o1, o2, o3, . . . i);	o1, o2, o3, . .	i

Example for a typical A-O-I gate circuit

The commonly used A-O-I gate is shown in Figure 1 for a simple case. The module and the test bench for the same are given in Figure 2. The circuit has been realized here by instantiating the AND and NOR gate primitives. The names of signals and gates used in the instantiations in the module of Figure 2 remain the same as those in the circuit of Figure 1. The module `aoi_gate` in the figure has input and output ports since it describes a circuit with signal inputs and an output. The module `aoi_st` is a stimulus module. It generates inputs to the `aoi_gate` module and gets its output. It has no input or output ports.

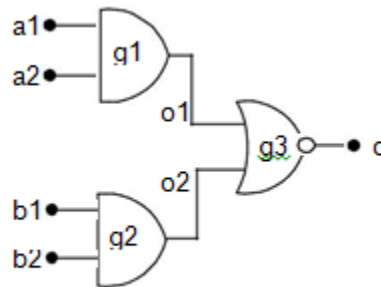


Figure for a typical A-O-I gate circuit.

```
/*module for the aoi-gate of figure 1 instantiating the gate primitives – fig 2*/
module aoi_gate(o,a1,a2,b1,b2);

input a1,a2,b1,b2;    // a1,a2,b1,b2 form the input //ports of the module

output o;             //o is the single output port of the module

wire o1,o2;           //o1 and o2 are intermediate signals //within the module

and g1(o1,a1,a2); //The AND gate primitive has two and g2(o2,b1,b2);

                    // instantiations with assigned //names g1 & g2.

nor g3(o,o1,o2); //The nor gate has one instantiation with assigned name g3.

endmodule

//Test-bench for the aoi_gate above
module aoi_st;
reg a1,a2,b1,b2;

//specific values will be assigned to a1,a2,b1, // and b2 and these connected
//to input ports of the gate insatntiations;
```



```
//hence these variables are declared as reg
wire o;
initial
begin
a1 = 0;
a2 = 0;
b1 = 0;
b2 = 0;
#3 a1 = 1;
#3 a2 = 1;
#3 b1 = 1;
#3 b2 = 0;
#3 a1 = 1;
#3 a2 = 0;
#3 b1 = 0;
end
initial #100 $stop;//the simulation ends after //running for 100 tu's.
initial $monitor($time , " o = %b , a1 = %b , a2 = %b , b1 = %b ,b2 = %b ",o,a1,a2,b1,b2);
aoi_gate gg(o,a1,a2,b1,b2);
endmodule
```

Tri-State Gates

Four types of tri-state buffers are available in Verilog as primitives. Their outputs can be turned ON or OFF by a control signal. The direct buffer is instantiated as
 Bufif1 nn (out, in, control);

The symbol of the buffer is shown in Figure

1. We have

- out as the single output variable
- in as the single input variable and
- control as the single control signal variable.

When
 control = 1,
 out = in.

When
 control = 0,
 out=tri-stated

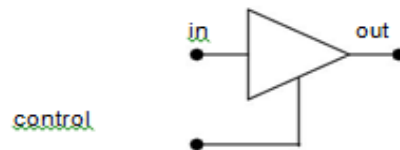
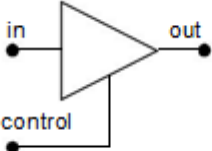
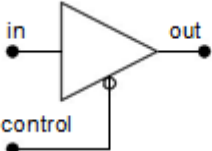
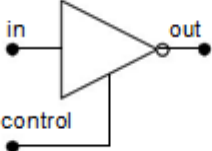
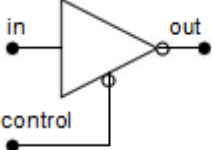


Figure 1 A tri-state buffer.

out is cut off from the input and tri-stated. The output, input and control signals should appear in the instantiation in the same order as above. Details of buif1 as well as the other tri-state type primitives are shown in Table 1.

In all the cases shown in Table 1, out is the output; in is the input, and control, the control variable.

Table 1 Instantiation and functional details of tri-state buffer primitives

Typical instantiation	Functional representation	Functional description
<code>bufif1 (out, in, control);</code>		Out = in if control = 1; else out = z
<code>bufif0 (out, in, control);</code>		Out = in if control = 0; else out = z
<code>notif1 (out, in, control);</code>		Out = complement of in if control = 1; else out = z
<code>notif0 (out, in, control);</code>		Out = complement of in if control = 0; else out = z

Array of Instances of Primitives

The primitives available in Verilog can also be instantiated as arrays. A judicious use of such array instantiations often leads to compact design descriptions. A typical array instantiation has the form

```
and gate [7 : 4 ] (a, b, c);
```

where a, b, and c are to be 4 bit vectors. The above instantiation is equivalent to combining the following 4 instantiations:

```
and gate [7] (a[3], b[3], c[3]), gate [6] (a[2], b[2], c[2]), gate [5] (a[1], b[1], c[1]), gate [4] (a[0], b[0], c[0]);
```

The assignment of different bits of input vectors to respective gates is implicit in the basic declaration itself. A more general instantiation of array type has the form

```
and gate[mm : nn](a, b, c);
```

where mm and nn can be expressions involving previously defined parameters, integers and algebra with them. The range for the gate is $1+(mm-nn)$; mm and nn do not have restrictions of sign; either can be larger than the other.

Gate Delays

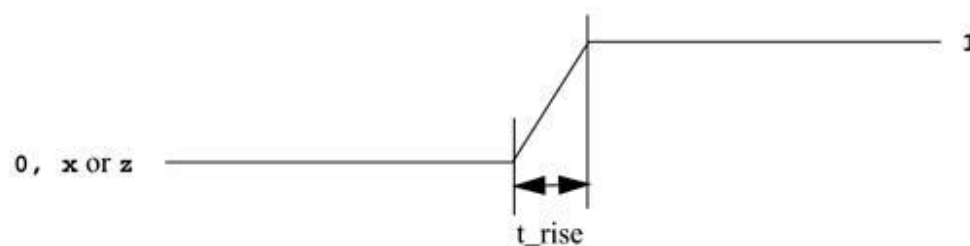
Until now, we described circuits without any delays (i.e., zero delay). In real circuits, logic gates have delays associated with them. Gate delays allow the Verilog user to specify delays through the logic circuits. Pin-to-pin delays can also be specified in Verilog.

Rise, Fall, and Turn-off Delays

There are three types of delays from the inputs to the output of a primitive gate.

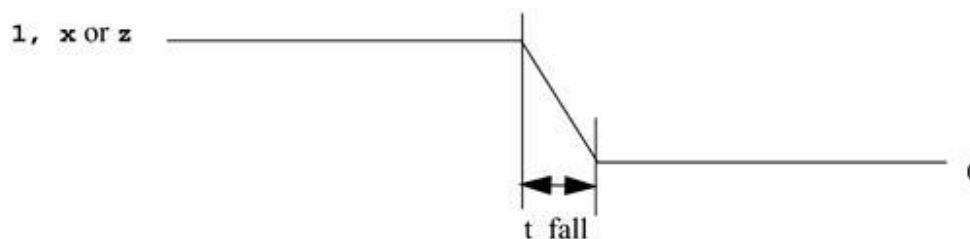
Rise delay

The rise delay is associated with a gate output transition to a 1 from another value.



Fall delay

The fall delay is associated with a gate output transition to a 0 from another value.



Turn-off delay

The turn-off delay is associated with a gate output transition to the high impedance value (z) from another value.

If the value changes to x, the minimum of the three delays is considered.

Three types of delay specifications are allowed. If only one delay is specified, this value is used for all transitions. If two delays are specified, they refer to the rise and fall delay values. The turn-off delay is the minimum of the two delays. If all three delays are specified, they refer to rise, fall, and turn-off delay values. If no delays are specified, the default value is zero.

Example--Types of Delay Specification

```
//Delay of delay_time for all transitions  
and #(delay_time) a1(out, i1, i2);
```

```
// Rise and Fall Delay Specification.
```

```
and #(rise_val, fall_val) a2(out, i1, i2);
```

```
// Rise, Fall, and Turn-off Delay Specification
```

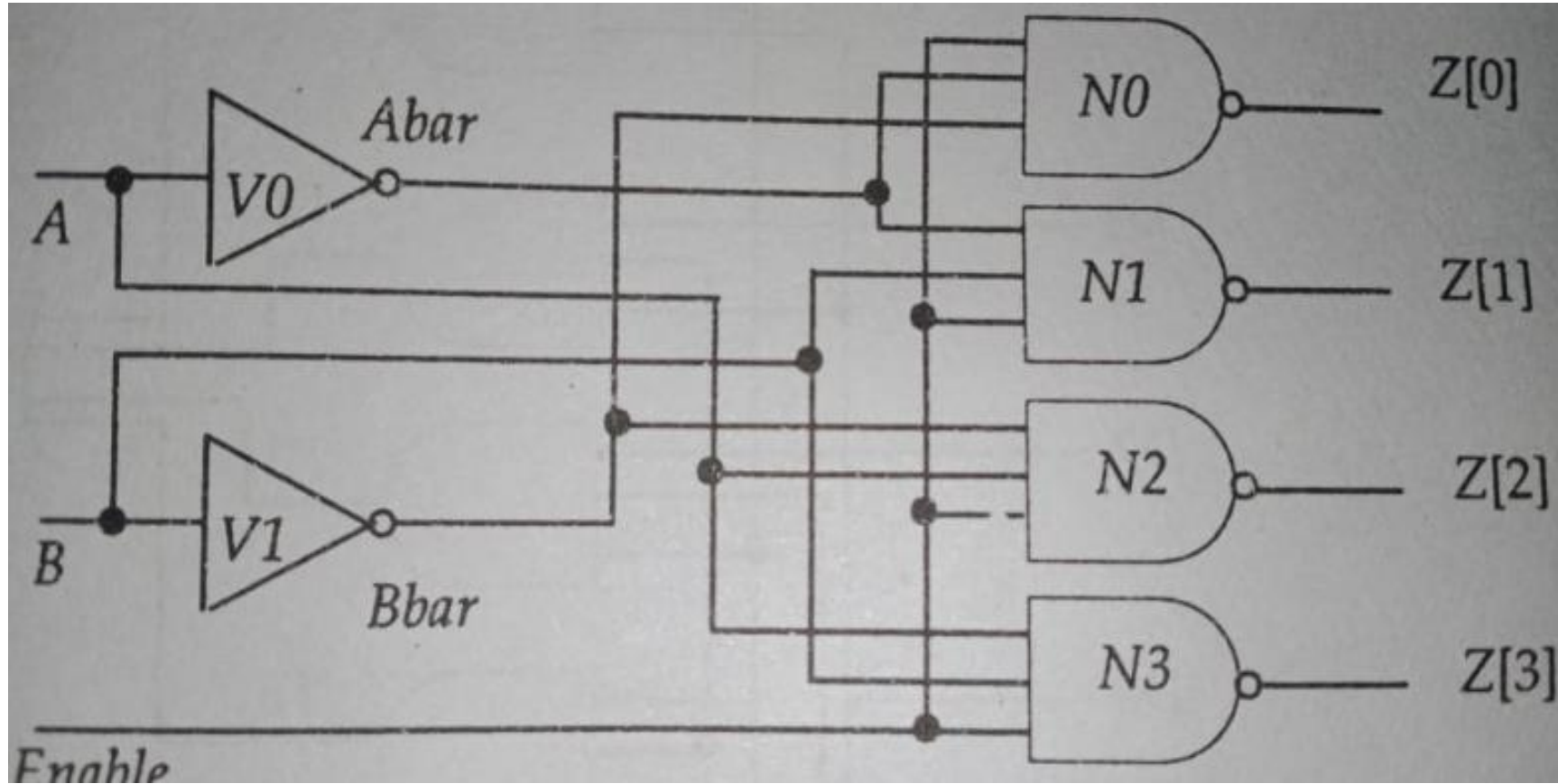
```
bufif0 #(rise_val, fall_val, turnoff_val) b1 (out, in, control);
```

Examples of delay specification are shown below.

```
and #(5) a1(out, i1, i2); //Delay of 5 for all transitions and #(4,6) a2(out, i1, i2); // Rise  
= 4, Fall = 6
```

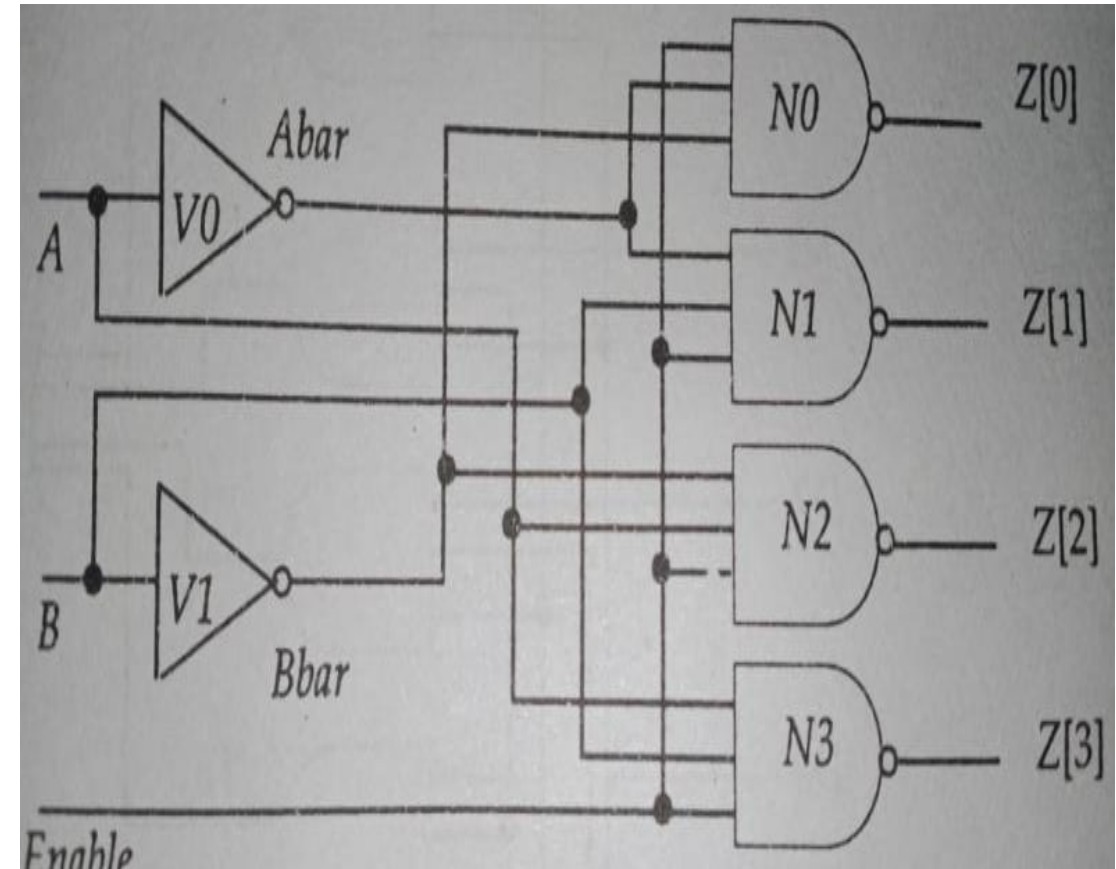
```
bufif0 #(3,4,5) b1 (out, in, control); // Rise = 3, Fall = 4, Turn-off = 5
```

2 to 4 Decoder



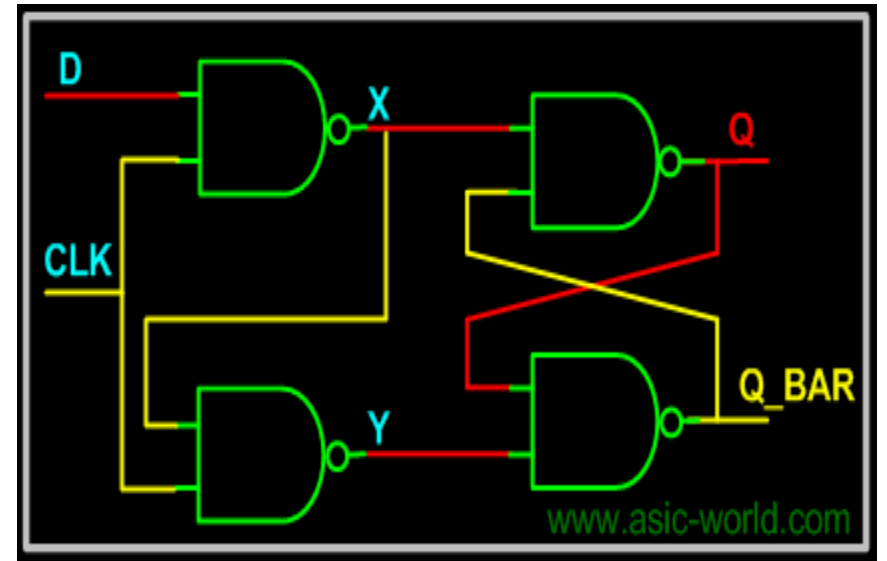
EXAMPLE :2 to 4 Decoder

```
module 2 to 4 dec (Z,A,B,Enable);  
input A,B,Enable;  
output [3:0] Z;  
wire Abar,Bbar;  
  
not V0(Abar,A);  
not V1(Bbar,B);  
  
nand N0 (Z[0],Enable,Abar,Bbar);  
nand N1 (Z[1],Enable,Abar,B);  
nand N2(Z[2],Enable,A,Bbar);  
nand N3 (Z[3],Enable,A,B);  
end module
```

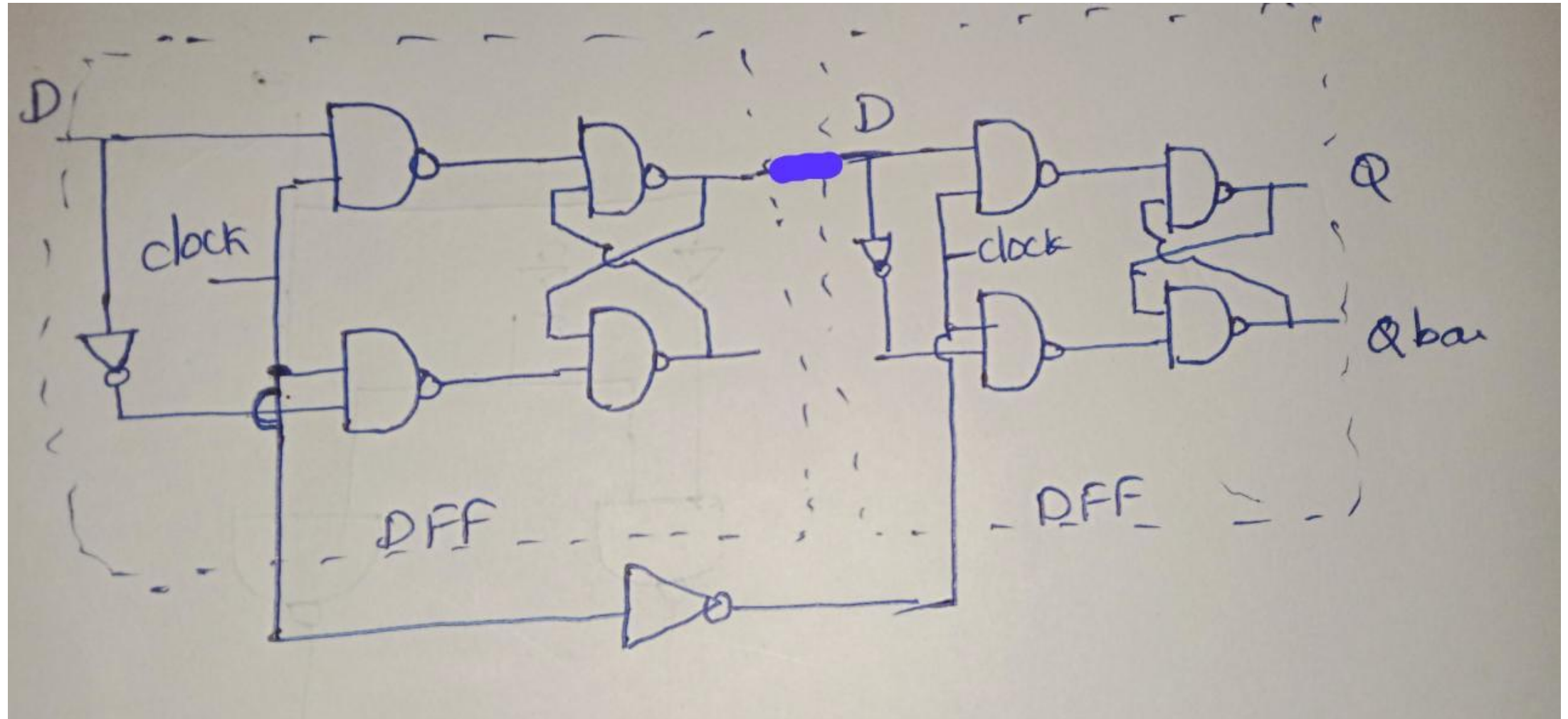


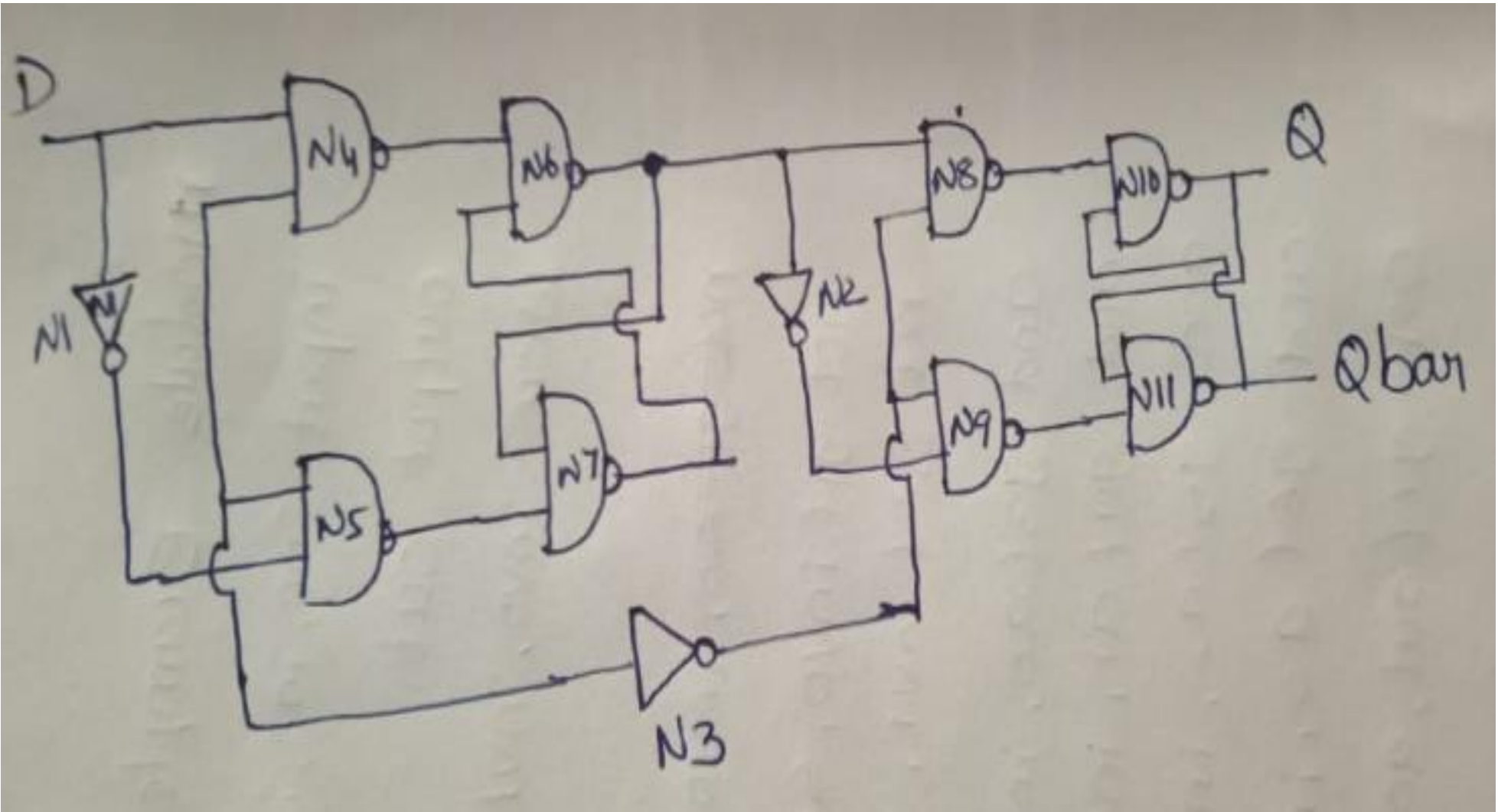
D Flip flop

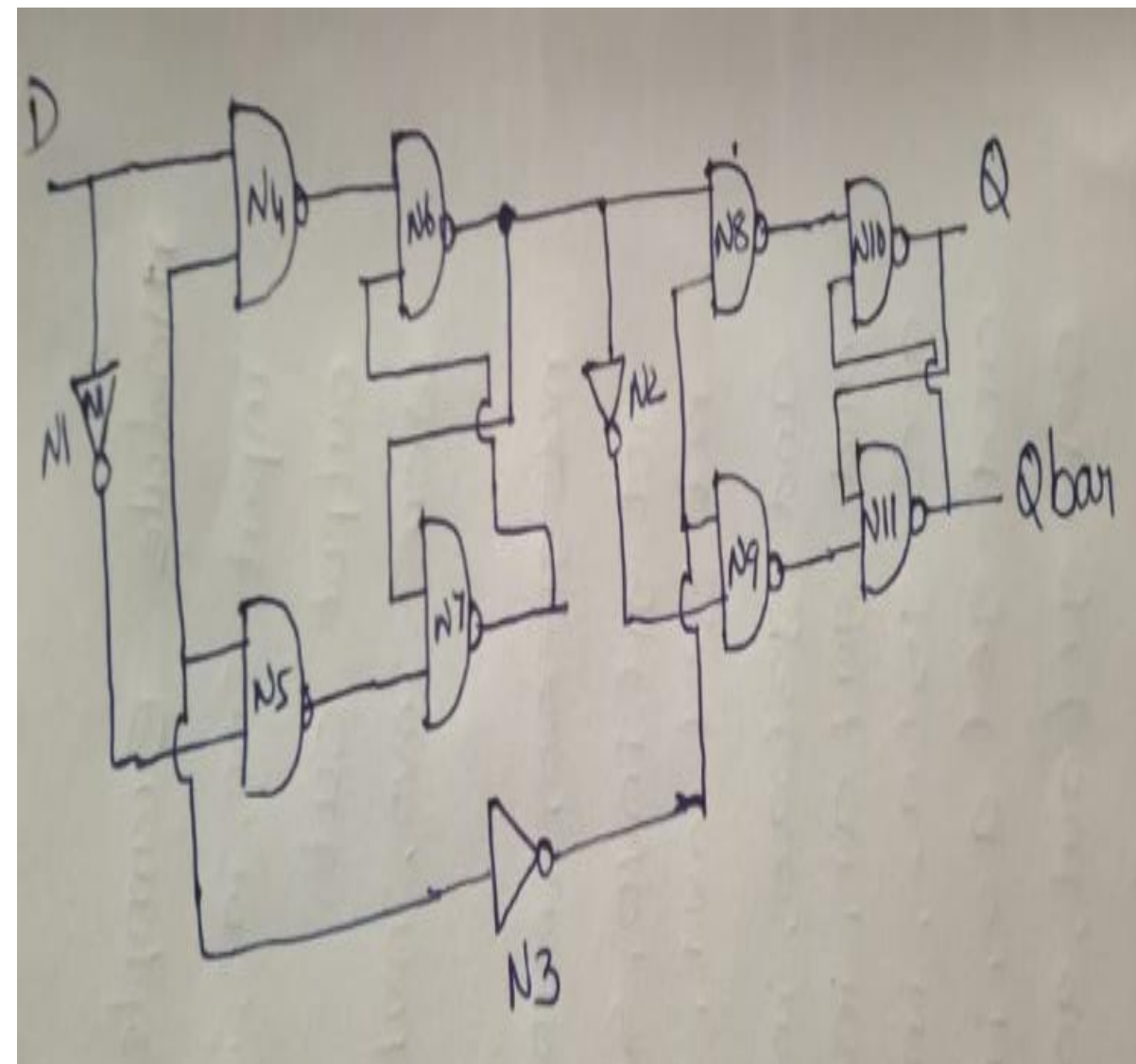
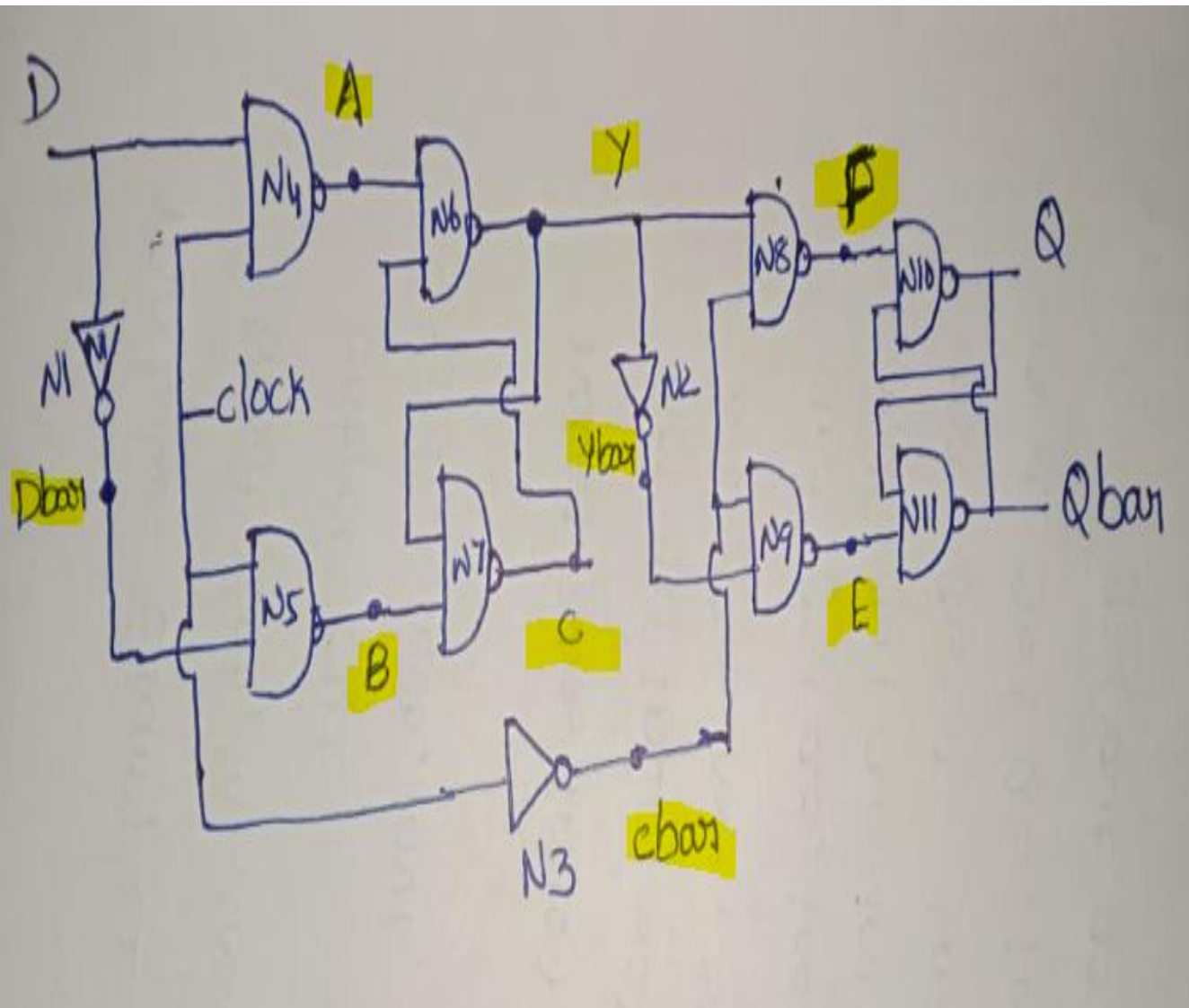
```
module dff_from_nand(Q,Q_BAR,D,CLK);  
input D,CLK;  
output Q,Q_BAR;  
wire X,Y;  
nand U1 (X,D,CLK) ;  
nand U2 (Y,X,CLK) ;  
nand U3 (Q,Q_BAR,X);  
nand U4 (Q_BAR,Q,Y);  
end module
```



MASTER SLAVE FLIP FLOP







```

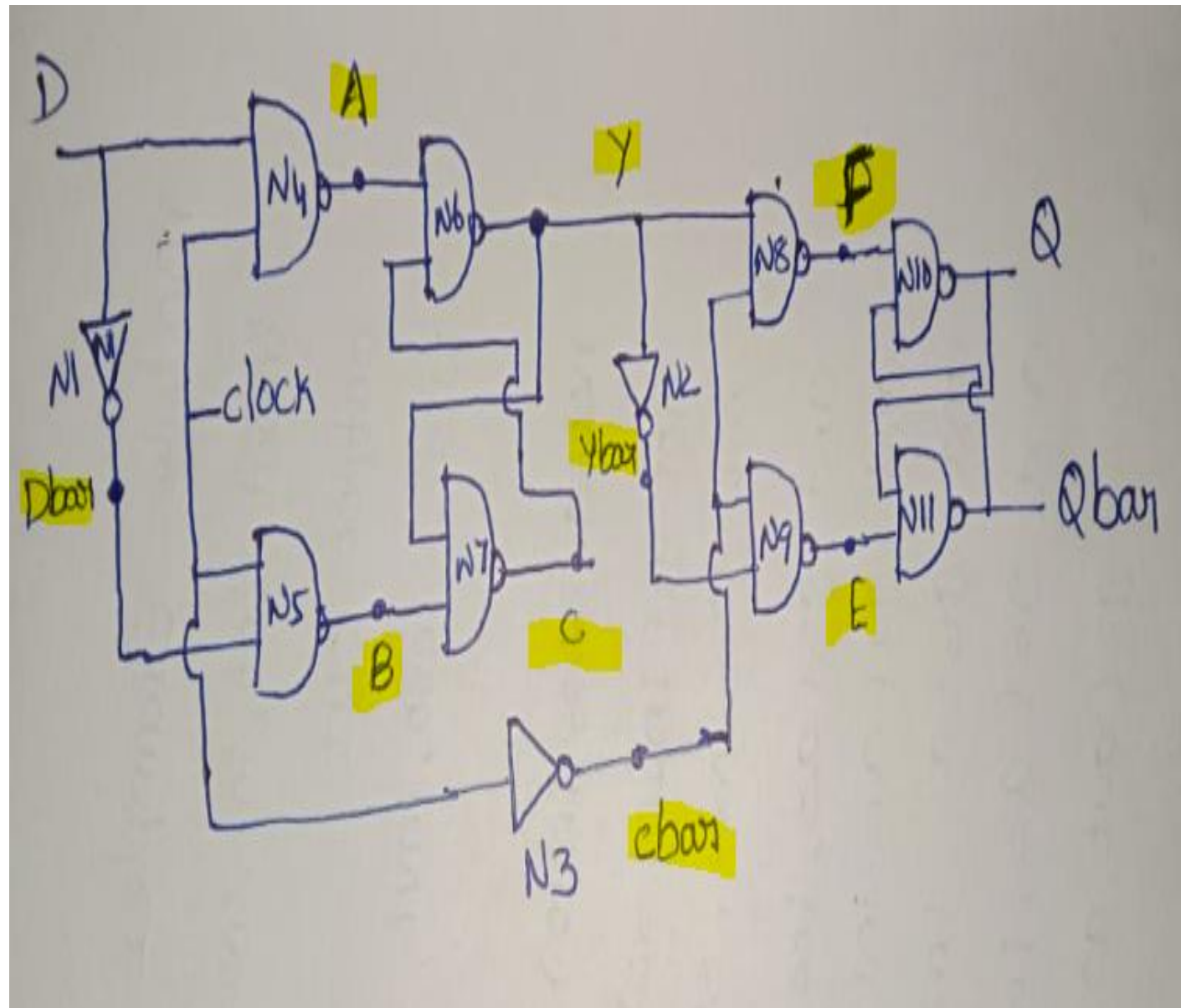
module MSFF ( Q, Qbar, D, clock);
output Q, Qbar;
input D, clock;
wire Dbar, cbar, ybar, A, B, C, E, F;

not N1 (Dbar, D);
not N2 (ybar, Y);
not N3 (cbar, clock);

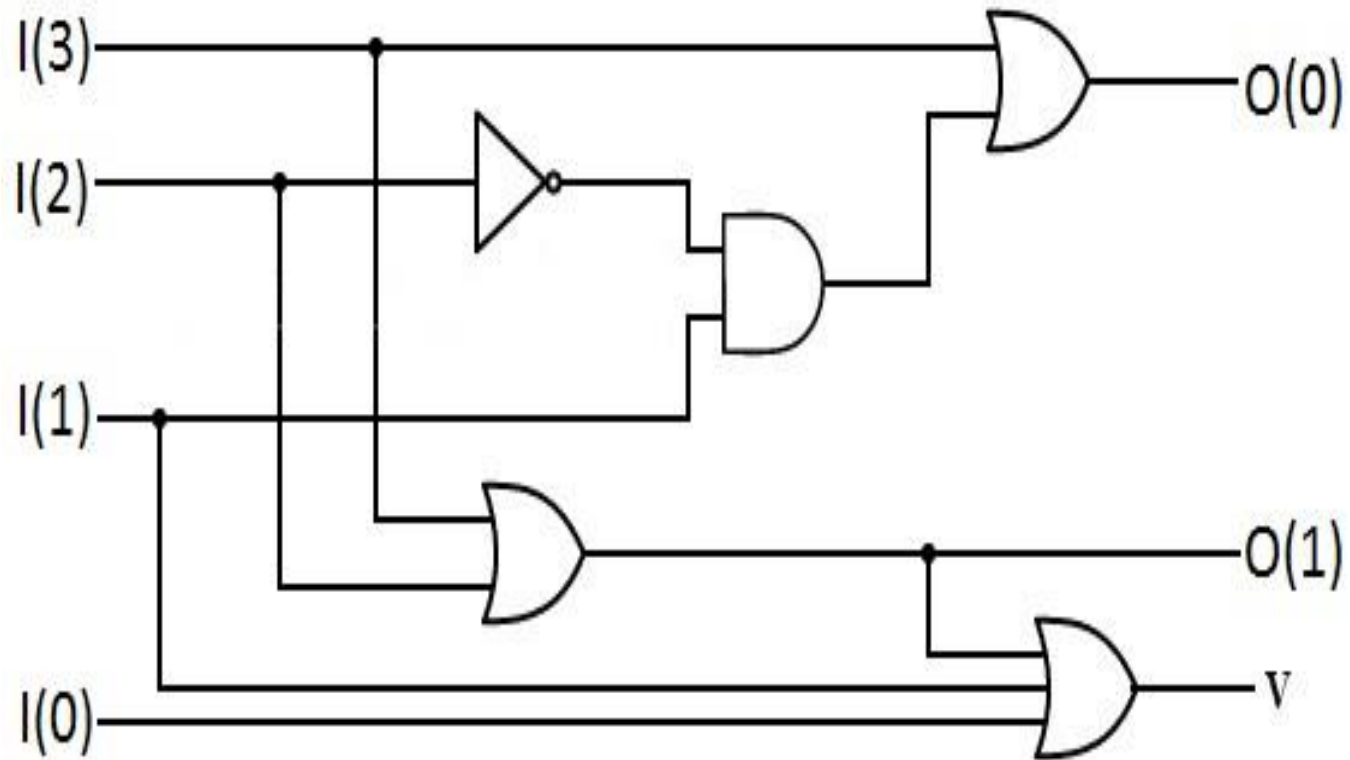
nand N4 (A, D, clock);
nand N5 (B, clock, Dbar);
nand N6 (Y, A, C);
nand N7 (C, B, Y);
nand N8 (F, Y, cbar), N9 (E, cbar, ybar);
nand N10 (Q, F, Qbar);
N11 (Qbar, E, Q);

end module

```

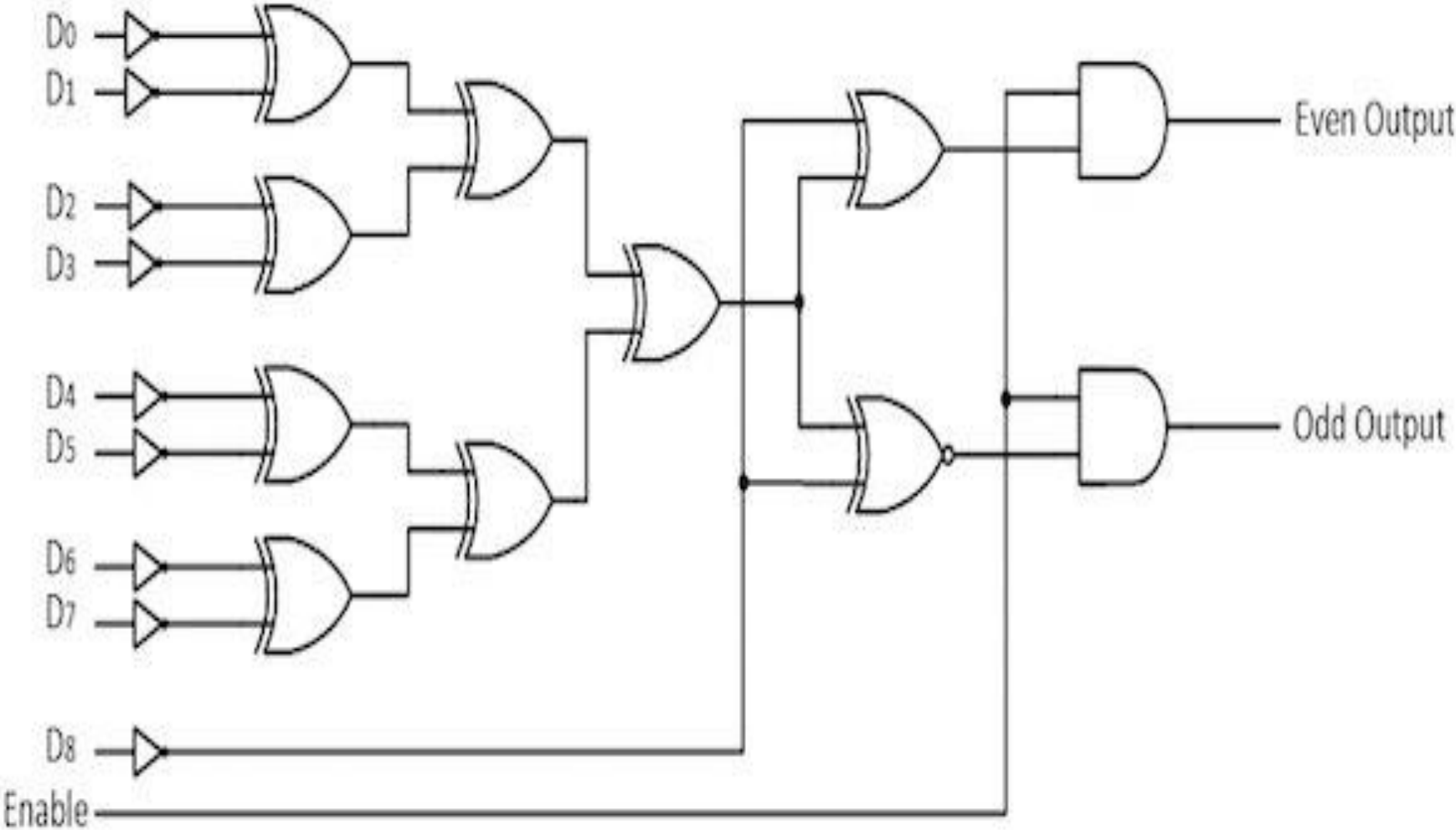


PRIORITY ENCODER



Inputs				Outputs		
D ₀	D ₁	D ₂	D ₃	Y ₁	Y ₀	V
0	0	0	0	×	×	0
1	0	0	0	0	0	1
×	1	0	0	0	1	1
×	×	1	0	1	0	1
×	×	×	1	1	1	1

PARITY GENERATOR



Implicit Nets

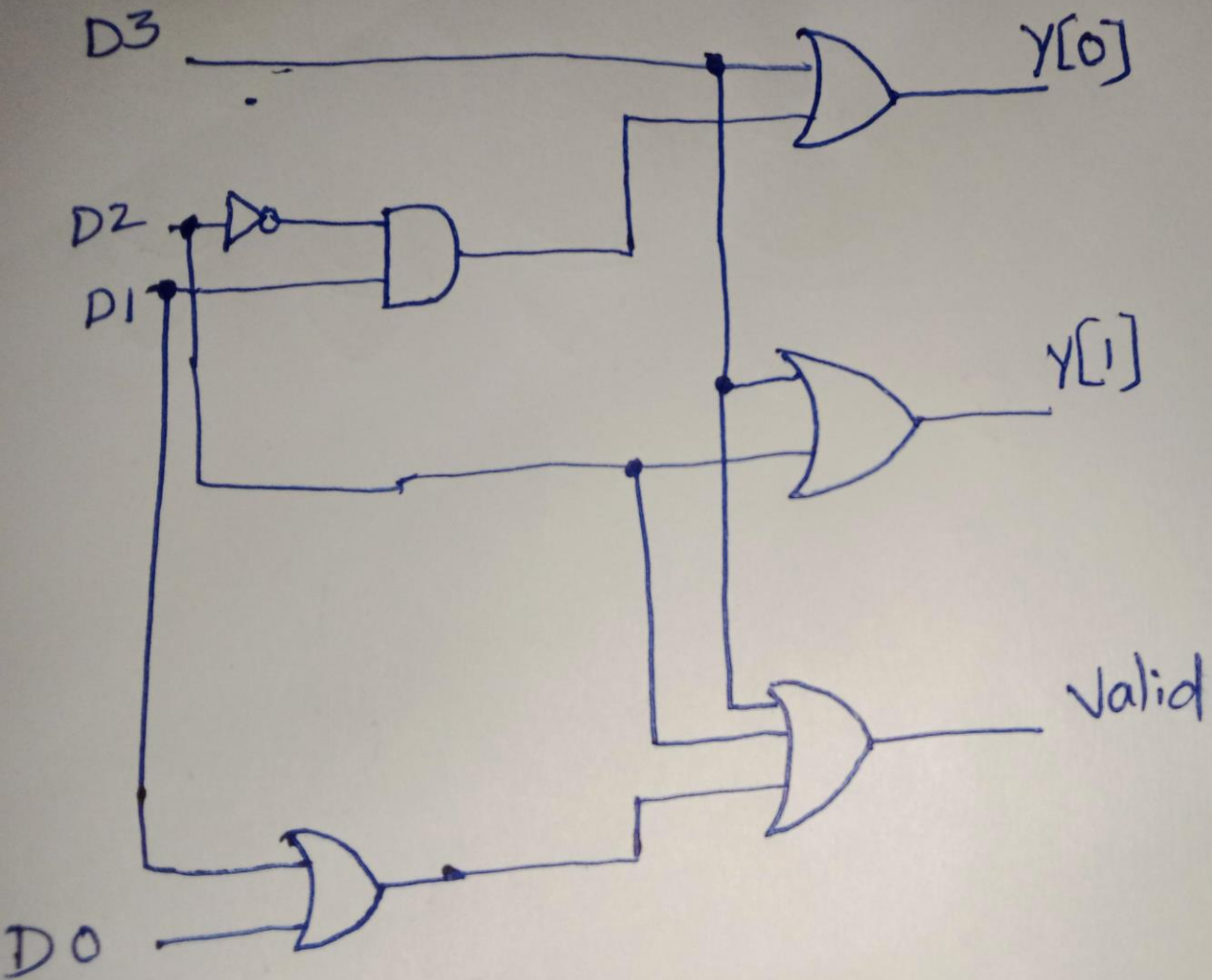
- If a net is not declared in a verilog model, by default it is implicitly declared as 1 bit wire.
- For this purpose we use compiler directive i.e `default_nettype`
- **Syntax**

```
default_nettype net_type;
```

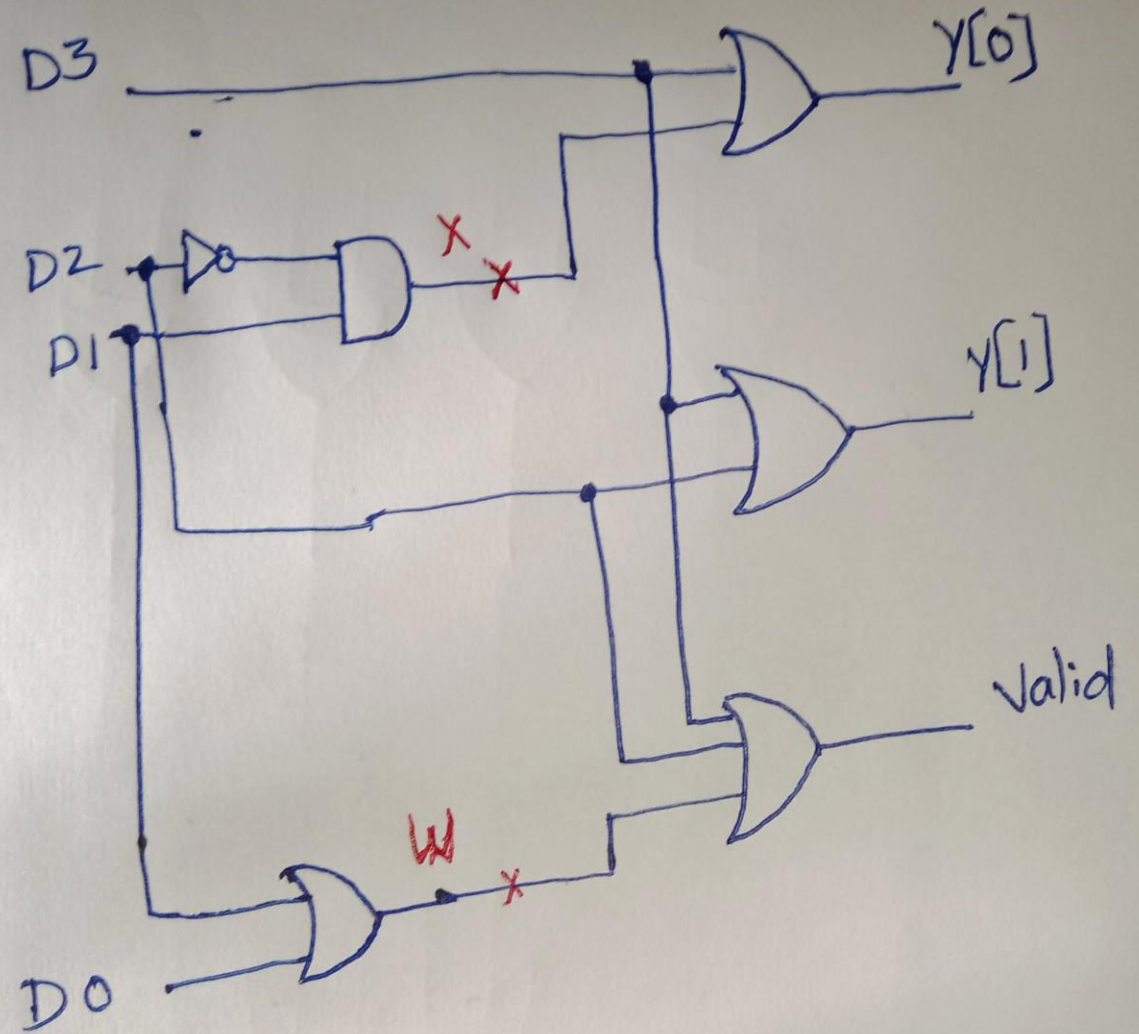
- Example

```
default_nettype wand;
```

- With the above declaration all uncared nets are type wand.



II. Priority encoder



II. Priority encoder

```
module Pri_enc (Y, valid, D);
```

```
    output [1:0] Y;
```

```
    output valid;
```

```
    input [3:0] D;
```

```
    not g1 (D2bar, D[2]);
```

```
    and g2 (x, D2bar, D[1]);
```

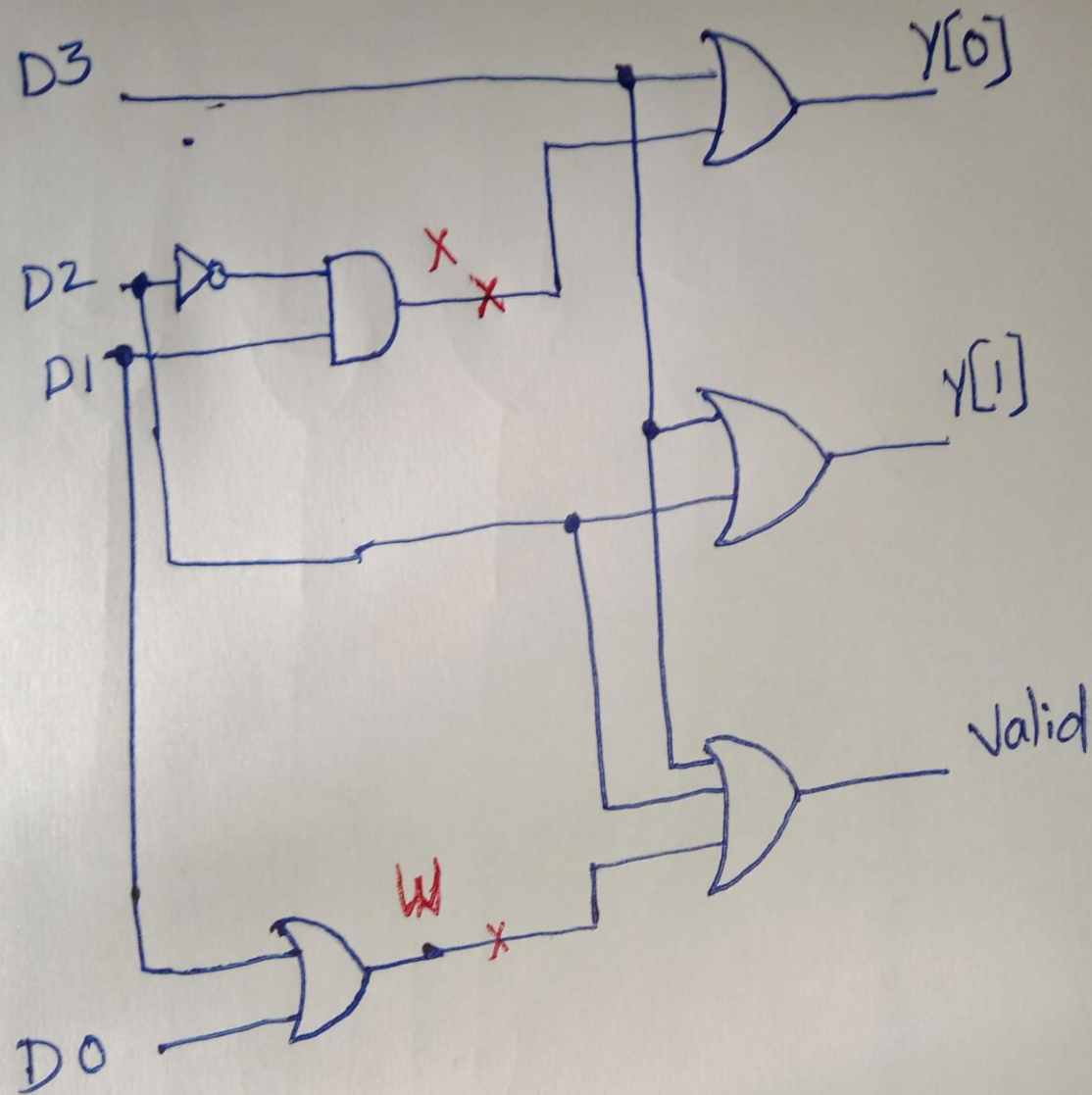
```
    or g3 (w, D[0], D[1]);
```

```
    or g4 (Y[0], D[3], x);
```

```
    or g5 (Y[1], D[3], D[2]);
```

```
    or g6 (valid, D[3], D[2], w);
```

```
end module
```



II. Priority encoder

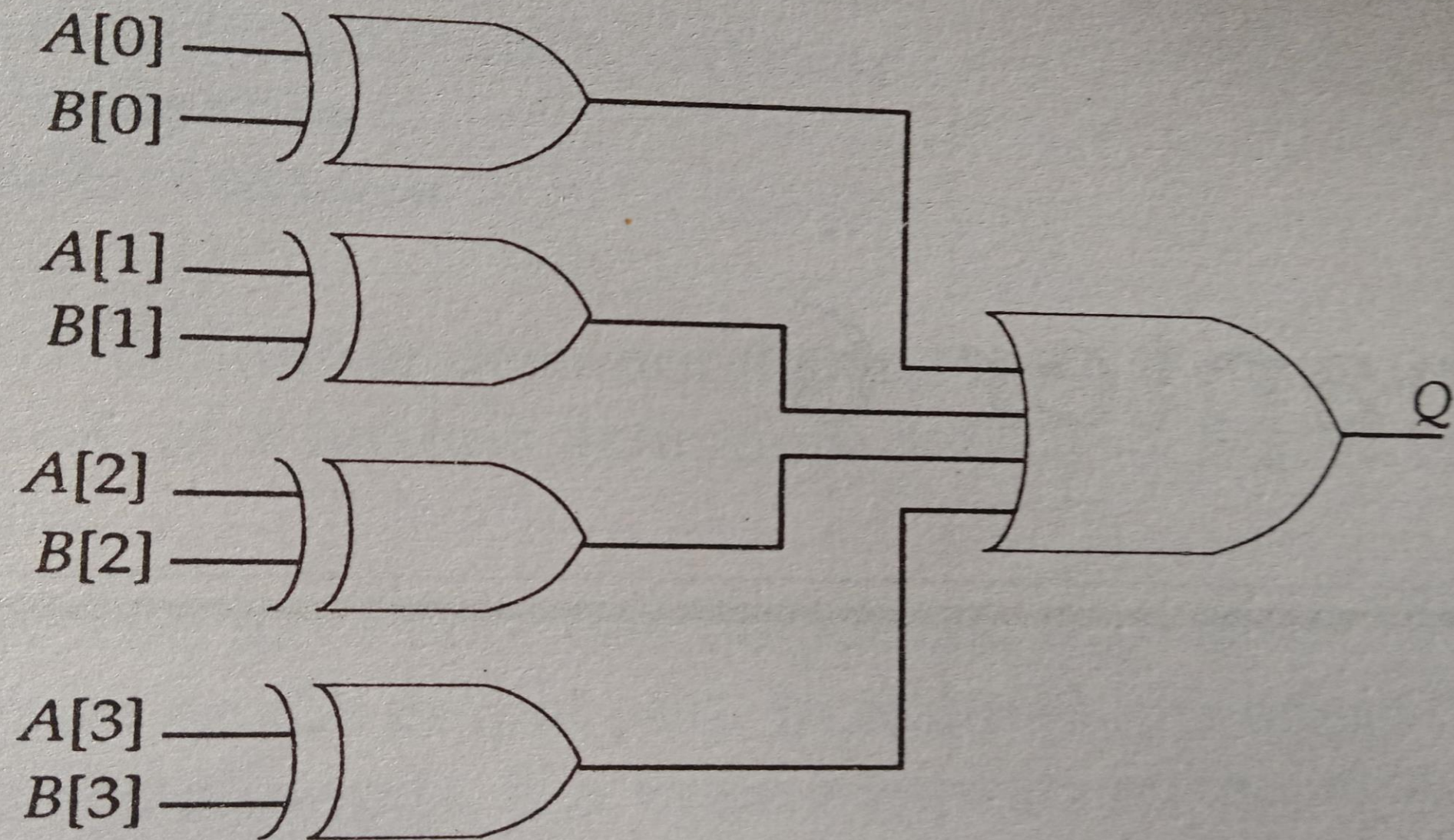


Figure 5-11 Logic for A not equals B .

Verilog code gate level

Module example (Q,A);

output Q;

input [3:0]A,B;

xor g1(w,A[0],B[0]);

xor g2(X,A[1],B[1]);

xor g3(Y,A[2],B[2]);

xor g4(Z,A[3],B[3]);

or g5(Q,W,X,Y,Z);

end module

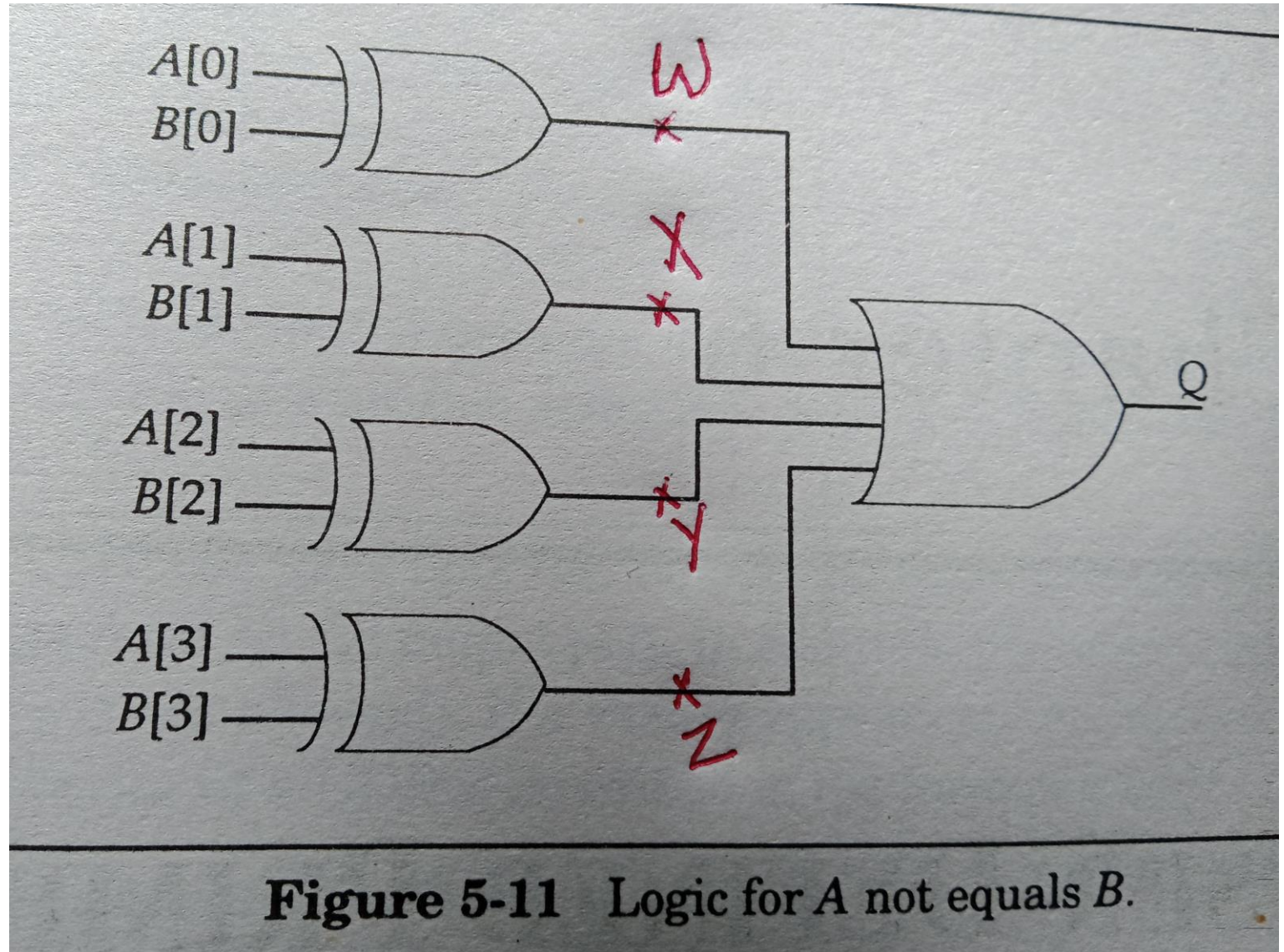
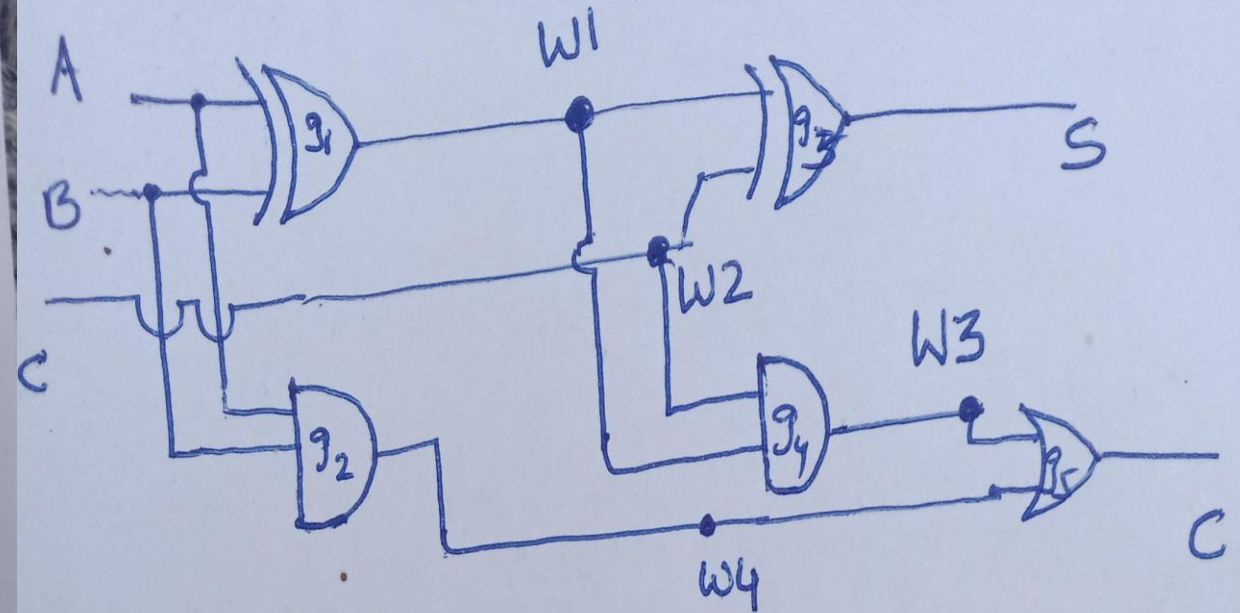
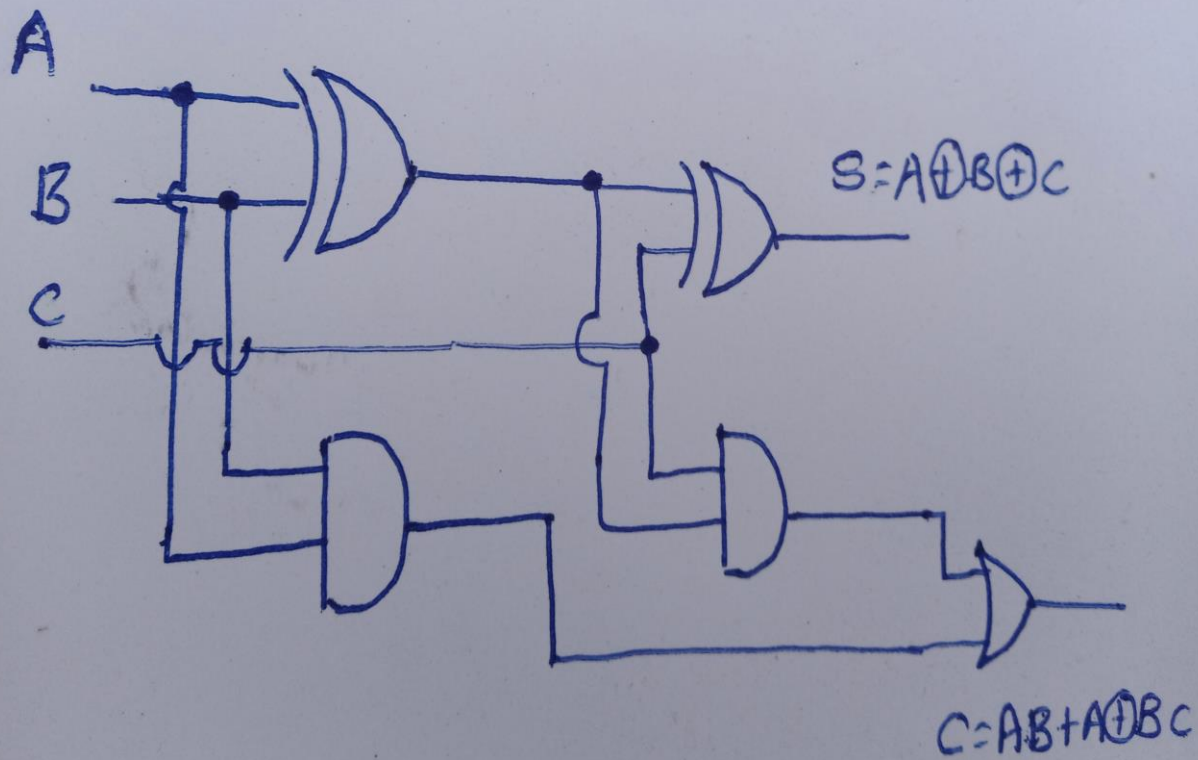


Figure 5-11 Logic for A not equals B.

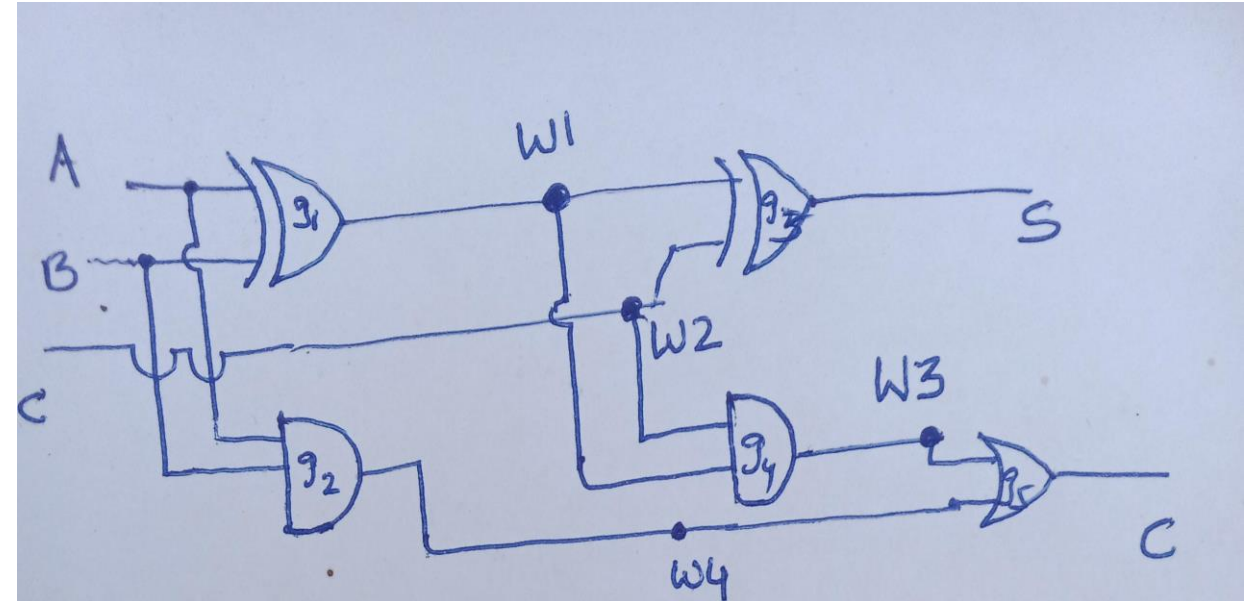
Full adder



⑥ FULL ADDER IN Gate level model

Verilog program

```
module full-adder(S, c, A, B, C);  
  input A, B, C;  
  output S, c;  
  wire w1, w2, w3, w4;  
  xor g1(w1, A, B);  
  and g2(w2, A, B);  
  xor g3(S, w1, w2);  
  and g4(w3, w2, w1);  
  or (c, w3, w4);  
end module;
```



⑥ FULL ADDER IN Gate level model

```

module Example2 (out, in0, in1, in2, in3, s0, s1);
    input in0, in1, in2, in3, s0, s1;
    output out;
    wire inv0, inv1, a0, a1, a2, a3;

```

```

not g1(s0, inv0);

```

```

not g1(inv0, s0);

```

```

not g2(inv1, s1);

```

```

and g3(a0, in0, inv0, inv1);

```

```

and g4(a1, in1, inv0, s1);

```

```

and g5(a2, in2, s0, inv1);

```

```

and g6(a3, in3, s0, s1);

```

```

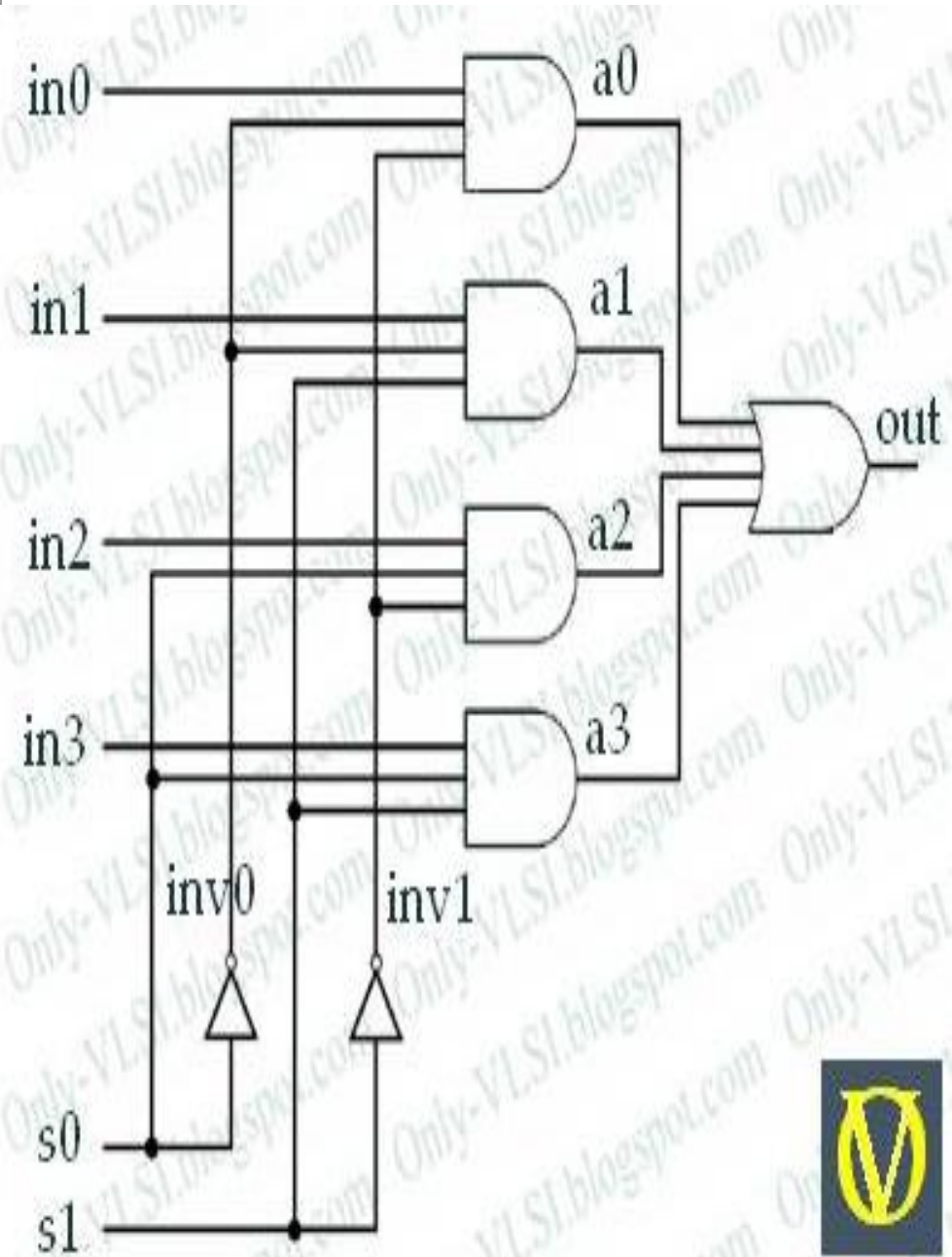
or g7(out, a0, a1, a2, a3);

```

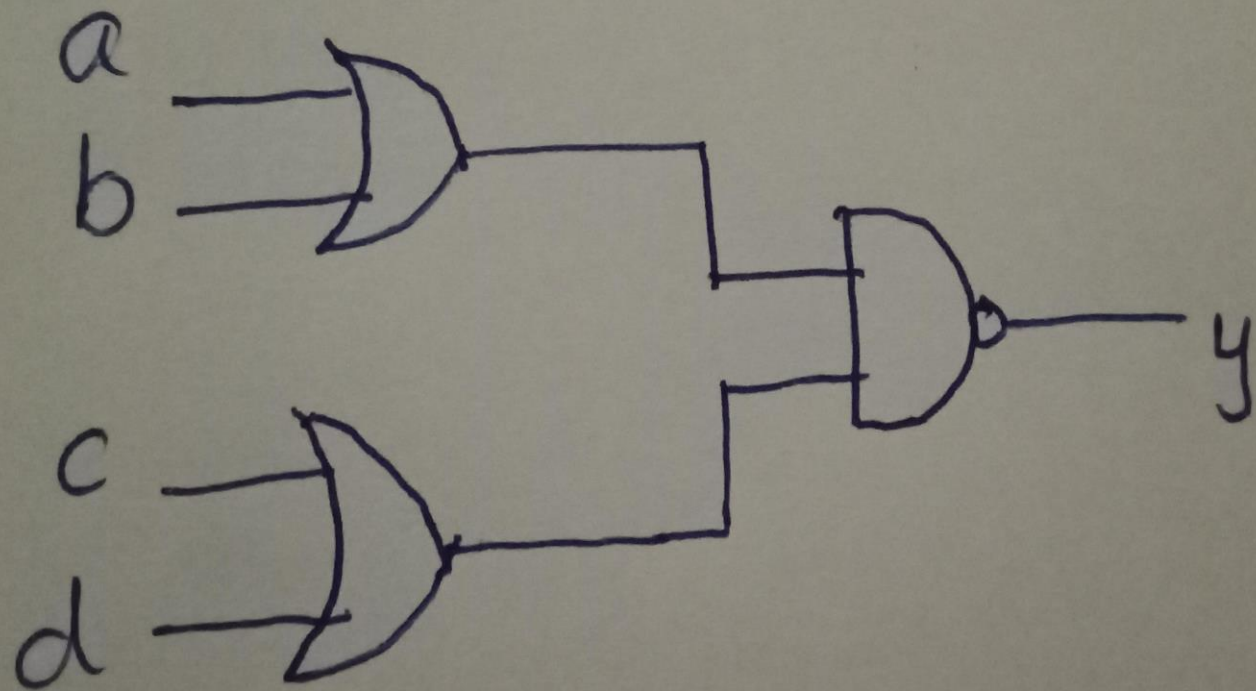
```

end module

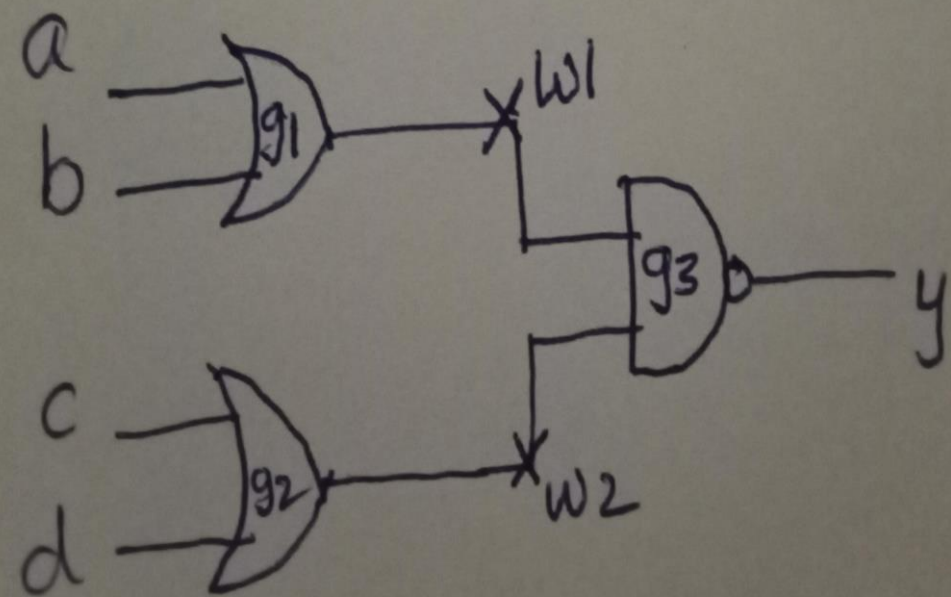
```



OR - AND - Invert



OR - AND - Invert



```
module OAIgate (y,a,b,c,d)
```

```
output y;
```

```
input a,b,c,d;
```

```
Wire w1,w2;
```

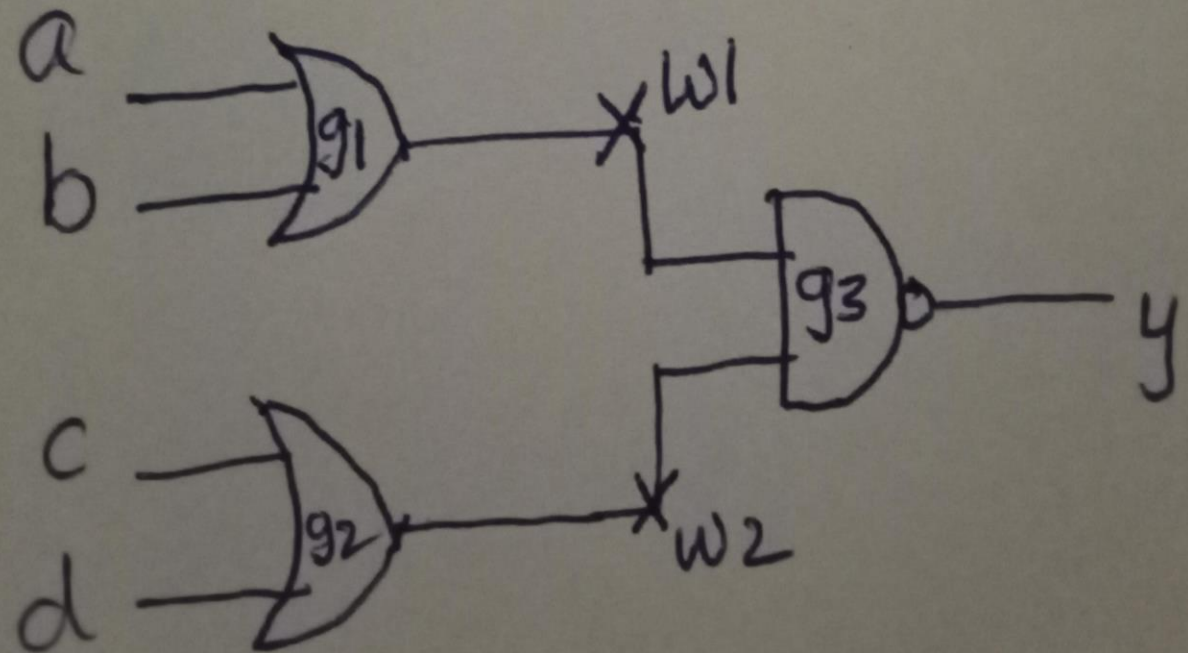
```
or g1 (w1,a,b);
```

```
or g2 (w2,c,d);
```

```
nand g3 (y,w1,w2);
```

```
end module;
```

OR-AND-Invert



Example aoi gate

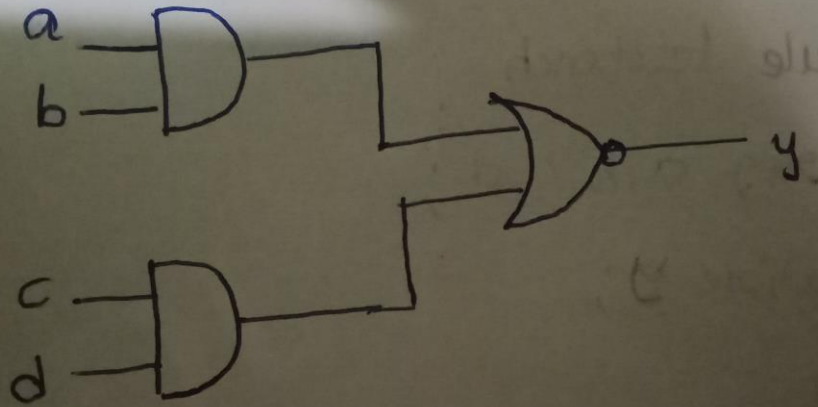
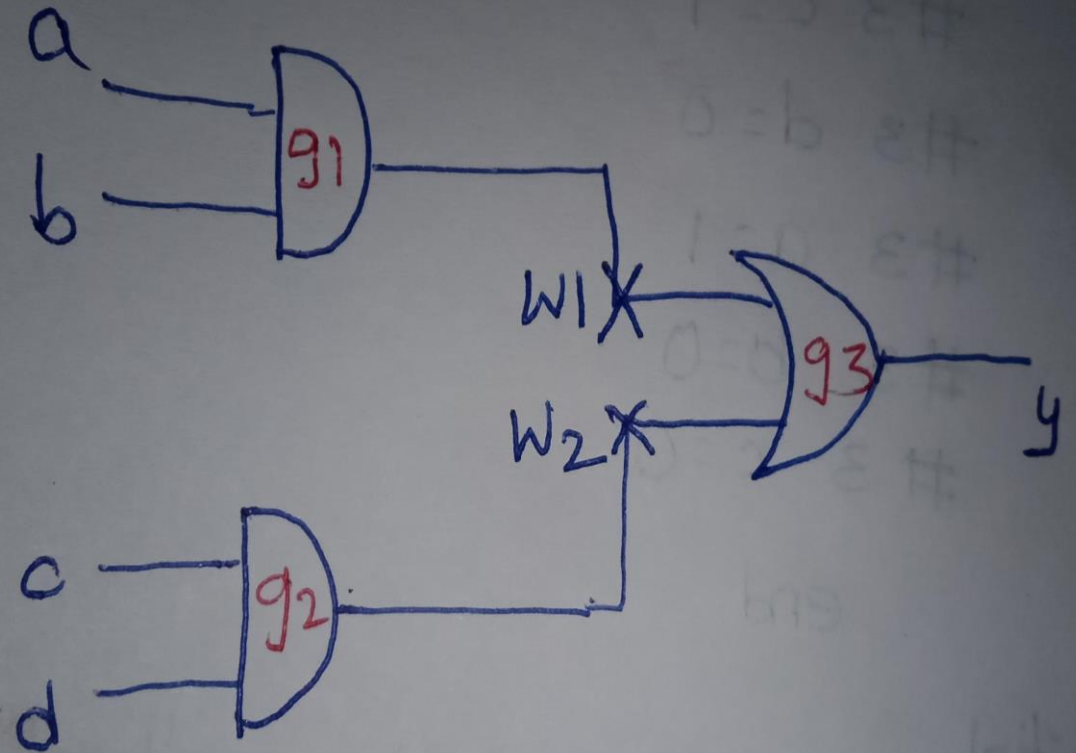


fig: a typical AOI gate



Verilog code

// Verilog code for A-O-I logic circuit

```
module aoigate (y,a,b,c,d);
```

```
input a,b,c,d;
```

```
output y;
```

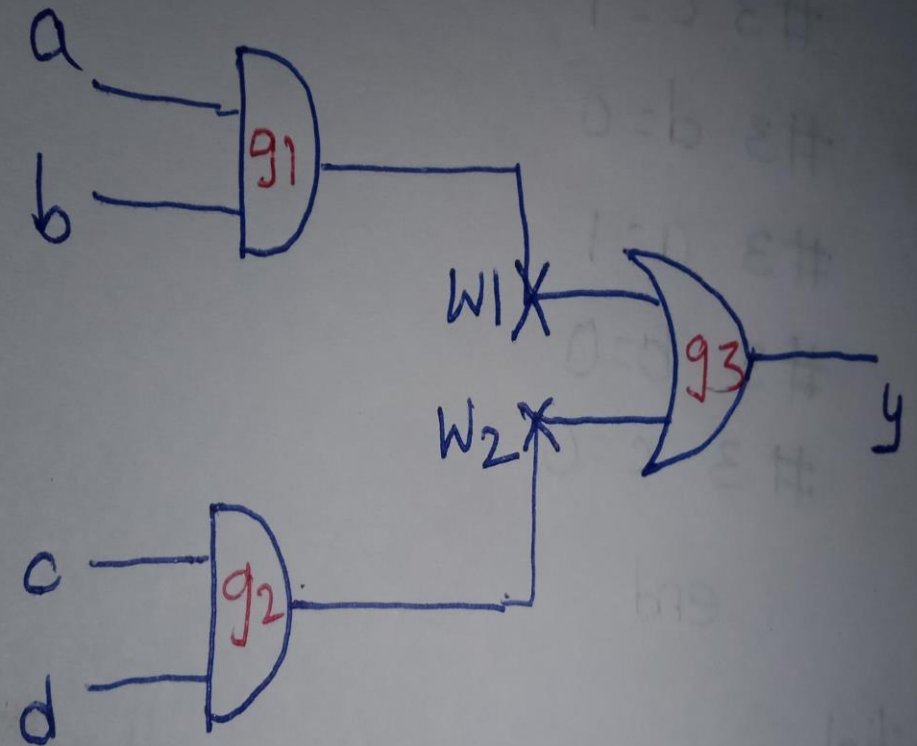
```
wire w1,w2;
```

```
and g1(w1,a,b);
```

```
and g2(w2,c,d);
```

```
nor g3(y,w1,w2);
```

```
end module;
```



Test bench for aoi gate

```
// Test bench code for aoi gate

module testbench
    reg a,b,c,d;
    wire y;

    initial
        begin
            a=0; b=0; c=0; d=0;
            #3 a=1
            #3 b=1
            #3 c=1
            #3 d=0
            #3 a=1
            #3 b=0
            #3 c=0
        end

    initial
        #100 $monitor($time, "y=%b, a=%b, b=%b, c=%b, d=%b", y,a,b,c,d);

    aoi gate gg(y,a,b,c,d);
end module
```

	a	b	c	d
	0	0	0	0
#3	1	0	0	0
#3	1	1	0	0
	1	1	1	0
	1	1	1	1
	1	1	1	0
	1	0	1	1
	1	0	0	

BEHAVIORAL MODELLING.

INTRODUCTION.

Behavioural level modelling constitutes design description at an abstract level. One can visualize the circuit in terms of its key modular functions and their behaviour. It can be described at a functional level itself instead of getting bogged down with implementation details.

Operations and Assignments

The design description at the behavioural level is done through a sequence of assignments. These are called 'procedural assignments'.

The procedure assignment is characterized by the following:

→ The assignment is done through the "=" symbol as was the case with the continuous assignment earlier.

→ An operation is carried out and the result assigned through the "=" operator to an operand specified on the left side of the "=" sign. For example, $N = \sim N$. Here the content of reg N is complemented and assigned to the reg N itself. The assignment is essentially an updating activity.

→ The operation on the right can involve operands and operators. The operands can be of different types - logical variables, numbers - real or integer and so on.

→ All the operands are given in tables. The format of using them and the rules of precedence remain the same.

→ The operands on the right side can be of the net or variable type. They can be scalars or vectors.

→ It is necessary to maintain consistency of the net to the operands in the operation expression -

eg, $N = m / l$; here m and l have to be same

types of quantities - Specifically a reg, integer, time, real, ② realtime, or memory type of data - declared in advance.

→ The operand to the left of the "=" operator has to be of the variable. It is specifically declared accordingly. It can be a scalar, a vector, a part vector, or a concatenated vector.

→ procedural assignments are very much like sequential statements in C. Normally they are carried out one at a time sequentially. As soon as a specified operation on the right is carried out, the result is assigned to the quantity on the left.

- for example $N = m + 1$; $N_1 = N * N$; The above form a set of two procedures placed within an always block. Generally they are carried out sequentially in the order specified.

The sequential nature of the assignments requires the operands on the left of the assignment to be of reg type.

FUNCTIONAL BIFURCATION

Design description at the behavioural level is done in terms of procedures of two types; one involves functional description and interlinks of functional units. It is carried out through a series of blocks under an "always".

The second concerns simulation - its starting point, steering the simulation flow, observing the process; all these can be carried out under the "always" banner, an "initial" banner, or their combinations. However, each always and each initial block initiates an activity simulation process.

In general the activity with all such blocks starts at the simulation time and flows concurrently during the whole simulation process.

A procedure - block of either type - initial or always.

Type of block is specified here: only two types are possible:- initial & always

The symbol signifies an event Control (only for always blocks).

Specifies the event which flags off the execution of the block (only for always blocks)

type of Block @ (sensitivity_list)

begin; name of block

local variable declarations

procedure assignment statements;

end

All the activities within the blocks are enclosed within the begin-end construct

The procedural statements from the body of the block.

All variable etc., local to the block are declared at the beginning of the block. The block can be assigned a name which can be referred.

INITIAL CONSTRUCT

(1)

A set of procedural assignment within an initial Construct are executed only once - and, that too, at the times specified for the respective assignments. The initial process is to be characterized by the following.

→ In any assignment statement the left-hand side has to be a storage type of element. It can be a reg, integer, or real type of variable. The right-hand side can be a storage type of variable (reg, integer, or real type of variable) or a net.

→ All the procedural assignments appear within a begin-end block

→ All the procedural assignments are executed sequentially - in the same order as they appear in the design description. The initial block above does three Controlling activities the simulation run.

→ Initialize the selected set of reg's at the start.

→ Change values of reg's at predetermined instances of time. These form the inputs to the module under test and test it for a desired test sequence

→ Stop simulation at the specified time.

ALWAYS CONSTRUCT

The always process signifies activities to be executed on an "always basis". Its essential characteristics are :

→ Any behavioural level design description is done using an always block.

→ The process has to be flagged off by an event or a change in a net or a reg.

→ The process can have one assignment statement or multiple assignment statements. In the latter case all the

the assignments are grouped together within a "begin - (5) end" Construct.

→ Normally the statements are executed sequentially in the order they appear.

Event Control

The always block is executed repeatedly and endlessly. It is necessary to specify a condition or a set of conditions, which will steer the system to the execution of the block. Alternatively such as flagging-off can be done by specifying an event preceded by the symbol "@".

@(negedge) clk : executes the following block at the negative edge of the reg (Variable) clk.

@(posedge clk): executes the following block at the positive edge of the reg (Variable) clk.

@ clk : executes the following block at the both edges of clk.

The events can be changes in reg, integer, real or signal on a net. These should be declared beforehand.

No algebra or logic operation is permitted as an event. The OR'ing signifies "execute the block if any one of the events takes place".

The positive transition for a reg type single bit variable is a change from 0 to 1.

for a logic variable it is a transition from false to true.

The "posedge" transition for a signal on a net can of three types:

0 to 1

X or Z to 1

0 to X or Z

The "negedge" transition for a signal on a net ⁽⁶⁾
can be of three different types

1 to 0
1 to x or z
x or z to 0.

If the event specified is in terms of a multibit reg, only its least significant bit is considered for the transition. changes in the other bits are ignored. The event-based flagging-off of a block is applicable only to be always block.

According to the recent version of the LRM the comma operator (,) plays the same role as the keyword only to the always block, or the two can be used interchangeably or in a mixed form. Thus, the following are identical: @ (a or b or c).

@ (a or b, c)

@ (a, b, c)

@ (a, b or c).

EXAMPLES

Versatile Counter module.

```
Counterup (a, clk, N);
```

```
input clk;
```

```
input [3:0] N;
```

```
output [3:0] a;
```

```
reg [3:0] a;
```

```
initial a = 4'b0000;
```

```
always @ (negedge clk) a = (a == N) ? 4'b0000
```

```
    a + 1'b1;
```

```
endmodule
```


TEST - BENCH

7

```
module tst_Counterup;
```

```
    reg clk;
```

```
    reg [3:0] N;
```

```
    wire [3:0] a;
```

```
    Counterup ci (a, clk, N);
```

```
    initial
```

```
    begin
```

```
        clk = 0;
```

```
        N = 4'b1011;
```

```
    end
```

```
    always #2 clk = ~clk;
```

```
    initial $monitor($time, " a = %b, clk = %b, N = %b", clk, N);
```

```
endmodule
```

```
module Counterdn (a, clk, N);
```

```
    input clk;
```

```
    input [3:0] N;
```

```
    output [3:0] a;
```

```
    reg [3:0] a;
```

```
    initial a = 4'b0000;
```

```
    always @ (negedge clk) a = (a == 4'b0000) ? N : a - 1'b1;
```

```
endmodule
```

```
module updCounter (a, clk, N, u_d);
```

```
    input clk, u_d
```

```
    input [3:0] N;
```

```
    output [3:0] a;
```

```
    reg [3:0] a;
```

```
    initial a = 4'b0000;
```

```
    always @ (negedge clk) a = (u_d) ? ( (a == N) ?
```

```
4'b0000 : a + 1'b1) : ( (a == 4'b0000 ? N : a - 1'b1);
```

endmodule

8

```
module ctrupdcou (a, clr, clk, N, u_d);
```

```
    input clr, clk, u_d;
```

```
    Output [3:0] a;
```

```
    input [3:0] N;
```

```
    reg [3:0] a;
```

```
    initial a = 4'b0000;
```

```
    always @ (negedge clk or posedge clr) a = (clr)? 4'h0 :  
    ((u_d)? ((a==N)? 4'b0000 : a+1'b1) : ((a==4'b0000)? N : a-1'b1));
```

```
endmodule.
```

Example shift register

```
module shiftr (a, clk, r-1);
```

```
    input clk, r-1;
```

```
    Output [7:0] a;
```

```
    reg [7:0] a;
```

```
    initial a = 8'h01;
```

```
    always @ (negedge clk) begin a = (r-1)?
```

```
        (a >> 1'b1) : (a << 1'b1);
```

```
    end
```

```
endmodule
```

clocked flip-flop

```
module dff (do, di, clk); Output
```

```
do; input di, clk; reg do;
```

```
    initial
```

```
        do = 1'b0;
```

```
    always @ (negedge clk) do = di;
```

```
endmodule.
```

```

module
diffen (do, di, en);
Output do; Input di, en;
reg do;
    initial
        do = 1'b0;
    always @(di or en)
        if (en) do = di;
endmodule.

```

clock waveform

Consider the design description line always
`#3 clk = ~clk;`

The sequence of operation taking place within this line segment as follows:

when the system comes across the statement, it schedules an activity 3 ns later.

At the end of the 3ns, the value of clk is sensed the sensed value is complemented and then stored temporarily.

Then, the stored value is assigned to the clock which completes the activity of the always blocks; Once again, execution resumes at step 1.

ASSIGNMENT WITH DELAYS

The delays refers to the specific activity it qualifies. A variety of possibilities of specifying delays to assignments exist. Consider the assignment always `#3b=a;`

Simulator encounters this at zero time and posts the the entire activity to be done 3ns later the assignment is scheduled to be repeated every 3ns, irrespective of whether a

Changes in the Meantime.

(10)

Intra-assignment delays

In contrast, the "intra-assignment" delay carries out the assignment in two parts.

$A = \#dl \text{ expression};$

Here the expression is scheduled to be reevaluated as soon as it is encountered. However, the result of the evaluation is assigned to the right-hand side quantity a after a delay specified by dl . dl can change to be an integer or a constant expression.

Zero delay

A delay of 0 ns does not really cause any delay. However, it ensures that the assignment following is executed last in the concerned time slot. Often it is used to avoid indecision in the precedence of execution of assignments.

wait Construct

The wait Construct makes the simulator wait for the specified expression to be true before proceeding with the following assignment or group of assignments. Its syntax has the form $\text{wait} (\alpha) \text{ assignment}_1$; α can be variable, the value on a net, or an expression involving them. If α is an expression, it is evaluated; if true, assignment $_1$ is carried out. One can also have a group of assignments within a block in place of assignment $_1$. The activity is level-sensitive in nature in contrast to the edge-sensitive nature of event specified through @.

Specifically, the procedural assignment.

@clk $a = b$; assigns the value of b to a when clk changes; if the value of b changes when clk is steady, the value of a remains unaltered.

$\text{wait} (clk) \# 2 a = b$; the simulator waits for the clock to be high and then changes or assigns b to a with a delay

of 2 ns. The assignment will be refreshed as long as (11)
-the CLK remains high.

DESIGNS OF BEHAVIORAL LEVEL

```
module aobeh(o,a,b); output o; input [1:0] a,b;  
    reg o,a1,b1,o1; always @(a[1] or a[0] or  
        b[1] or b[0]) begin a1 = &a; b1 = &b;  
        o1 = a1 || b1; o = ~o1;  
    end  
endmodule
```

```
module aobeh(o,a,b); output o;  
    input [1:0] a,b; reg o;  
    always @(a[1] or a[0] or b[1] or b[0])  
        o = ~( (&a) || (&b));
```

BLOCKING AND NON-BLOCKING ASSIGNMENTS

These are executed sequentially - that is, one statement is executed, and only then the following one is executed. Such assignments block the execution of the following lot of assignments at any time step. Hence, they are called "blocking assignments".

A facility called the "nonblocking assignment" is available for such situations. The symbol " \leq " signifies a non-blocking assignment. The same symbol signifies the "less than or equal to" operator in the context of an operation. The context decides the role of the symbol. The main characteristic of a nonblocking assignment is that its execution is concurrent with that of the following assignment or activity.

Non Blocking Assignment and Delays

These are executed delays - of the assignment type and the intra-assignment type - can be associated with nonblocking assignments also. The principle of their operation is similar to that with blocking assignment.

THE CASE STATEMENT

(12)

The case statement is an elegant and simple construct for multiple branching in a module. The keywords case, endcase, and default are associated with the case construct.

format of the case construct is

Case (expression)

Ref1 : statement1;

Ref2 : statement2;

Ref3 : statement3;

....

default statementd;

endcase

If the evaluated value matches ref1, statement1 is executed and the simulator exits the block; Else expression is compared with ref2 and in case of a match, statement2 is executed and so on. If none of the ref1, ref2, etc... matches the value of the expression, the default statement is executed. A statement or a group of statements is executed if and only if there is an exact bit by bit - match between the evaluated expression and the specified ref1, ref2, etc....

for any of the matches, one can have a block of statements defined for execution. The block should appear within the begin - end construct.

There can be only one default statement or default block. It can appear anywhere in the case statement.

One can have multiple signal combinations values specified for the same statement for execution. Commas separate all of them.

```
module dec2_4 beh(0,i);
```

```
    Output[3:0] o;
```

```
    input [1:0] i;
```

```
    reg [3:0] o;
```

```
    always @ (i) begin
```

```
        Case(i)
```

```
            2'b00: o = 4'h0;
```

```
            2'b01: o = 4'h1;
```

```
            2'b10: o = 4'h2;
```

```
            2'b11: o = 4'h4;
```

```
            default: begin
```

```
                $display("error"); o = 4'h0;
```

```
            end
```

```
        endcase
```

```
    endmodule
```

```
module dec2_4 beh1(0,i);
```

```
    Output[3:0] o;
```

```
    input [1:0] i;
```

```
    reg [3:0] o;
```

```
    always @ (i)
```

```
        begin Case(i)
```

```
            2'b00: o[0] = 1'b1;
```

```
            2'b01: o[1] = 1'b1;
```

```
            2'b10: o[2] = 1'b1;
```

```
            2'b11: o[3] = 1'b1;
```

```
            2'b0x, 2'b1x, 2'bx0, 2'bx1: o = 4'b0000; default:
```

```
                begin
```

```
                    $display("error"); o = 4'h0;
```

```
                end
```

```
            endcase
```

```
    endmodule
```

```
module alubeh(c,s,a,b,f);
```

```
    Output[3:0] c; Output s;
```

```

input [3:0] a, b;
output [1:0] f; reg s;
reg [3:0] c; always @(a or b or f)
begin case (f)
    2'b00: c = a+b;
    2'b01: c = a-b;
    2'b10: c = a&b;
    2'b11: c = a|b;
endcase end
endmodule.

```

Casex and Casez

The Case statement executes a multiway branching where every bit of the Case expression contributes to the branching decision. The statement has two variants where some of the bits of the Case expression can be selectively treated as don't cares - that is ignored. Casez allows z to be treated as don't care. "?" character also can be used in place of z. Casex treats x or z as a don't care.

```

module pri_enc(a,b);
    output [1:0] a;
    input [3:0] b; reg [1:0] a;
    always @(b) casez(b)
        4'bzzz1: a = 2'b00;
        4'bzz10: a = 2'b01;
        4'bz100: a = 2'b10;
        4'b1000: a = 2'b11; endcase
    endmodule.

```

SIMULATION

Verilog has to be an inherently parallel processing language. The fact that all the elements of a digital circuit function and interact continuously is structured through the following.

Simulation-time: Simulation is carried out in simulation time.

The simulator functions with simulation-time advancing in discrete steps.

→ At every simulation step a number of active events are sequentially carried out.

→ The simulator maintains an event queue - called the "Stratified Event Queue" with an active segment at its top. The top most event in the active segment of the queue is taken up for execution next.

→ The active event can be of an update type or evaluation type. The evaluation event can be for evaluation of variables, values on nets, expressions, etc. Refreshing the queue and rearranging it constitutes the update event.

→ Any updating can call for a subsequent evaluation and vice versa.

→ Only after all the active events in a time step are executed, the simulation advances to the next time step.

Completion of the sequence of operations above at any time step signifies the parallel nature of the HDL. A number of active events can be present for execution at any simulation time steps; all may vie for "attention". Amongst these, an event specified at #0 time is scheduled for execution at the end.

if AND if-else CONSTRUCTS

The if construct checks a specific condition and decides execution based on the result. The structure of a segment of a module with an if statement. After execution of assignment1, the condition specified is checked. If it is satisfied assignment2, is executed; if not, it is skipped. In either case the execution continues through assignment3, assignment4 etc. Execution of assignment2 alone is dependent on the condition.

The rest of Sequence remains.

(16)

```
...  
assignment1; if (condition)  
assignment2; assignment3;  
assignment4;  
...
```

Use of the if-else Construct

```
...  
assignment1;  
if (condition) begin //  
Alternative 1  
assignment2;  
assignment3; end else  
begin // alternative 2  
assignment4;  
assignment5; end  
assignment6;  
...  
...
```

After the execution of assignment1, if the condition is verified or satisfied, alternative1 is following and assignment2 and assignment3 are executed. Assignment4 and assignment5 are skipped and execution proceeds with assignment6.

If the condition is not satisfied, assignment2 and assignment3 are skipped and assignment4 and assignment5 are executed. Then execution continues with assignment6.

```
module demux(a,b,s); Output [3:0]a; input b;  
input [1:0]s; reg [3:0]a; always @(b or s)  
begin if (s == 2'b00) begin a[2'b0] = b;  
a[3:1] = 3'bzz  
end else  
if (s == 2'b01) begin  
a[2'd1] = b;
```


end else if (s == 2'b10) begin
a[2'd2] = b;

(17)

{a[3], a[1], a[0]} = 3'bzzz;

end else begin a[2'd3] = b;

a[2:0] = 3'bzzz;

end

end

endmodule

// Counter using if else if; module

Counterif(a, clk); output[7:0]a; input

clk; reg[7:0]a, n; initial begin

n = 8'h0a; a = 8'b00000000;

#45 n = 8'h23;

end always @ (posedge clk) begin

\$write ("time = %0d", \$time);

if (a == n) a = 8'h00, else

a = a + 1'b1;

end endmodule

assign - deassign CONSTRUCT

The assign - deassign constructs allow continuous assignments within a behavioural block.

always @ (posedge clk) a = b;

By way of execution, at the positive edge of clk the value of b is assigned to variable a, and a remains frozen at that value until the next positive edge of clk. Changes in b in the interval are ignored.

Consider the block always @ (posedge clk)

assign c = d;

Here at the positive edge of clk, c is assigned the value of d in a continuous manner; subsequent changes in d are directly reflected as change in variable c; The assignment here is

akin to a direct electrical connection to c from d established at the positive edge of clk
(18)
assign c = d;

Consider an enhanced version of the above block as

Always Begin

@ (posedge
clk) assign c = d;

@ (negedge
clk) deassign c;

end.

The above block signifies two activities:

→ At the positive edge of clk, c is assigned the value of d in a Continuous manner.

→ At the following negative edge of clk, the continuous assignment to c is removed; subsequent changes to d are not passed on to c; it is as though c is electrically disconnected from d.

In short, assign allows a variable or a net change in the sensitivity list to mandate a subsequent continuous assignment within. deassign terminates the assignment done through the assign-based statement.

```
module demux1(a0, a1, a2, b, s); output  
a0, a1, a2, a3; input b; input [1:0] s; reg a0, a1, a2, a3;  
always @ (s) if (s == 2'b00) assign  
{a0, a1, a2, a3} = {b, 3'b0};
```

```
else if (s == 2'b01) assign  
{a0, a1, a2, a3} = {1'b1, b, 2'b0};
```

```
else if (s == 2'b10) assign  
{a0, a1, a2, a3} = {2'b0, b, 1'b1};
```

```
else if (s == 2'b11)  
assign {a0, a1, a2, a3} = {3'b0, b};
```

```
endmodule.
```

D flip-flop through assign - deassign Constructs module (19)

```
dff assign (q, qb, di, clk, clr, pr); output q, qb; input di, clk,
clr, pr; reg q; assign
```

```
qb = ~q; always@ (clr or pr) begin if (clr) assign q = 1'b0;
clear and if (pr) assign q = 1'b1;
```

-the synchronous behaviour

```
end always@ (posedge clk) q = di;
(clocked) value assigned to q
```

```
endmodule
```

repeat CONSTRUCT

The repeat Construct is used to repeat a specified block a specified number of times. The quantity a can be a number or an expression evaluated to a number. As soon as the repeat statement is encountered, a is evaluated. The following block is executed "a" times. If "a" evaluates to 0 or n or z, the block is not executed.

Structure of a repeat block.

```
...
repeat (a) begin
assignment 1;
assignment 2;
...
end
...
```

-A module to illustrate the use of the repeat Construct

```
module trial_8b; reg [7:0] m[15:0]; integer i; reg
clk; always begin repeat (8) begin
```

```
@ (negedge clk)
```

```
m[i] = i * 8, i = i + 1;
```

```
end repeat (8)
```

```
begin
```

```
@ (negedge clk) i = i - 1;
```

```
$display("t = %.0d, i = %.0d, m[i] = %.0d",
```


\$ time, i, m[i]); end

(20)

end

initial begin clk = 1'b0; i = 0;

#70 \$ stop; end

always #2 clk = ~clk;

endmodule

for loop

The for loop in Verilog is quite similar to the for loop in C; the format of the for loop is

```
....  
for (assignment1; expression; assignment2) statement;  
....
```

It has four parts; the sequence of execution is as follows;

1. Execute assignment1
2. Evaluate expression
3. If the expression evaluates to the true state (1), carry out statement. Go to step 5.
4. If expression evaluates to the false state (0), exit the loop.
5. Execute assignment2. Go to step 2.

```
reg [8:0] c; reg co; reg [7:0] s; integer i;  
always @ (posedge en) begin c[0] = cin;  
for (i=0; i<=7; i=i+1) begin  
    {c[i+1], s[i]} = {a[i] + b[i] + c[i]};  
end co = c[8];
```

end

endmodule.

THE disable CONSTRUCT

(21)

There can be situations where one has to break out of a block or loop. The disable statement terminates a named block or task. Control is transferred to the statement immediately following the block. Conditional termination of a loop, interrupt servicing etc... are typical contexts is functionally similar to use. Often the disabling is carried out from within the block itself. The disable construct is functionally similar to the break in C. OR gate module to demonstrate the use of the disable construct.

```
module or_gate (b, a, en); input [3:0] a; input en; output b; reg b; integer i;
```

```
    always@ (posedge en)
    begin i = 0; OR_gate b = 1'b0;
    for (i = 0; i <= 3; i = i + 1) if (a[i] = 1'b1)
    begin b = 1'b1; disable OR_gate
    end
```

```
    end
end module
```

The disable statement has to have a block identifier tagged to it in this respect it differs from "break" in C.

Once encountered, it terminates execution of the block; the following statements within the block are not executed.

Typically it can be used to handle exceptions to regularly assigned activities for example, interrupt, hold, reset, etc.

WHILE LOOP

The format for the while loop is shown as while (expression) statement;

The boolean expression is evaluated. If it is true, the statement (or block of statements) is executed and expression evaluated and checked. If the expression evaluates to false, the loop is terminated and the following statement is taken for execution. If the expression evaluates to true, execution of statement is repeated. Thus the loop is terminated and broken only if the expression evaluates to false.

To generate a pulse of definite width.

```
module while (b, n, en, clk); input [7:0] n; input
    clk, en; output b; reg [7:0] a; reg b;
    always @ (posedge en) begin a = n;
    while (1a) begin b = 1'b1; @ (posedge clk)
        a = a - 1'b1;
        end b = 1'b0
    end initial
endmodule.
```

Forever LOOP

Repeated execution of a block in an endless manner is best done with the forever loop.

module to generate a clock waveform using the forever Construct

```
module clk; reg clk, en; always @ (posedge en)
    forever #2
        clk = ~clk
    initial begin clk = 1'b0; en = 1'b0; #1 clk = 1'b1;
    #4 en = 1'b1; #30 $ stop;
```


end initial \$ monitor ("CLK = %b, t = %0d, en = %b," (23)

CLK, \$time, en);

endmodule

PARALLEL LOOP BLOCKS

All the procedural assignments within a begin-end block are executed sequentially. join block is an alternate one where all the assignments are carried out concurrently. One can use a fork-join block within a begin-end block or vice versa.

<pre> module fk_jn-a; integer a; initial begin a = 0; #1 a = 1; #2 a = 2; #3 a = 3; #4 \$ stop; end initial \$ monitor ("a = %0d, t = %0d", a, \$time); end module </pre>	<pre> module fk_jn-b; integer a; initial fork a = 0; #1 a = 1; #2 a = 2; #3 a = 3; #4 \$ stop; join initial \$ monitor ("a = %0d, t = %0d", a, \$time); endmodule </pre>
<p>// Simulation results</p> <pre> # a=0, t=0 # a=1, t=1 # a=2, t=3 # a=3, t=6 </pre>	<p>// Simulation results</p> <pre> # a=0, t=0 # a=1, t=1 # a=2, t=2 # a=3, t=3. </pre>

Force - release CONSTRUCT

(21)

when debugging a design with a number of instantiations, one may be stuck with an unexpected behaviour in a localized area. Tracing the paths of individual signals and debugging the design may prove to be too tedious or difficult. In such cases suspect blocks may be isolated, tested, and debugged and status quo ante established. The force-release Construct is for such a localized isolation for a limited period.

force a = 1'b0;

forces the variable a to take the value 0.

force b = c & d;

forces the variable b to the value obtained by evaluating the expression c & d.

The force-release Construct is similar to the assign-deassign Construct. The latter Construct is for Conditional assignment in a design description. The force-release Construct is for "short time" assignments in a test-bench. Synthesis tools will not support the force-release Constructs.

→ The force-release Construct is equally valid for net-type variables and reg-type variables. The net-type variables revert to their normal values on release. With reg-type variables the value forced remains until another assignment to the reg.

→ The variable, on which the values are forced during testing, must be properly dereferenced.

→ In the illustration above, each variable was forced one at a time. It was done only to simplify the illustration sequence and focus attention on the possible use of the Construct. In practice, different variables can be forced together before the special debug sequence. Their release too can be together.

OR gate module and its test bench to illustrate the use of force - release Construct (25)

```
module or_tr_rl(a,b,c); input b,c; output a; wire a,b,c;  
assign a = b|c; initial begin
```

```
#1 $display("display: time = %0d, b = %b, c = %b, a = %b",  
            $time, b, c, a);
```

```
#6 force b = 1'b1;
```

```
#1 $display("display: time = %0d, b = %b, c = %b,  
a = %b", $time, b, c, a);
```

```
#6 release b;
```

```
#1 $display("display: time = %0d, b = %b, c = %b, a = %b",  
            $time, b, c, a); end
```

```
endmodule.
```

EVENT

The keyword event allows an abstract event to be declared. The event is not a data type with any specific values, it is not a variable (reg) or a net. It signifies a change that, can be used as a trigger to communicate between modules or to synchronize events in different modules.

```
...  
event changes
```

```
...  
always
```

```
... change
```

```
...  
always @ change
```

```
...
```

In the course of execution of an always block, the event is triggered. The operator signifies the triggering. Subsequently, another activity can be started in the module by the event

change. The always @ (change) block activities this. (26)

The event change can be used in other modules also by proper dereferencing; with such usage an activity in a module can be synchronized to an event in another module.

The event construct is quite useful, especially in the early stages of a design. It can be used to establish the functionality of a design at the behavioural level; it allows communication amongst different instantiated modules without associated inputs or outputs.

→ A module to illustrate the event construct: A serial data receiver

```
module rec (a, ddi, clk); Output [8:1] a; Input ddi, clk;
reg [8:1]
a; integer j, jj; event buf_full; always for (j=0; j<20;
j=j+1)
begin #0 jj=0; repeat (8) @ (negedge clk) begin
jj=jj+1;
a[jj]=ddi;
#5 $display ("b = %b", a[jj]); end
#0 -> buf_full; end
endmodule.
```

UNIT-IV DATAFLOW LEVEL AND SWITCH LEVEL MODELLING: Introduction, continuous assignment structures, delays and continuous assignments, assignment to vectors, basic transistor switches, CMOS switch, Bidirectional gates and time delays with switch primitives, instantiations with strengths and delays, strength contention with trireg nets

4.1 Introduction

Gate level design description makes use of the gate primitives available in Verilog. These are repeatedly and judiciously instantiated to achieve the full design description. Digital designers familiar with the basic logic gates and SSI / MSI circuits can describe the desired target circuit in terms of them on paper and proceed with the design description based on them. This was the approach followed in the last two chapters; it is practical for comparatively smaller designs – say those involving tens of gates. One can define modules in terms of primitives involving tens of gates and instantiate them in macro-modules. This increases the complexity of designs that can be handled by one order. Beyond that the gate level design description becomes too complicated to be practical.

Data flow level description of a digital circuit is at a higher level. It makes the circuit description more compact as compared to design through gate primitives. We have a number of operands and operations representing the simulations directly or indirectly. The operations are carried out on the operand(s) in singles or in combinations. The results are assigned to nets. The operand-operation-assignments representing data flow are carried out repeatedly to complete the design description. Further, these can be combined judiciously with the gate instantiations wherever necessary. With such combinations, design description of a comprehensive nature can be accommodated.

4.2 CONTINUOUS ASSIGNMENT STRUCTURES

A continuous assignment is the most basic statement in dataflow modeling, used to drive a value onto a net. This assignment replaces gates in the description of the circuit and describes the circuit at a higher level of abstraction. The assignment statement starts with the keyword **assign**. The syntax of an **assign** statement is as follows.

continuous_assign ::= assign [drive_strength] [delay3] list_of_net_assignments ;

A simple two input AND gate in data flow format has the form

assign c = a && b;

Here “**assign**” is the keyword carrying out the assignment operation. This type of assignment is called a continuous assignment.

a and b are operands – typically single-bit logic variables.

“&&” is a logic operator. It does the bit-wise AND operation on the two operands a and b.

“=” is an assignment activity carried out.

c is a net representing the signal which is the result of the assignment.

In general, an **operand** can be of any one of the following types:

- ✓ A constant number [including real].
- ✓ Net of a scalar or vector type including part of a vector.
- ✓ Register variable of a scalar or vector type including part of a vector.
- ✓ Memory element.

A call to a function that returns any of the above. The function itself can be a user-defined or of a system type .

There are other types of operators as well . All types of combinational circuits can be modeled using continuous assignments. One need not necessarily resort to instantiation of gate primitives.

An AND gate module which uses the above assignment is shown in Figure 4.1. The test bench for the same is shown in Figure 4.2, and the waveforms of nets a, b, and c obtained with the simulation are shown in Figure 4.3. [The simulation software used has the facility to capture the waveforms of selected signals in the “run” phase; this has been invoked to get the waveforms in Figure 4.3. No separate **\$monitor** command is included in the test bench of Figure 4.2. The same approach has been adopted with many of the test benches.

```
module andgdf(c,a,b);  
output c;  
input a,b;  
wire c;  
assign c = a&& b;  
endmodule
```

fig 4.1 :A module with an AND gate at the data flow level.

```
//TESTBENCH  
module tst_andgdf;  
reg a,b;  
wire c;  
initial  
begin  
a = 1'b0; b = 1'b0; #4 a = 1'b1;  
#4 b = 1'b1; #4 a = 1'b0; #4 b = 1'b0; #4 a = 1'b1;  
end  
andgdf g1(c,a,b);  
initial #20 $stop;  
endmodule
```

fig 4.2 A test bench for the above module

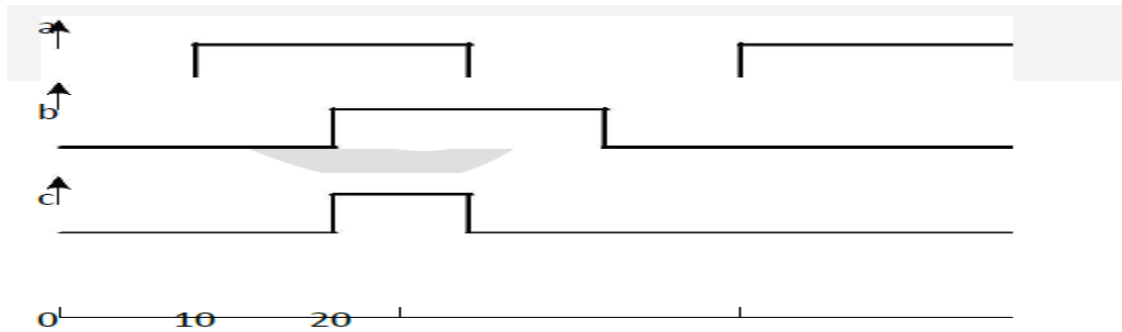
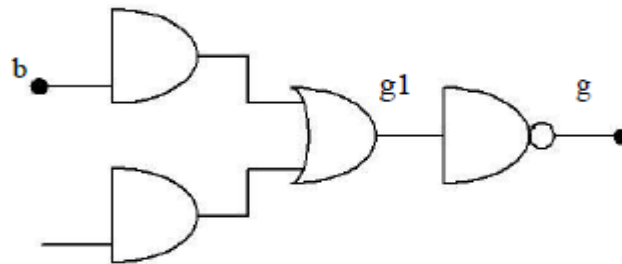


Fig4.3 :Waveforms of nets a, b, and c obtained with the simulation of the module



An A-O-I gate circuit.

Fig 4.4 :

Multiple assignments can be carried out through a direct extension of the structure adopted in the above case. Consider the AOI gate in Figure 4.4. A few patterns of the assignments for the circuit are given in Figure 4.5 to Figure 4.7.

✓ `assign e = a&&b, f = c&&d, g1 = e|f, g = ~g1;`

Figure 4.5 A data flow level assignment statement to realize the A-O-I gate in Figure

✓ `assign e = a & b, f = c & d; assign g1 = e|f, g = ~g1;`

Figure 4.6 Another set of data flow level assignment statements to realize the A-O-I gate in Figure 4.4.

```
assign e = a & b;
assign f = c & d;
assign g1 = e ! f;
assign g = ~g1;
```

Figure 4.7 Yet another set of data flow level assignment statements to realize the A-O-I gate in Figure 4.4

Observations:

- ✓ **The semicolon** terminates an assignment statement. **Commas separate** different assignments in an assignment statement.
- ✓ “|” is the bit-wise OR operator and “~” the bit-wise negation operator in Verilog.
- ✓ All the quantities in the left-hand side of a continuous assignment have to be of net type. Thus e, f, g, and g1 have to be declared as nets.
- ✓ All the operations in an assignment are evaluated whenever any of the operands in the assignment changes value. Further, all the assignments are carried out concurrently. Hence the order of the assignments or the statements is immaterial.
- ✓ The right-hand sides of assignment statements can be nets, regs, or function calls. Here a, b, c, and d can be nets or regs. All other variables have to be nets.

The module for the A-O-I gate of Figure 4.4 is given in Figure 4.8 – it is formed around the assignment statement of Figure 4.5. The same can be tested through a test bench.

4.2.1 Assignment and Net Declarations

The assignment statement can be combined with the net declaration itself making the assignment implicit in the net declaration itself. Thus the two statements

```
wire c;  
  
assign c = a & b;
```

can be combined as

```
wire c = a & b;
```

The above simplification cannot be carried over to multiple declarations. With this provision, the module of Figure 4.8 can be modified as shown in Figure 4.9.

In the modules of Figures 4.8 and 4.9, a, b, c, and d are declared as input and g as output. These would be taken as nets if there are no separate declarations concerning their types.

However, the intermediate quantities – e, f, and g1– should be declared as wire. Synthesized version of the A-O-I circuit is shown in Figure 4.10.

```
module aoi2(g,a,b,c,d);  
  
output g;  
  
input a,b,c,d;  
  
wire e,f,g1,g;  
  
assign e = a && b,f = c && d, g1 = e | f, g=~g1;  
  
endmodule
```

Figure 4.8 A compact description of the AOI module at the data flow level.


```

module aoi3(g,a,b,c,d);
    output g;
    input a,b,c,d;

    wire g;

    wire e = a && b;
    wire f = c && d;
    wire g1 = e || f;
    assign g = ~g1;
endmodule

```

Figure 4.9 Alternate design module to realize the A-O-I gate in Figure 4.4.

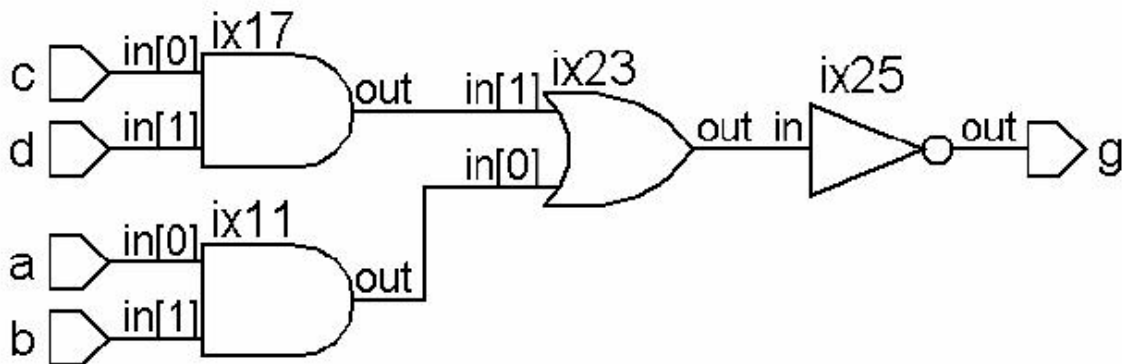


Figure 5.10 Synthesized circuit of the A-O-I gate module of Figure 5.9.

4.2.2 Continuous Assignments and Strengths

A net to which a continuous assignment is being made can be assigned strengths for its logic levels. The procedure is akin to the strength allocation to the outputs of primitives. The AOI gate of Figure 4.9 is modified with strength allocations to the output and is shown in Figure 4.11. The assignment to g can be combined with the wire declaration into a single statement as

wire (pull1, strong0)g = ~g1;

As mentioned earlier, one can have only one assignment in the statement here. In a bigger design, g in Figure 4.11 can be assigned to other expressions or primitives also. Any resulting contention in the output values will be resolved on the lines.

```

module aoi4 (g, a, b, c, d);
    output g;
    input a, b, c, d;
    wire g;
    wire e = a && b;
    wire f = c && d;

```

```

wire g1 = e || f;
assign (pull1, strong0)g = ~g1;
endmodule

```

Figure 4.11 The module of Figure 5.9 modified with strength allocation to the output.

4.3 DELAYS AND CONTINUOUS ASSIGNMENTS

Delays can be incorporated at the data flow level in different ways . Consider the combination of statements in Figure 4.12. The assignment takes effect with a time delay of 2 time steps. If a or b changes in value, the program waits for 2 time steps, computes the value of c based on the values of a and b at the time of computation, and assigns it to c. In the interim period, a or b may change further, but c changes and takes the new value only 2 time steps after the change in a or b initiates it. Typical waveforms for a, b, and c are shown in Figure 4.13. Note that the changes in a and b of duration less than 2 time steps are ignored *vis-à-vis* assignment to the net c. The following may be noted with respect to the waveforms:

- ✓ a changes at 0 ns, 2 ns, 5 ns, 8 ns, 9 ns, 12 ns and 13 ns;
- ✓ b changes at 0 ns, 2 ns, 5 ns, 8 ns and 13 ns.
- ✓ All these trigger changes to c.
- ✓ In every case change to c comes into effect with a time delay of 2 time steps – that is, at the 2nd, 4th, 7th, 8th, 10th, 11th, 14th and 15th ns, respectively.
- ✓ Whenever c changes, its new value is decided by the values of a and b at that instant of time. In effect, c changes at 2nd, 4th and 7th ns only.

```

wire c, a, b;

assign #2 c = a & b;

```

Figure 4.12 Illustration of combining delays with assignments.

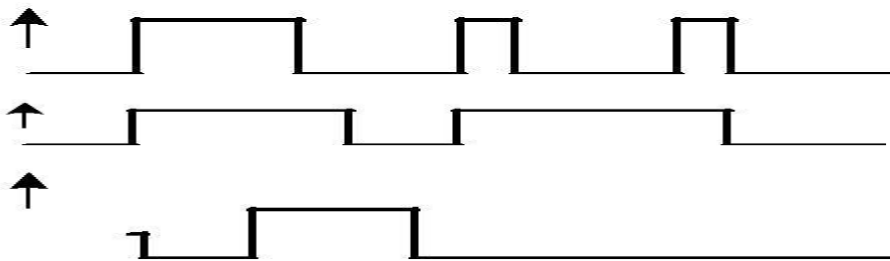


FIG 4.13 Waveforms of signals a, b, and c for the design segment of Figure 5.12

The program segment in Figure 4.14 also gives the same output as shown in Figure 4.13. If the time delay is in the net and not in the assignment proper, its effect is not any different. Consider the program segment in Figure 4.15. Here the changes in the values of d are computed immediately following those in a and b. The assignment takes effect immediately. The delay in the net c causes a delay of 2 time steps in the assignment to c. Such a delay is not present for d. Typical waveforms for the program segment are shown in Figure 4.15.

Note the following:

- ✓ a changes at 2 ns, 5 ns, 8 ns, 9 ns, 12 ns and 13 ns;
- ✓ b changes at 2 ns, 5 ns, 8 ns and 13 ns.
- ✓ All these trigger changes to c and d also.
- ✓ In every case, change to c comes into effect with a time delay of 2 time steps that is, in effect, c changes at 2nd, 4th and 7th ns only.
- ✓ Whenever c changes, its new value is decided by the values of a and b at that instant of time.
- ✓ In every case, changes to d come into effect immediately.

wire a, b;

wire #2 c = a & b;

fig 4.14 :Alternate description for the program segment of Figure 5.10.

wire a, b, d;

wire #2 c;

assign c = a & b;

assign d = a & b;

Figure 4.15 Illustration of combining delays with assignments.

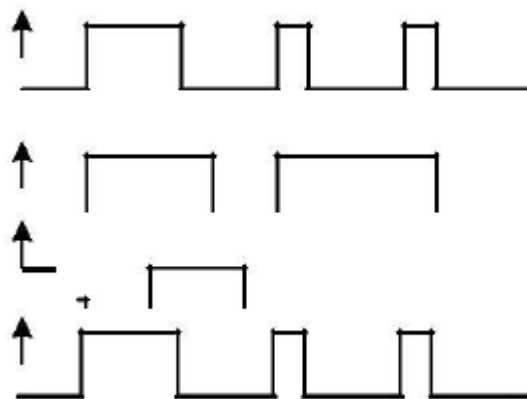


Figure 4.16 Waveforms of Signals a, b, c, and d for the design segment of Figure 5.15.

4.4 ASSIGNMENT TO VECTORS

The continuous assignments are equally applicable to vectors. A single statement can describe operations involving vectors wherever possible. This is illustrated in the adder module in Figure 4.17, which adds two 8-bit numbers. Here it is assumed that the sum is also of 8 bits.

However to account for the possibility of a carry bit being generated in the course of the addition process, it is desirable to increase the vector size of c by one bit.

4.4.1 Concatenation of Vectors

- ✓ One can concatenate vectors, scalars, and part vectors to form other vectors.
- ✓ The concatenated vector is enclosed within braces.
- ✓ Commas separate the components –scalars, vectors, and part vectors.
- ✓ If a and b are 8- and 4-bit wide vectors, respectively and c is a scalar

{a, b, c}

stands for a concatenated vector of 13 bits width. The vector components are formed in the order shown – c is the least significant bit and a[7] the most significant bit and the other bits are in between in the order specified. The concatenation can be with selected segments of vectors also.

For example,

{a(7:4), b(2:0)}

represents a 7-bit vector formed by combining the 4 most significant bits of vector a with the 3 least significant bits of vector b.

The size of each operand within the braces has to be specified fully to form the concatenated vector. Hence unsized constant numbers cannot be used as operands here.

Example 5.1 Eight-Bit Adder

Figure 4.18 shows the design description of an 8-bit adder, where the output vector is formed directly by concatenation.

The adder takes a carry input and gives out a carry output. The adder module here can form the “seed” adder block in a multi-byte adder chain.

```
module add_8(a,b,c);  
  input[7:0]a,b;  
  output[7:0]c;  
  assign c = a + b ;  
endmodule
```

Figure 4.17 An adder module at data flow level where the nets are vectors.

```

module add_8_c(c,cco,a,b,cci);
  input[7:0]a,b;
  output[7:0]c;
  input cci;
  output cco;
  assign {cco,c} = (a + b + cci);
endmodule

```

Figure 4.18 A complete 8-bit adder module at data flow level.

When it is necessary to replicate vectors, scalars, *etc.*, to form other vectors, the same can be arrived at in a compact manner using the repetition multiplier again through concatenation. Thus,

$\{2\{p\}\}$ represents the concatenated vector $\{p, p\}$

$\{2\{p\}, q\}$ represents the concatenated vector $\{p, p, q\}$.

The two statements

assign GND=supply0;

$p=\{8\{GND\}\}$; together ground the 8 bits of the vector p.

Concatenation operation can be nested to form bigger vectors when component combinations are repeated. For example,

$$\{a, 3 \{2\{b, c\}, d\}\}$$

is equivalent to the vector

$$\{a, b, c, b, c, d, b, c, b, c, d, b, c, b, c, d\}$$

ALU

Figure 4.20 shows an ALU module. It is built around a single executable statement present as a continuous assignment. A test bench for the ALU is also shown in the figure.

The synthesized circuit is shown in Figure 4.21. Results of running the test bench are shown in Figure 4.22. Some of the combinational circuit operations required are realized inside the “modgen” blocks of the FPGA used.

The nature of the ALU description in the module decides the translation into circuit. Contrast this with the ALU considered at the gate level of design where each functional block is instantiated separately and the selected set of outputs steered to the final output.

Each such instantiated module translates into a separate circuit block. Their outputs are mux’ed into the final output vector.

There is a one-to-one correspondence between the elements of the design description and their respective realizations

```

module alu_df1 (d, co, a, b, f, cci);    //a SIMPLE ALU FOR ILLUSTRATION PURPOSES

    output [3:0] d;
    output co;
    wire[3:0]d;
    wire co;
    input cci;
    input [3 : 0 ] a, b;
    input [1 : 0] f;        //f is a two-bit function select input;
assign {co,d}=(f==2'b00)?(a+b+cci):((f==2'b01)?(a-b) :((f==2'b10)? {1'bz,a^b}:{1'bz,~a}));

/*co is the carry bit in case of addition; it is the borrow bit in case of subtraction. In the other two
cases, co is not required. Hence it is assigned z value.*/

endmodule


//test-bench

module tst_aludf1;
reg [3:0]a,b;
reg[1:0] f;
reg cci;
wire[3:0]d;
wire co;
alu_df1 aa(d,co,a,b,f,cci);
    initial
        begin
            cci= 1'b0; f = 2'b00; a = 4'b0;
            b = 4'h0;
        end

    always
    begin

        #2 cci = 1'b0; f = 2'b00; a = 4'h1; b = 4'h0; #2 cci = 1'b1; f = 2'b00; a = 4'h8; b = 4'hf; #2 cci = 1'b1; f =
        2'b01; a = 4'h2; b = 4'h1; #2 cci = 1'b0; f = 2'b01; a = 4'h3; b = 4'h7; #2 cci = 1'b1; f = 2'b10; a = 4'h3; b = 4'h3;
        #2 cci = 1'b1; f = 2'b11; a = 4'hf; b = 4'hc;

    end

        initial $monitor($time, " cci = %b , a= %b , b = %b , f = %b , d = %b , co= %b ", cci , a, b, f, d, co);

        initial #30 $stop;

endmodule

```

Figure 4.20 A 4-bit 4-function ALU and a test bench for the same.

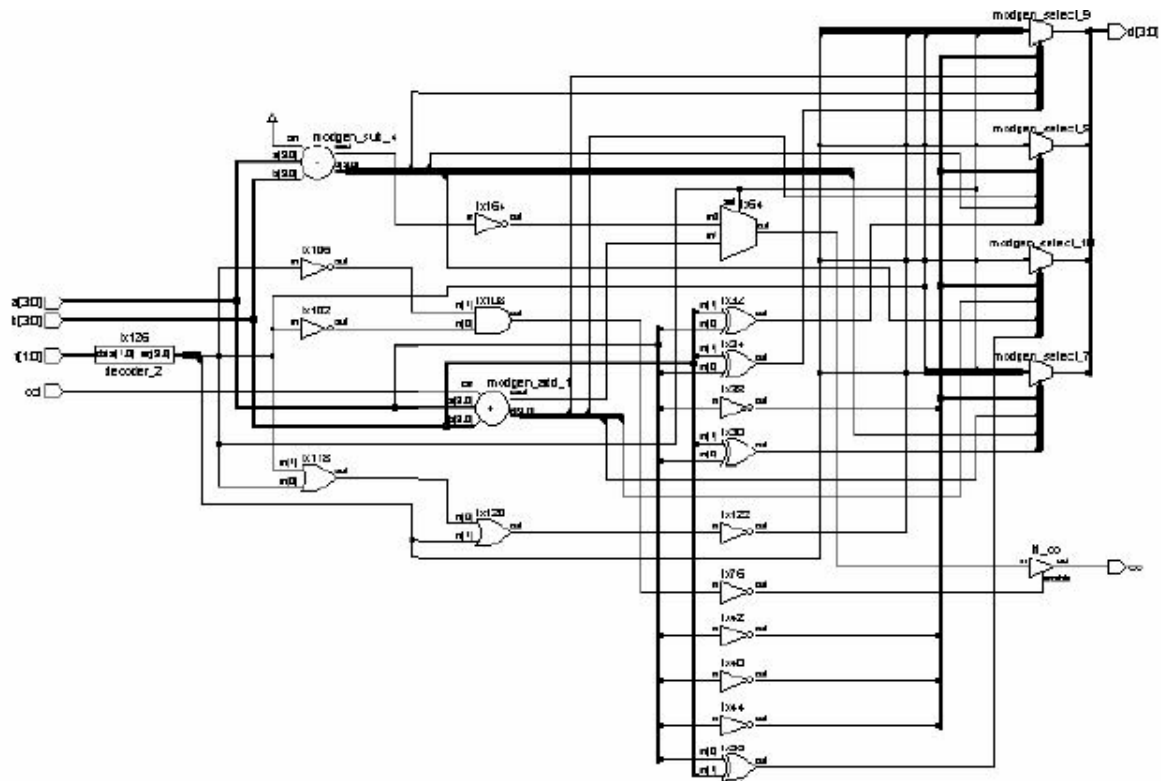


Figure 4.21 Synthesized circuit of the ALU in Example 4.18.

```
# 0 cci = 0 , a= 0000 ,b = 0000 ,f = 00 ,d =0000 ,co= 0
# 2 cci = 0 , a= 0001 ,b = 0000 ,f = 00 ,d =0001 ,co= 0
# 4 cci = 1 , a= 1000 ,b = 1111 ,f = 00 ,d =1000 ,co= 1
# 5 cci = 1 , a= 0010 ,b = 0001 ,f = 01 ,d =0001 ,co= 0
# 8 cci = 0 , a= 0011 ,b = 0111 ,f = 01 ,d =1100 ,co= 1 #10 cci = 1 , a= 0011 ,b = 0011 ,f = 10 ,d =0000 ,co= z
#12 cci = 1 , a= 1111 ,b = 1100 ,f = 11 ,d =0000 ,co= z #14 cci = 0 , a= 0001 ,b = 0000 ,f = 00 ,d =0001 ,co=
0 #15 cci = 1 , a= 1000 ,b = 1111 ,f = 00 ,d =1000 ,co= 1 #18 cci = 1 , a= 0010 ,b = 0001 ,f = 01 ,d =0001
,co= 0 #20 cci = 0 , a= 0011 ,b = 0111 ,f = 01 ,d =1100 ,co= 1 #22 cci = 1 , a= 0011 ,b = 0011 ,f = 10 ,d
=0000 ,co= z #24 cci = 1 , a= 1111 ,b = 1100 ,f = 11 ,d =0000 ,co= z #25 cci = 0 , a= 0001 ,b = 0000 ,f = 00
,d =0001 ,co= 0 #28 cci = 1 , a= 1000 ,b = 1111 ,f = 00 ,d =1000 ,co= 1
```

Figure 4.22 Results of running the test bench for the ALU module in Figure 4.20

basic transistor switches, CMOS switch, Bidirectional gates and time delays with switch primitives, instantiations with strengths and delays, strength contention with trireg nets

Introduction

In today's environment the MOS transistor is the basic element around which a VLSI is built. Designers familiar with logic gates and their configurations at the circuit level may choose to do their designs using MOS transistors. Verilog has the provision to do the design description at the switch level using such MOS transistors.

Switch level modeling forms the basic level of modeling digital circuits. The switches are available as primitives in Verilog; they are central to design description at this level. Basic gates can be defined in terms of such switches. By repeated and successive instantiation of such switches, more involved circuits can be modeled – on the same lines as was done with the gate level primitives.

BASIC TRANSISTOR SWITCHES

Consider an NMOS transistor of the depletion type. When used in a digital circuit, it can be in one of three modes:

- ✓ $V_G < V_S$ where V_G and V_S are the gate and source voltages with respect to the drain: The transistor is OFF and offers very high impedance across the source and the drain. It is in the z state.
- ✓ $V_G = V_S$: The transistor is in the active region. It presents a resistance between the source and the drain. The value depends on the technology. Such a resistive state of the transistor can be modeled in Verilog. A transistor in this mode can be represented as a resistance in Verilog – as pull1 or pull0 depending on whether the drain is connected to supply1 or source is connected to supply0.
- ✓ $V_G > V_S$: The transistor is fully turned on. It presents very low resistance between the source and drain.

An enhanced-mode NMOS transistor also has the above three modes of operation.

- ✓ It is OFF when $V_G = V_S$. It is moderately ON or in the active region when V_G is slightly greater than V_S , representing a resistive (pull1 or pull0) mode of operation. When V_G is sufficiently greater than V_S , the transistor is in the on state representing very low resistance. Similar modes are possible for the PMOS transistor also.

Basic Switch Primitive

Different switch primitives are available in Verilog.

Consider an nmos switch. A typical instantiation has the form

nmos (out, in, control);

- ✓ nmos – a keyword – represents an NMOS transistor functioning as a switch.
- ✓ The switch has three terminals – in, out, and control.

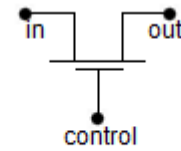


Figure 1 An NMOS switch with terminals

- When the control input is at 1 (high) state, the switch is on. It connects the input lead to the output side and offers zero impedance.
- When the control input is low, the switch is OFF and output is left floating (z state).
- If the control is in the z or the x state, output may take corresponding values.

The keyword pmos represents a PMOS transistor functioning as a switch.

The PMOS switch has three terminals (see Figure 2).

A typical instantiation of the switch has the form

pmos (out, in, control);

- When the control is at 1 (high) state, the switch is off. Output is left floating.
- When control is at 0 (low) state, the switch is on, input is connected to output, and output is at the same state as input.

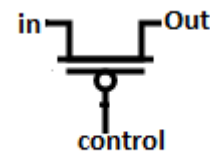


fig. 2 Pmos with 3-terminals

Resistive Switches

nmos and **pmos** represent switches of low impedance in the on-state. **rnmos** and **rpmos** represent the resistive counterparts of these respectively. Typical instantiations have the form

rnmos (output1, input1, control1);

rpmos (output2, input2, control2);

- The rnmos if the control1 input is at 1 (high) state, the switch is ON and functions as a definite resistance. It connects input1 to output1 through a resistance. When control1 is at the 0 (low) state, the switch is OFF and leaves output1 floating.
- The rpmos switch is ON when control2 is at 0 (low) state. It inserts a definite resistance between the input and the output signals but retains the signal value.

rpmos and rnmos are resistive switches, they reduce the signal strength when in the on state. The reduced strength is mostly one level below the original strength.

The rpmos and rnmos switches function as unidirectional switches; the signal flow is from the input to the output side.

pullup and pulldown

A MOS transistor functions as a resistive element when in the active state. Realization of resistance in this form takes less silicon area in the IC as compared to a resistance realized directly. pullup and pulldown represent such resistive elements.

A typical instantiation here has the form

pullup (x);

Here the net x is pulled up to the supply1 through a resistance. Similarly, the instantiation

pulldown (y);

pulls y down to the supply0 level through a resistance. The pullup and pulldown primitives can be used as loads for switches or to connect the unused input ports to VCC or GND, respectively. They can also form loads of switches in logic circuits.

The default strengths for pullup and pulldown are pull1 and pull0 respectively. One can also specify strength values for the respective nets. For example,

pullup (strong1) (x)

specifies a resistive pullup of net x to supply1. One can also assign names to the pullup and pulldown primitives. Thus

pullup (strong1) rs(x)

represents an instantiation of pullup designated rs having strength strong1.

CMOS SWITCH

A CMOS switch is formed by connecting a PMOS and an NMOS switch in parallel – the input leads are connected on the one side and the output leads are connected together on the other side. Figure 10.15 shows the switch so formed. It has two control inputs:

- N_control turns ON the NMOS transistor and keeps it ON when it is in the 1 state.
- P_control turns ON the PMOS transistor and keeps it ON when it is in the 0 state.

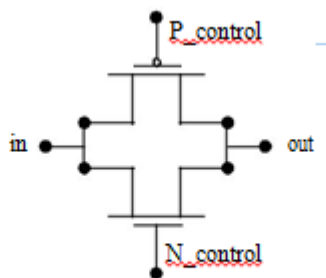


Figure 10.15 A CMOS switch formed by connecting a PMOS transistor and an NMOS transistor in parallel.

The CMOS switch is instantiated as shown below.

```
cmos csw (out, in, N_control, P_control );
```

Significance of the different terms is as follows:

cmos: The keyword for the switch instantiation

csw: Name assigned to the switch in the instantiation

out: Name assigned to the output variable in the instantiation

in: Name assigned to the input variable in the instantiation

N_control: Name assigned to the control variable of the NMOS transistor in the instantiation

P_control: Name assigned to the control variable of the PMOS transistor in the instantiation

```
module cmos(out,in,nctr,pctr);
```

```
    input in,nctr,pctr;
```

```
    output out;
```

```
    nmos gn(out,in,nctr);
```

```
    pmos(out,in,pctr);
```

```
endmodule
```

BI-DIRECTIONAL GATES

Verilog has a set of primitives for bi-directional switches as well. They connect the nets on either side when ON and isolate them when OFF. The signal flow can be in either direction. None of the continuous-type assignments at higher levels dealt with so far has a functionality equivalent to the bi-directional gates. There are six types of bi-directional gates.

- tran rtran
- tranif1 rtanif1
- tranif0 rtranif0

tran and rtran

The tran gate is a bi-directional gate of two ports. When instantiated, it connects the two ports directly. Thus the instantiation

```
tran (s1, s2);
```

connects the signal lines s1 and s2. Either line can be input, inout or output.

rtran is the resistive counterpart of tran.

tranif1 and rtranif1

tranif1 is a bi-directional switch turned ON/OFF through a control line. It is in the ON-state when the control signal is at 1 (high) state. When the control line is at state 0 (low), the switch is in the OFF state. A typical instantiation has the form

tranif1 (s1, s2, c);

Here c is the control line. If c=1, s1 and s2 are connected and signal transmission can be in either direction.

rtranif1 is the resistive counterpart of tranif1. It is instantiated in an identical manner.

tranif0 and rtranif0

tranif0 and rtranif0 are again bi-directional switches. The switch is OFF if the control line is in the 1 (high) state, and it is ON when the control line is in the 0 (low) state. A typical instantiation has the form

tranif0 (s1, s2, c);

With the above instantiation, if c = 0, s1 and s2 are connected and signal transmission can be in either direction. If c = 1, the switch is OFF and s1 and s2 are isolated from each other.

rtranif0 is the resistive counterpart of tranif0.

Type of Bi-directional switch	Typical instantiation	Condition to be ON	Remarks
2 port	tran (a, b);	Always ON (if instantiated)	Acts essentially as a buffer
	rtran (a, b);	– do –	Acts essentially as a buffer with reduction in the strength of the signal
3 port	tranif1 (a, b, c);	ON if c = 1	Acts as a buffer if ON. Otherwise provides isolation
	tranif0 (a, b, c);	ON if c = 0	– do –
	rtranif1 (a, b, c);	ON if c = 1	Acts as a buffer if ON. Otherwise provides isolation; signal strength on the output side is lower than that on the input side
	rtranif0 (a, b, c);	ON if c = 0	– do –

TIME DELAYS WITH SWITCH PRIMITIVES

Propagation delays can be specified for switch primitives on the same lines as was done with the gate primitives in Chapter 5. For example, an NMOS switch instantiated as

```
nmos g1 (out, in, ctrl );
```

has no delay associated with it. The instantiation

```
nmos (delay1) g2 (out, in, ctrl );
```

has delay1 as the delay for the output to rise, fall, and turn OFF. The instantiation

```
nmos (delay_r, delay_f) g3 (out, in, ctrl );
```

has delay_r as the rise-time for the output. delay_f is the fall-time for the output. The turn-off time is zero. The instantiation

```
nmos (delay_r, delay_f, delay_o) g4 (out, in, ctrl );
```

has delay_r as the rise-time for the output. delay_f is the fall-time for the output delay_o is the time to turn OFF when the control signal ctrl goes from 0 to 1. Delays can be assigned to the other uni-directional gates (rcmos, pmos, rpmos, cmos, and rcmos) in a similar manner. Bi-directional switches do not delay transmission – their rise- and fall-times are zero. They can have only turn-on and turn-off delays associated with them. tran has no delay associated with it.

```
tranif1 (delay_r, delay_f) g5 (out, in, ctrl );
```

represents an instantiation of the controlled bi-directional switch. When control changes from 0 to 1, the switch turns on with a delay of delay_r. When control changes from 1 to 0, the switch turns off with a delay of delay_f.

```
transif1 (delay0) g2 (out, in, ctrl );
```

represents an instantiation with delay0 as the delay for the switch to turn on when control changes from 0 to 1, with the same delay for it to turn off when control changes from 1 to 0. When a delay value is not specified in an instantiation, the turn-on and turn-off are ideal that is, instantaneous. Delay values similar to the above illustrations can be associated with rtranif1, tranif0, and rtranif0 as well.

INSTANTIATIONS WITH STRENGTHS AND DELAYS

In the most general form of instantiation, strength values and delay values can be combined. For example, the instantiation

```
nmos (strong1, strong0) (delay_r, delay_f, delay_o ) gg (s1, s2, ctrl) ;
```

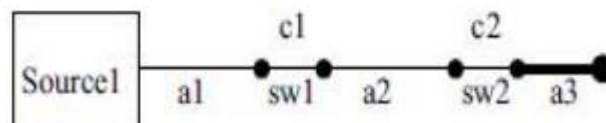
means the following:

- It has strength strong0 when in the low state and strength strong1 when in the high state.
- When output changes state from low to high, it has a delay time of delay_r.
- When the output changes state from high to low, it has a delay time of delay_f.
- When output turns-off it has a turn-off delay time of delay_o.

rnmos, pmos, and rpmos switches too can be instantiated in the general form in the same manner. The general instantiation for the bi-directional gates too can be done similarly.

STRENGTH CONTENTION WITH TRIREG NETS :

- nets declared as trireg can have capacitive storage. Such storage can be assigned one of three strengths – large, medium, or small.
- Driving such a net from different sources can lead to contention ,the relative strength levels of the sources also have a say in the signal level taken by the net.



An Example circuit to illustrate strength contention in switch primitives


```

module demo_1;
trireg(large)a3; trireg(small)a2; wire a1; reg c1,c2,b;
buf(strong1,strong0) source1(a1,b);
tranif1 sw1(a2,a1,c1), sw2(a3,a2,c2);
initial begin
    $display("t\ta1\tc1\ta2\tc2\ta3");
    #0 {c1,c2,b}=3'b111; #1 {c1,c2,b}=3'b011; #1 {c1,c2,b}=3'b001;
    #1 {c1,c2,b}=3'b000; #1 {c1,c2,b}=3'b100; #1 {c1,c2,b}=3'b000;
    #1 {c1,c2,b}=3'b010; #1 {c1,c2,b}=3'b000; #1 {c1,c2,b}=3'b100;
    #1 {c1,c2,b}=3'b000; #1 {c1,c2,b}=3'b010; #1 {c1,c2,b}=3'b000;
    #1 {c1,c2,b}=3'b001; #1 {c1,c2,b}=3'b101; #1 {c1,c2,b}=3'b111;
    #1 $stop;
end
initial $monitor("%0d\t%b\t%b\t%b\t%b\t%b", $time,a1,c1,a2,c2,a3);
endmodule

```

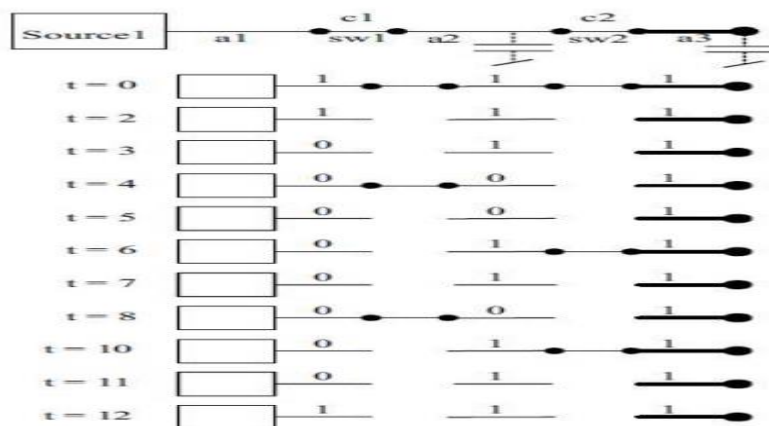


Fig. 172 Changes in signal values at different times

#t	a1	c1	a2	c	a3
#0	1	1	1	1	1
#1	1	0	1	1	1
#2	1	0	1	0	1
#3	0	0	1	0	1
#4	0	1	0	0	1
#5	0	0	0	0	1
#6	0	0	1	1	1
#7	0	0	1	0	1
#8	0	1	0	0	1
#9	0	0	0	0	1
#10	0	0	1	1	1
#11	0	0	1	0	1
#12	1	0	1	0	1
#13	1	1	1	0	1
#14	1	1	1	1	1

Simulation Results

RAM cell :

The figure shows a basic ram cell with facilities for writing data, storing data, and reading data. When switch sw2 is on, qb - the output of inverter g1 - forms the input to the inverter g2 and vice versa. The g1-g2 combination functions as a latch and freezes the last state entry before SW2 turns on. The step-by-step function of the cell is as follows:

- When WSb (write/store) is high, switch SW1 is ON, and switch SW2 OFF. With Sw1 on, input Din is connected to the input of gate g1 and remains so connected.
- When WSb goes low, din is isolated, since SW1 is OFF. But SW2 is ON and the data remains latched in the latch formed by g1-g2. In other words the data Din is stored in the RAM cell formed by g1-g2.
- When RD (Read) goes active (=1), the latched state is available as output Do. Reading is normally done when the latch is in the stored state.

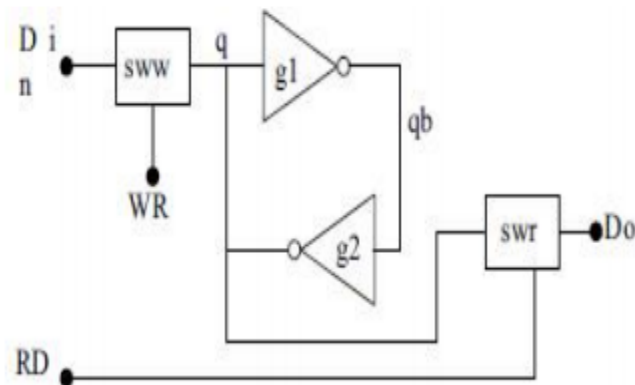


Fig. 167 A Dynamic RAM Cell

```
module ram1(do,din,wr,rd);
output do; input din,wr,rd;
wire qb,q;
tri do;
scw sww(q,din,wr), swr(do,q,rd);
not (pull1,pull0) n1(qb,q), n2(q,qb);
endmodule
```

```
module scw(out,in,n_ctr);
output out; input in,n_ctr;
wire p_ctr;
assign p_ctr = ~n_ctr;
cmos sw(out,in,n_ctr,p_ctr);
endmodule
```

Synthesis

Synthesis converts Verilog (or other HDL) descriptions to an implementation using technology-specific primitives



•Verilog and VHDL started out as simulation languages, but soon programs were written to automatically convert Verilog code in to low-level circuit descriptions (netlists).

- ✓ For FPGAs: LUTs, flip-flops, and RAM blocks–
- ✓ For ASICs: standard cell gate and flip-flop libraries, and memory blocks

Synthesis tool used to

- detect and eliminate redundant logic
- detect combinational feedback loops
- exploit don't-care conditions • detect unused states
- detect and collapse equivalent states , make state assignments
- synthesize optimal, multilevel realizations of logic subject to constraints on area and/or speed physical technology.

Need of Logic Synthesis

1. Automatically manages many details of the design process:

- Fewer bugs
- Improves productivity

2. Abstracts the design data (HDL description) from any particular implementation technology

• Designs can be re-synthesized targeting different chip technologies; E.g.: first implement in FPGA then later in ASIC.

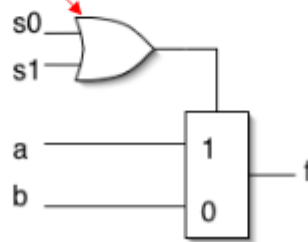
3. In some cases, leads to a more optimal design than could be achieved by manual means (e.g.: logic optimization)

4. If synthesis is not available may lead to less than optimal designs in some cases.

Example :

```
module foo (a,b,s0,s1,f);  
input [3:0] a;  
input [3:0] b;  
input s0,s1;  
output [3:0] f;  
reg f;  
always @ (a or b or s0 or s1)  
    if (!s0 && s1 || s0) f=a; else f=b;  
endmodule
```

- Should expand if-else into 4-bit wide multiplexer (a, b, f are 4-bit vectors) and optimize/minimize the control logic:



Supported Verilog Constructs:

- Net types: wire, tri, supply1, supply0;
- register types: reg, integer, time (64bit reg); arrays of reg
- Continuous assignments
- Gate primitive and module instantiations
- always blocks, user tasks, user functions—inputs, outputs, and in outs to a module
- All operators (+, -, *, /, %, <, >, <=, >=, ==, !=, ==~, !=~, &&, ||, !, ~, &, ~&, |, ~|, ^~, ~^, ^, <<, >>, ?:, { }, {{ } })
- Procedural statements: if-else-if, case, casex, casez, for, repeat, while, forever, begin, end, fork, join
- Procedural assignments: blocking assignments =, non blocking assignments <= (Note: <= cannot be mixed with = for the same register).
- Compiler directives: `define, `ifdef, `else, `endif, `include, `undef:
- Integer ranges and parameter ranges
- Local declarations to begin-end block
- Variable indexing of bit vectors on the left and right sides of assign

Un Supported Verilog Constructs:

Generate error and halt synthesis

- Net types: trireg, wor, trior, wand, triand, tri0, tri1, and charge strength;
- register type: real
- Built-in unidirectional and bidirectional switches, and pull-up, pull-down
- Procedural statements: assign (different from the "continuous assignment"), deassign, wait
- Named events and event triggers
- UDPs (user defined primitives) and specify blocks
- force, release, and hierarchical net names (for simulation only)

Simply ignored

- Delay, delay control, and drive strength
- Scaled, vectored
- Initial block
- Compiler directives (except for `define, `ifdef, `else, `endif, `include, and `undef, which are supported)
- Calls to system tasks and system functions (they are only for simulation)

Synthesis - Combinational Logic

Combination logic function can be expressed as:

$$\text{logic_output}(t) = f(\text{logic_inputs}(t))$$



- ✓ Avoid technology dependent modeling; i.e. implement functionality, not timing.
- ✓ The combinational logic must not have feedback.
- ✓ Specify the output of a combinational behavior for all possible cases of its inputs.
- ✓ Logic that is not combinational will be synthesized as sequential

Combination logic can be generated using

- Netlist of primitives: AND, OR, etc.
- User-defined primitive
- Continuous assignments
- Level-sensitive cyclic behavior
- Procedural continuous assignment (assign ... deassign)

Net list of structured primitives

- ✓ Synthesis tools further optimize a gate netlist specified in terms of Verilog primitives

```
module or_nand_1 (enable, x1, x2, x3, x4, y);
```

```
  input enable, x1, x2, x3, x4;
```

```
  output y;
```

```
  wire w1, w2, w3;
```

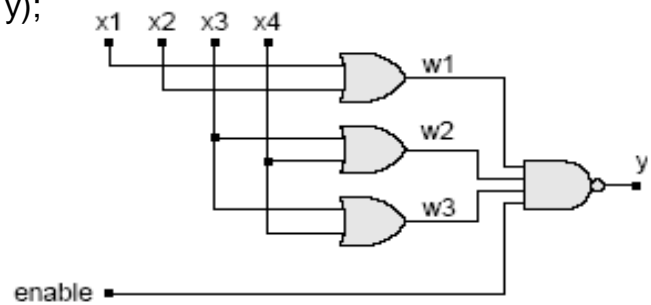
```
  or (w1, x1, x2);
```

```
  or (w2, x3, x4);
```

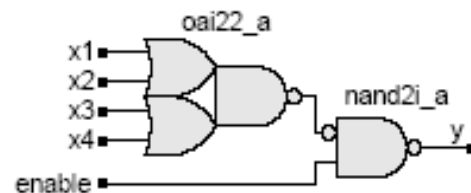
```
  or (w3, x3, x4); // redundant
```

```
  nand (y, w1, w2, w3, enable);
```

```
endmodule
```



Pre-synthesis



Post-synthesis

Synthesis: Continuous Assignment

Continuous assignment statements are synthesizable and they will produce

- (1) combinational logic, (2) latch, (3) three-state output

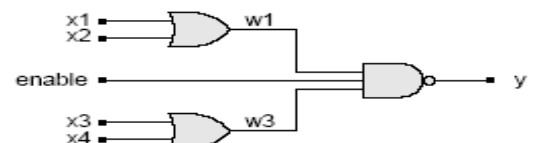
```
module or_nand (y, enable, x1, x2, x3, x4);
```

```
  output y;
```

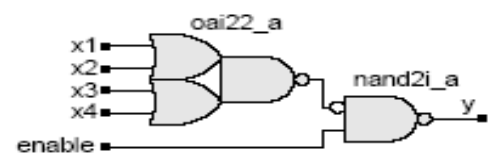
```
  input enable, x1, x2, x3, x4;
```

```
  assign y = ~(enable & (x1 | x2) & (x3 | x4));
```

```
endmodule
```



Pre-synthesis



Post-synthesis

Synthesis: Level-Sensitive Cyclic Behavior

A level-sensitive cyclic behavior will synthesize to combinational logic if it assigns a value to each output for every possible value of its inputs.

- The event control expression of the behavior must be sensitive to every input
- Every path of the activity flow must assign value to every output.

The data words are identical if all of their bits match in each position. Otherwise, the most significant bit at which the words differ determines their relative magnitude.


```

module comparator (a_gt_b, a_lt_b, a_eq_b, a, b); // Alternative algorithm
parameter    size = 2;
output       a_gt_b, a_lt_b, a_eq_b;
input        [size: 1]    a, b;
reg          a_gt_b, a_lt_b, a_eq_b;
integer      k;

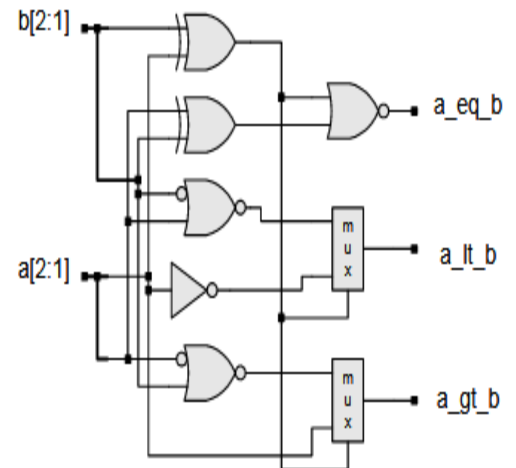
```

```

always @ ( a or b) begin compare_loop
  for (k = size; k > 0; k = k-1) begin
    if (a[k] != b[k]) begin
      a_gt_b = a[k];
      a_lt_b = ~a[k];
      a_eq_b = 0;
      disable compare_loop;
    end // if
  end // for loop
  a_gt_b = 0;
  a_lt_b = 0;
  a_eq_b = 1;
end // compare_loop
endmodule

```

Synthesis Result:



Synthesis of Combinational Logic – Functions

Example:

```

module or_nand_4 (enable, x1, x2, x3, x4, y);
  input enable, x1, x2, x3, x4;
  output y;
  assign y = or_nand(enable, x1, x2, x3, x4);
  function or_nand;
    input enable, x1, x2, x3, x4;
    begin
      or_nand = ~(enable & (x1 | x2) & (x3 | x4));
    end
  endfunction
endmodule

```

Synthesis of Combinational Logic **Tasks :**

Example:

```

module or_nand_5 (enable, x1, x2, x3, x4, y);
  input enable, x1, x2, x3, x4;

```

```

output y;
reg y;
always @ (enable or x1 or x2 or x3 or x4)
  or_nand (enable, x1, x2, x3c, x4);
task or_nand;
  input enable, x1, x2, x3, x4;
  output y;
  begin
    y = !(enable & (x1 | x2) & (x3 | x4));
  end
endtask
endmodule

```

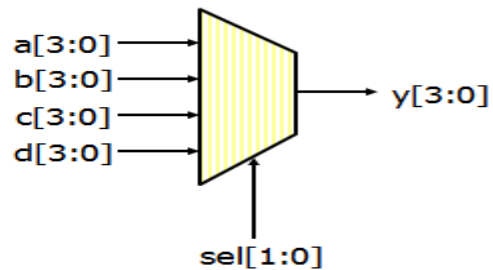
Synthesis of Multiplexors

if .. else Statement

```

module mux_4bits (y, a, b, c, d, sel);
  input [3:0] a, b, c, d;
  input [1:0] sel;
  output [3:0] y;
  reg [3:0] y;
  always @ (a or b or c or d or sel)
    if (sel == 0) y = a; else
    if (sel == 1) y = b; else
    if (sel == 2) y = c; else
    if (sel == 3) y = d;
    else y = 4'bx;
endmodule

```



Unwanted Latches

- ❑ Unintentional latches generally result from incomplete case statement or conditional branch

Example: case statement

```

always @ (sel_a or sel_b or data_a or data_b)

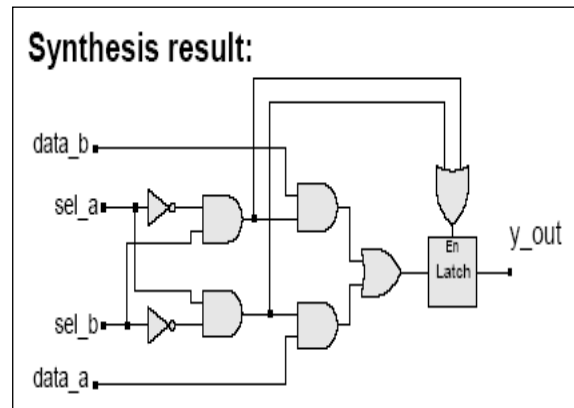
case ({sel_a, sel_b})

  2'b10: y_out = data_a;

  2'b01: y_out = data_b;

endcase

```



The latch is enabled by the "event or" of the cases under which assignment is explicitly made.
 e.g. $(\{sel_a, sel_b\} == 2'b10) \text{ or } (\{sel_a, sel_b\} == 2'b01)$

❑ Example: if .. else statement

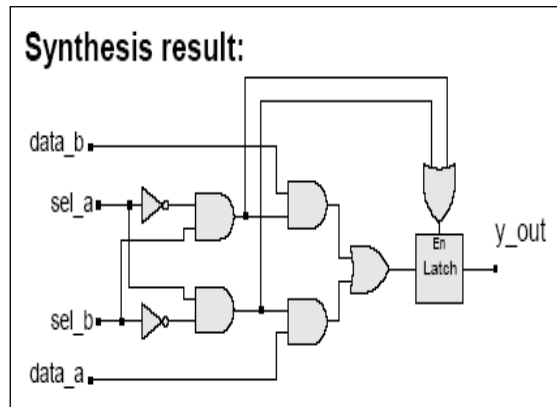
```
always @ (sel_a or sel_b or data_a or data_b)
```

```
if ({sel_a, sel_b} == 2'b10)
```

```
    y_out = data_a;
```

```
else if ({sel_a, sel_b} == 2'b01)
```

```
    y_out = data_b;
```

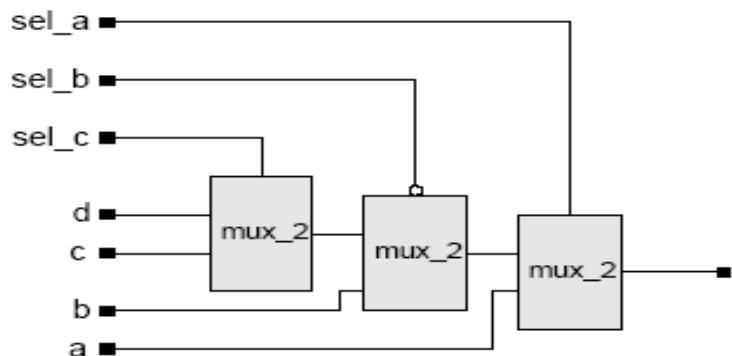


Synthesis of Priority Structures:

- A case statement implicitly attaches higher priority to the first item that it decodes than to the last one
- If the case items are mutually exclusive the synthesis tool will treat them as though they had equal priority and will synthesize a mux rather than a priority structure.
- Even when the list of case items is not mutually exclusive a synthesis tool might allow the user to direct that they be treated without priority (e.g., Synopsys parallel_case directive). This would be useful if only one case item could be selected at a time in actual operation.
- An if statement implies higher priority to the first branch than to the remaining branches.
- If branching is mutually exclusive, synthesis produces a mux structure
- Otherwise create a priority structure

When the branching of a conditional (if) is not mutually exclusive, or when the branches of a case statement are not mutually exclusive, the synthesis tool will create a priority structure.

```
module mux_4pri (y, a, b, c, d, sel_a, sel_b, sel_c);
input a, b, c, d, sel_a, sel_b, sel_c;
output y;
reg y;
always @ (sel_a or sel_b or sel_c or a or b or c or d)
begin
    if (sel_a == 1) y = a; else
    if (sel_b == 0) y = b; else
    if (sel_c == 1) y = c; else
        y = d;
end
endmodule
```



Exploiting Don't-Care Conditions

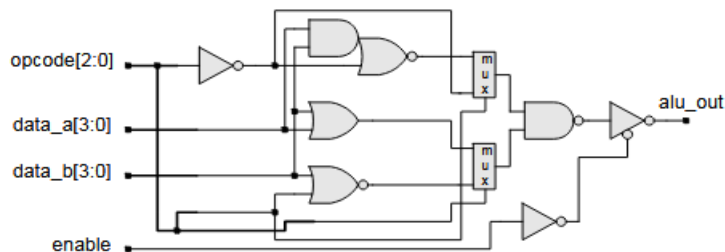
- An assignment to x in a case or an if statement will be treated as a don't care condition in synthesis
- If a conditional operator assigns the value z to the right-hand side expression of a continuous assignment in a level-sensitive behavior, the statement will synthesize to a three-state device driven by combinational logic

```
module alu_with_z1 (alu_out, data_a, data_b, enable, opcode);
input    [2: 0]    opcode;
input    [3: 0]    data_a, data_b;
input    enable;
output   alu_out;      // scalar for illustration
reg      [3: 0]    alu_reg;

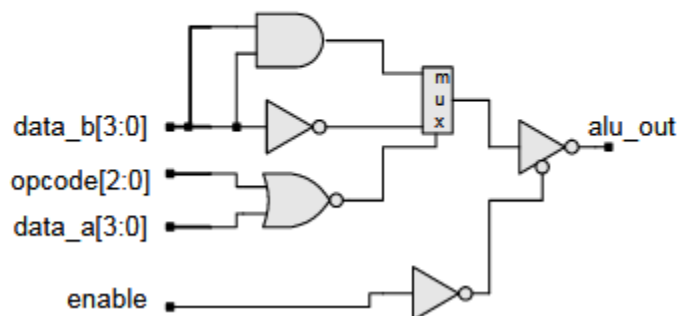
assign alu_out = (enable == 1) ? alu_reg : 4'bz;

always @ (opcode or data_a or data_b)
  case (opcode)
    3'b001:    alu_reg = data_a | data_b;
    3'b010:    alu_reg = data_a ^ data_b;
    3'b110:    alu_reg = ~data_b;
    default:    alu_reg = 4'b0; // alu_with_z2 has default: alu_reg = 4'bx;
  endcase
endmodule
```

Synthesis Result: alu_with_z1



Synthesis Result: alu_with_z2 (Exploit don't-cares)



UDP :USER DEFINED PRIMITIVES

```
primitive boolean_eqs (y, a, b, c);
```

```
  output y;
```

```
  input  a, b, c;
```

```
  table
```

```
    //      Inputs      Output
    //      a    b    c      y
```

```
      0    1    ?    :    1;
```

```
      0    0    ?    :    0;
```

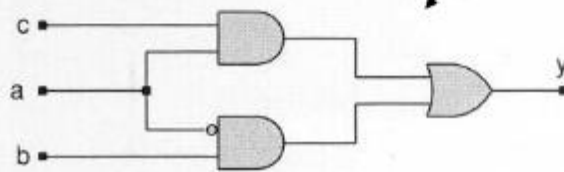
```
      1    ?    1    :    1;
```

```
      1    ?    0    :    0;
```

```
  endtable
```

```
endprimitive
```

Synthesis result



SYNTHESIS OF COMBINATIONAL AND SEQUENTIAL LOGIC USING VERILOG: Synthesis of combinational logic: Net list of structured primitives, a set of continuous assignment statements and level sensitive cyclic behavior with examples, Synthesis of priority structures, Exploiting logic don't care conditions. Synthesis of sequential logic with latches: Accidental synthesis of latches and Intentional synthesis of latches, Synthesis of sequential logic with flip-flops, Synthesis of explicit state machines.

Sequential components: their output values are computed using both the present and past input values. In other words, their outputs depend on the sequence of input values that have occurred over a period of time. This dependence on the past input values requires the presence of memory elements. The values stored in memory elements define the state of a sequential component. Since memory is finite, therefore, the sequence size must always be finite, which means that the sequential logic can contain only a finite number of states. So sequential circuits are sometimes called finite-state machines. Sequential circuits can be a

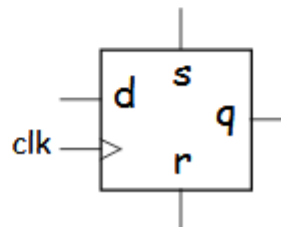
asynchronous or synchronous. Asynchronous sequential circuits change their state and output values whenever a change in input values occurs. Synchronous sequential circuits change their states and output values at fixed points of time, which are specified by the rising or falling edge of a free-running clock signal

- A feedback-free netlist of combinational primitives will synthesize into latch-free combinational logic.
 - A continuous assignment with feedback in a conditional operator will synthesize into a latch.
 - A set of feedback-free continuous assignments will synthesize into latch-free combinational logic.
- **Example: D flip-flop with synchronous set/reset:**

```
module dff(q, d, clk, set, rst);  
  input d, clk, set, rst;  
  output q;  
  reg q;  
  always @(posedge clk)  
    if (reset)  
      q <= 0;  
    else if (set)  
      q <= 1;  
    else  
      q <= d;  
endmodule
```

- "@ (posedge clk)" key to flip-flop generation
- Note in this case, priority logic is appropriate
- For Xilinx Virtex FPGAs, the tool infers a native flip-flop
 - No extra logic needed for set/reset

We prefer synchronous set/reset, but how would you specify asynchronous preset/clear?



FINITE STATE MACHINE :

```
module FSM1(clk,rst, enable, data_in, data_out);
input clk, rst, enable;
input data_in;
output data_out;
```

```
/* Defined state encoding;
this style preferred over 'defines*/
parameter default=2'bxx;
parameter idle=2'b00;
parameter read=2'b01;
parameter write=2'b10;
reg data_out;
reg [1:0] state, next_state;

/* always block for sequential logic*/
always @(posedge clk)
    if (rst) state <= idle;
    else state <= next_state;
```

```
/* always block for CL */
always @(state or enable or data_in)
begin
case (state)
/* For each state def output and next */
idle : begin
        data_out = 1'b0;
        if (enable)
            next_state = read;
        else next_state = idle;
    end
read : begin ... end
write : begin ... end

    default : begin
        next_state = default;
        data_out = 1'bx;
    end
endcase
end
endmodule
```

- Style guidelines (some of these are to get the right result, and some just for readability)
 - Must have reset
 - Use separate always blocks for sequential and combination logic parts
 - Represent states with defined labels or enumerated types
- Use CASE statement in an always to implement next state and output logic
- Always use default case and assert the state variable and output to 'bx':
 - Avoids implied latches
 - Allows use of don't cares leading to simplified logic
- "FSM compiler" within synthesis tool can re-encode your states; Process is controlled by using a synthesis attribute (passed in a comment).
 - Details in Synplify guide

Accidental Synthesis of Latches

A Verilog description of combinational logic must assign value to the outputs for all possible values of the inputs otherwise latches may occur.


```

module or4_behav (y, x_in);
  parameter    word_length = 4;
  output       y;
  input        [word_length - 1: 0] x_in;
  reg          y;
  integer      k;    // Eliminated in synthesis

```

```

always @ x_in
  begin: check_for_1
    y = 0;
    for (k = 0; k <= word_length - 1; k = k+1)
      if (x_in[k] == 1) begin
        y = 1;
        disable check_for_1;
      end
    end
  endmodule

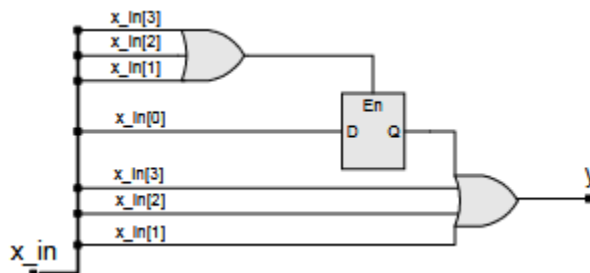
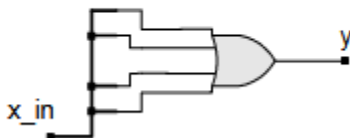
```

```

module or4_behav_latch (y, x_in);
  parameter    word_length = 4;
  output       y;
  input        [word_length - 1: 0] x_in;
  reg          y;
  integer      k;

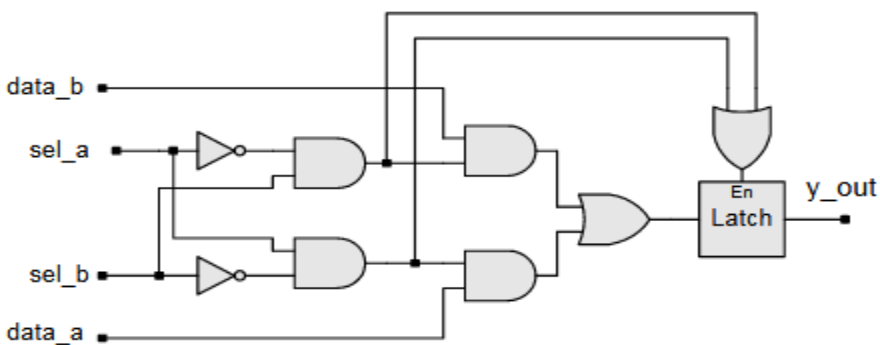
  always @ (x_in[3:1]) // incomplete event control expression
  begin: check_for_1
    y = 0;
    for (k = 0; k <= word_length - 1; k = k+1)
      if (x_in[k] == 1)
        begin
          y = 1;
          disable check_for_1;
        end
    end
  endmodule

```



EXAPMLE 2:

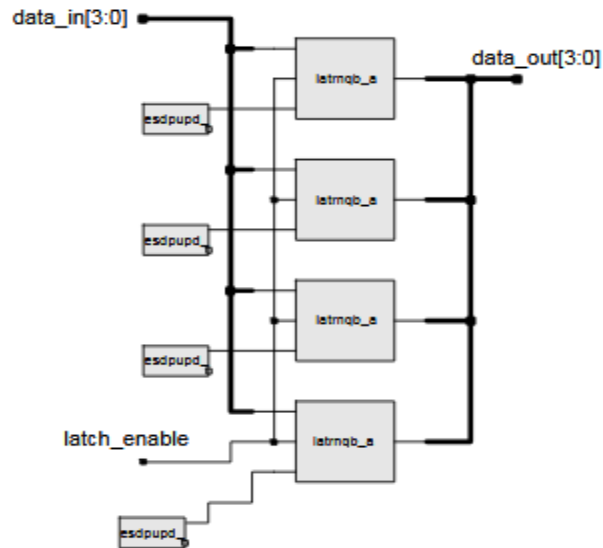
```
module mux_latch (y_out, sel_a, sel_b, data_a, data_b);  
  output      y_out;  
  input       sel_a, sel_b, data_a, data_b;  
  reg        y_out;  
  
  always @ ( sel_a or sel_b or data_a or data_b)  
    case ({sel_a, sel_b})  
      2'b10: y_out = data_a;  
      2'b01: y_out = data_b;  
    endcase  
endmodule
```



Intentional Synthesis of Latches

An if statement in a level-sensitive behavior will synthesize to a latch if the statement assigns value to a register variable in some, but not all, branches, i.e., the statement is incomplete.

```
module latch_if2 (data_out, data_in, latch_enable);  
  output [3:0] data_out;  
  input   [3:0] data_in;  
  input   latch_enable;  
  reg    [3:0] data_out;  
  
  always @ (latch_enable or data_in)  
    if (latch_enable) data_out = data_in; // Incompletely specified  
endmodule
```



Example - 8:3 Priority Encoder

```

module priority (Data, Code, valid_data);
    input  [7:0] Data;
    output [2:0] Code;
    output valid_data;
    reg    [2:0] Code;
    assign valid_data = !Data;           // reduction or

    always @ (Data)
    begin
        if (Data[7]) Code = 7; else
        if (Data[6]) Code = 6; else
        if (Data[5]) Code = 5; else
        if (Data[4]) Code = 4; else
        if (Data[3]) Code = 3; else
        if (Data[2]) Code = 2; else
        if (Data[1]) Code = 1; else
        if (Data[0]) Code = 0; else
            Code = 3'bx;
    end

    always @ (Data)
    case (Data)
        8'b1xxxxxxx : Code = 7;
        8'b01xxxxxx : Code = 6;
        8'b001xxxxx : Code = 5;
        8'b0001xxxx : Code = 4;
        8'b00001xxx : Code = 3;
        8'b000001xx : Code = 2;
        8'b0000001x : Code = 1;
        8'b00000001 : Code = 0;
        default      : Code = 3'bx;
    endcase
endmodule

```

UNIT-VI

VERILOG MODELS: Static RAM Memory, A simplified 486 Bus Model, Interfacing Memory to a Microprocessor Bus, UART Design and Design of Microcontroller CPU.

Static RAM Memory

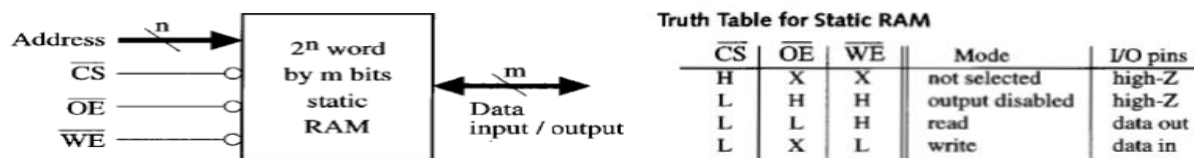


Fig . 1 Block Diagram of Static RAM

RAM stands for random access memory, which means that any word memory can be accessed in the same amount of the time as any other word. Figure 1 shows the block diagram of a static RAM with n address lines, M data lines, and three control lines. This memory can store a total of 2^n words, each m bits wide. The data lines are bi-directional in order to reduce the required number of pins and the package size of the memory chip. When reading from the RAM, the data lines are output; when writing to the RAM, the data lines serve as inputs. The three control lines function as follows: When asserted low, chip select selects the memory chip so that memory read and write operations are possible.

When asserted low, output enable enables the memory output onto an external bus. When asserted low, write enable allows data to be written to the RAM. (We say that a signal is asserted when it is in its active state. An active-low signal is asserted when it is low, and an active-high signal is asserted when it is high.)

The RAM contains address decoders and a memory array. The address inputs to the RAM are decoded to select cells within the RAM. Figure 2 shows the functional equivalent of a static RAM cell that stores one bit of data. The cell contains a transparent D latch, which stores the data. When \overline{CS} is asserted low and \overline{OE} is high, $G = 0$, the cell is in the read mode, and Data Out = Q. When \overline{CS} is asserted low and \overline{WE} is high, $G = 1$ and data can enter the transparent latch. When either \overline{CS} or \overline{WE} goes high, the data is stored in the latch. When \overline{OE} is high, Data Out is high-Z.

6116 static CMOS RAM

6116 static CMOS RAM can store 2K bytes of data. Figure 3 shows the block diagram of a 6116 static RAM, which can store 2048 8-bit words of data. This memory has 16,384 cells, arranged in a 128 x 128 memory matrix. The 11 address lines, which are needed to address the 2¹¹ bytes of data, are divided into two groups. Lines A10 through A4 select one of the 128 rows in the matrix. Lines A3 through A0 select 8 columns in the matrix at a time, since there are 8 data lines. The data outputs from the matrix go through tristate buffers before connecting to the data I/O lines. These buffers are disabled except when reading from the memory.

Block Diagram of 6116 Static RAM

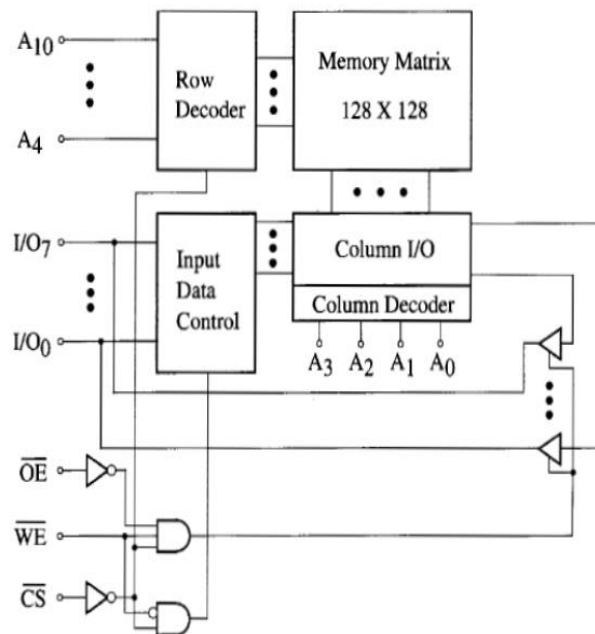


Figure 3 Block diagram of a 6116 static RAM

Truth Table for Static RAM

\overline{CS}	\overline{OE}	\overline{WE}	Mode	I/O pins
H	X	X	not selected	high-Z
L	H	H	output disabled	high-Z
L	L	H	read	data out
L	X	L	write	data in

The truth table for the RAM (given above) describes its basic operation. High-Z in the I/O column means that the output buffers have high-Z outputs, and the data inputs are not used. In the read mode, the address lines are decoded to select eight of the memory cells, and the data comes out on the I/O lines after the memory access time has elapsed. In the write mode, Input data is routed to the latch inputs in the selected memory cells when WE is low, but writing to the latches in the memory cells is not completed until either WE goes high or the chip is deselected. The truth table does not take memory timing into account.

Synchronous Static RAM

Memory is a basic element in any system whether the memory is volatile or non-volatile. In this example, a volatile memory unit is designed in the form of a Synchronous Static RAM. Static Random-Access Memory (SRAM) is a type of semiconductor memory that uses bi-stable latching circuitry to store each bit. The term Static differentiates it from Dynamic RAM (DRAM) which must be periodically refreshed. SRAM retains data, but it is still volatile as data is lost when the power to the memory unit is cut off.

Verilog Module

Figure 1 presents the Verilog module of the Synchronous SRAM. This Synchronous SRAM can store eight 8-bit values. The Synchronous SRAM module consists of a 8-bit data input line, dataIn and a 8-bit data output line, dataOut. The module uses an 8-bit address line, Addr to locate the position of data-byte within the memory array. With an 8-bit address line a 256-unit deep SRAM can be addressed, but in this example, an 8-unit deep SRAM is designed for simplicity. The module is clocked using the 1-bit input clock line Clk. The module also has a 1-bit chip select line, CS.

The 1-bit RD line is used to signal a data read operation on the Synchronous SRAM and the 1-bit WE line is used to signal a data write operation on the Synchronous SRAM. Both the RD and WE lines are active high.

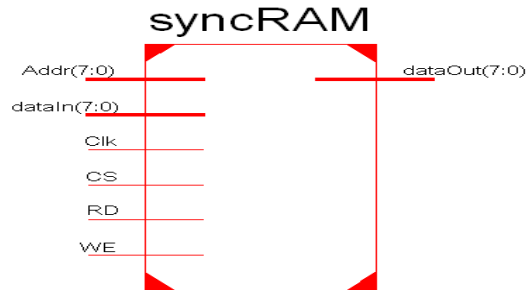


Figure 1. Verilog module of Synchronous SRAM

```

module syncRAM( dataIn, dataOut, Addr, CS, WE, RD, Clk );
// parameters for the width
parameter ADR = 8;
parameter DAT = 8;
parameter DPTH = 8;
//ports
    input  [DAT-1:0] dataIn;
    output reg [DAT-1:0] dataOut;
    input  [ADR-1:0] Addr;
    input  CS, WE, RD, Clk;
//internal variables
    reg [DAT-1:0] SRAM [DPTH-1:0];
always @ (posedge Clk)
    begin
        if (CS == 1'b1)
            begin
                if (WE == 1'b1 && RD == 1'b0)
                    begin
                        SRAM [Addr] = dataIn;
                    end
                else if (RD == 1'b1 && WE == 1'b0)
                    begin
                        dataOut = SRAM [Addr];
                    end
                else;
            end
        else;
    end
endmodule

```

Figure 2. Verilog Code for Synchronous SRAM

Verilog Test Bench for Synchronous SRAM (syncRAM_tb.v)

```
`timescale 1ns / 1ps

module syncRAM_tb;

    // Inputs
        reg [7:0] dataIn;
        reg [7:0] Addr;
        reg CS,WE,RD,Clk;
    // Outputs
        wire [7:0] dataOut;

    // Instantiate the Unit Under Test (UUT)
    syncRAM uut ( .dataIn(dataIn), .dataOut(dataOut),.Addr(Addr), .CS(CS), .WE(WE), .RD(RD), .Clk(Clk) );

    initial
        begin
            // Initialize Inputs
                dataIn = 8'h0; Addr = 8'h0; CS = 1'b0; WE = 1'b0; RD = 1'b0; Clk = 1'b0;

            // Wait 100 ns for global reset to finish
                #100;

            // Add stimulus here

                dataIn = 8'h0; Addr = 8'h0; CS = 1'b1; WE = 1'b1; RD = 1'b0;
                #20; dataIn = 8'h0; Addr = 8'h0;
                #20; dataIn = 8'h1; Addr = 8'h1;
                #20; dataIn = 8'h10; Addr = 8'h2;
                #20; dataIn = 8'h6; Addr = 8'h3;
                #20; dataIn = 8'h12; Addr = 8'h4;
                #40; Addr = 8'h0; WE = 1'b0; RD = 1'b1;
                #20; Addr = 8'h1;
                #20; Addr = 8'h2;
                #20; Addr = 8'h3;
                #20; Addr = 8'h4;

            end

        always #10 Clk = ~Clk;

endmodule
```

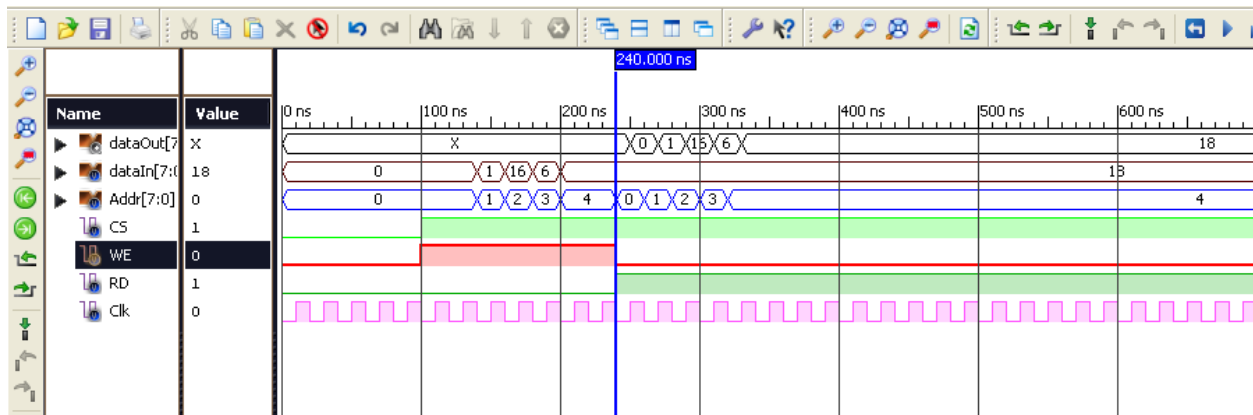



Figure 4. Timing diagram of **Synchronous SRAM** with four data

Microprocessor Bus Interface:

Memories and input-output devices are usually interfaced to microprocessor by means of a tristate bus. To assure proper transfer of data on this bus, the timing characteristics of both the microprocessor bus interface and the memory must be carefully considered. We have already developed a VHDL timing model for a RAM memory, and next we will develop a timing model for microprocessor bus interface. Then we will simulate the operation of a system containing a microprocessor and memory to determine whether the timing specifications are satisfied.

Figure 14 shows a typical bus interface to a memory system.

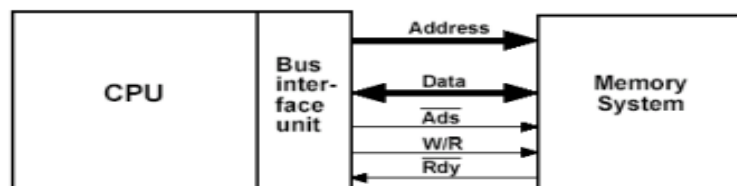


Figure 14 Microprocessor Bus Interface

The normal sequence of events for writing to memory is:

- (1) The microprocessor outputs an address on the address bus and asserts \overline{Ads} (address strobe) to indicate a valid address on the bus;
- (2) the processor places data on the data bus and asserts W/R (write/read) to initiate writing the data to memory. The memory system asserts Rdy (ready) to indicate that the data transfer is complete.

For reading from memory, step (1) is the same, but in step (2) the memory places data on the data bus and these data are stored inside the processor when the memory system asserts Rdy.

In the next section a simplified VHDL model for a 486 microprocessor bus interface developed. The actual 486 bus interface is very complex and supports many different types of bus cycles. Figures 15 and 16 illustrate two of these bus cycles.

Intel 486 Basic 2-2 Bus Cycle

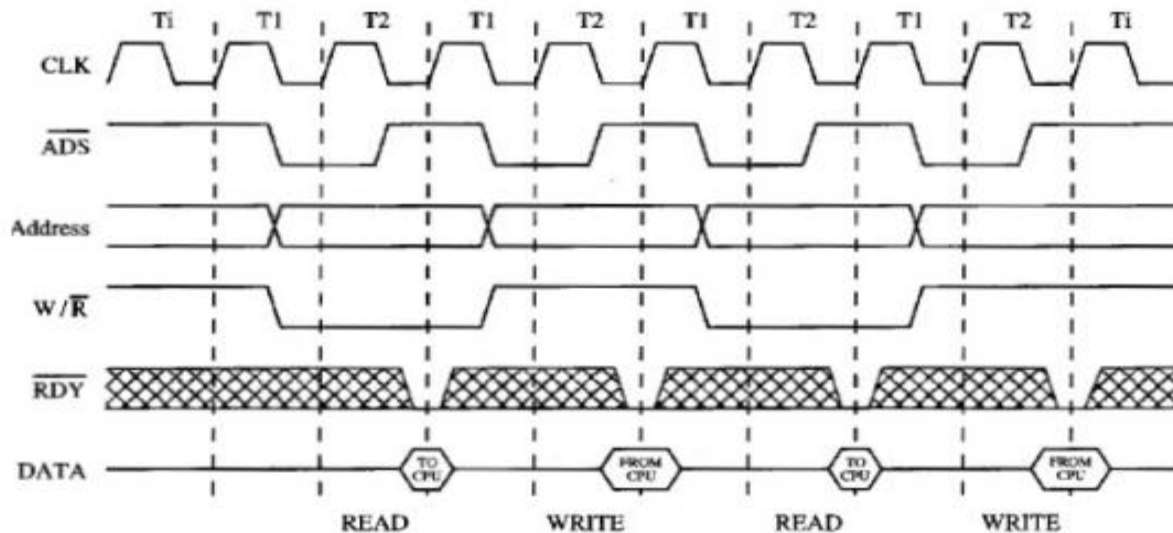


Figure 15

In Figure 15, one word of data is transferred between the CPU and the bus every two clock cycles. These clock cycles are labeled T1 and T2, and they correspond to states of the internal bus controller. In addition, the bus has an idle state, Ti. During Ti and between data transfers on the bus, the data bus is in a high-impedance state (indicated on the diagram by DATA being halfway between 0 and 1). The bus remains in the idle state until the bus interface receives a bus request from the CPU. In T1, the interface outputs a new address on the bus and asserts Ads low. For a read cycle, the read-write signal (W/R) is also asserted low during T1 and T2. During T2 of the read cycle, the memory responds to the new address and places data on the data bus (labeled "to CPU" on the diagram). The memory system also asserts Rdy low to indicate that valid data is on the bus. At the rising edge of the clock that ends T2, the bus interface senses that Rdy is low and the data is stored inside the CPU.

The next bus cycle in Figure15 is a write cycle. As before, the new address is output during T1 and Ads goes low, but W/ R remains high. During T2, the CPU places data on the bus. Near the end of T2, the memory system asserts Rdy low to indicate completion of the write cycle, and the data is stored in the memory at the end of T2 (rising edge of the clock). This is followed by another read and another write cycle.

16 Intel 486 Basic 3-3 Bus Cycle

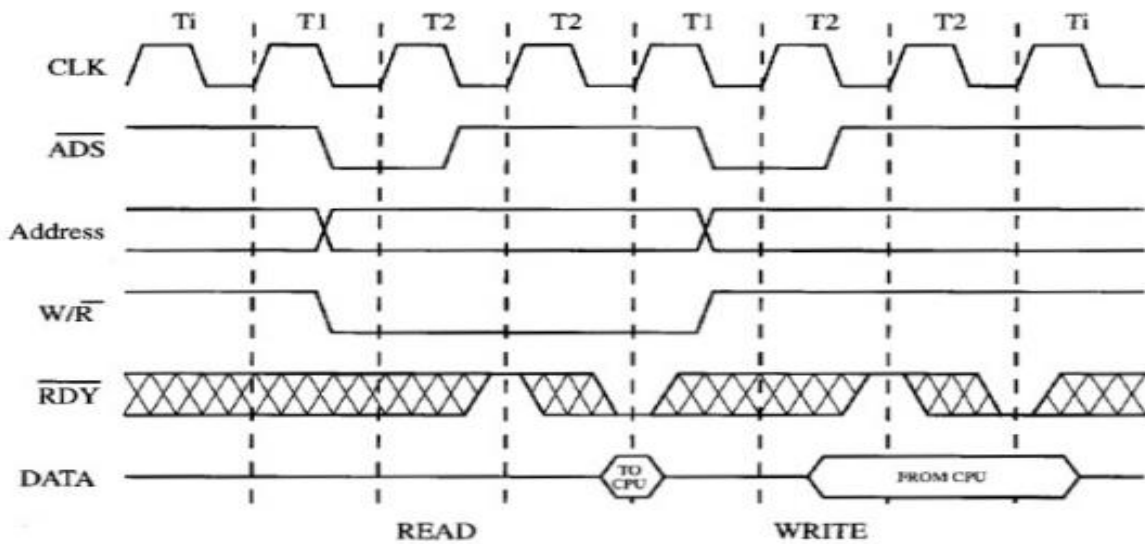


Figure 16

Figure 16 shows 486 read and write bus cycles for the case where the memory is slow and reading one word from memory or writing one word to memory requires three clock cycles. The read operation is similar to that in Figure 5, except at the end of the first T2 cycle, the bus interface senses Rdy is high and inserts another T2 cycle – called wait states.

SIMPLIFIED 486 BUS INTERFACE:

The internal bus interface in Figure 17 shows only those signals needed for transferring data between the bus interface unit and the CPU. If the CPU needs to write data to a memory attached to the external bus interface, it requests a write cycle by setting br (bus request) to 1 and wr = 0. When the write or read cycle is complete, the bus interface unit returns done = 1 to the CPU.

-17 Simplified 486 Bus Interface Unit

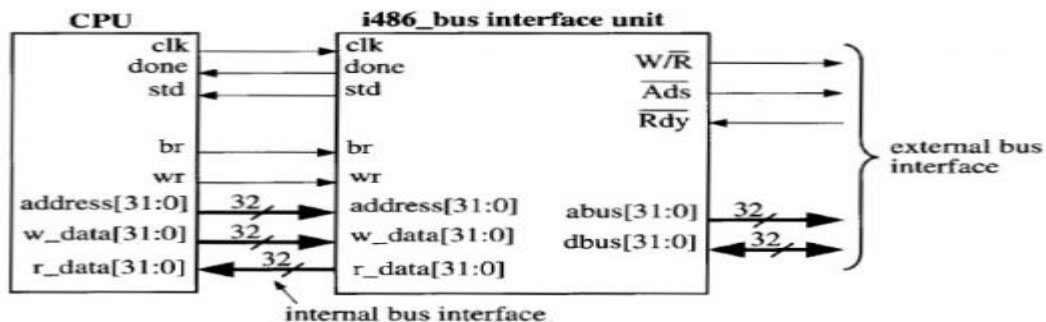


Figure 17

The 486 bus interface unit contains a state machine to control the bus operation. Figure 18 is a modified SM chart that represents a simplified version of this state machine.

- In state T_i , the bus interface is idle, and the data bus is driven to high-Z. When a bus requires (br) is received from the CPU. The controller goes to state T_1 .
- In T_1 , the new address is driven onto the address bus, and Ads is set to 0 to indicate a valid address on the bus. The write-read signal (W/R) is set low for a read cycle or high for a write cycle, and the controller goes to state T_2 .
- In T_2 , Ads returns to 1. For a read cycle, $wr = 0$ and the controller waits for $Rdy = 0$, which indicates valid data is available from the memory, and then std (store data) is asserted to indicate that the data should be stored in the CPU. For a write cycle, $wr = 1$ and data from the CPU is placed on the data bus. The controller then waits for $Rdy = 0$ to indicate that the data has been stored in memory. For both read and write, the done signal is turned on when $Rdy = 0$ to indicate completion of the bus cycle. After read or write is complete, the controller goes to T_i if no bus request is pending, otherwise it goes to state T_1 to initiate another read or write cycle. The done signal remains on in T_i .

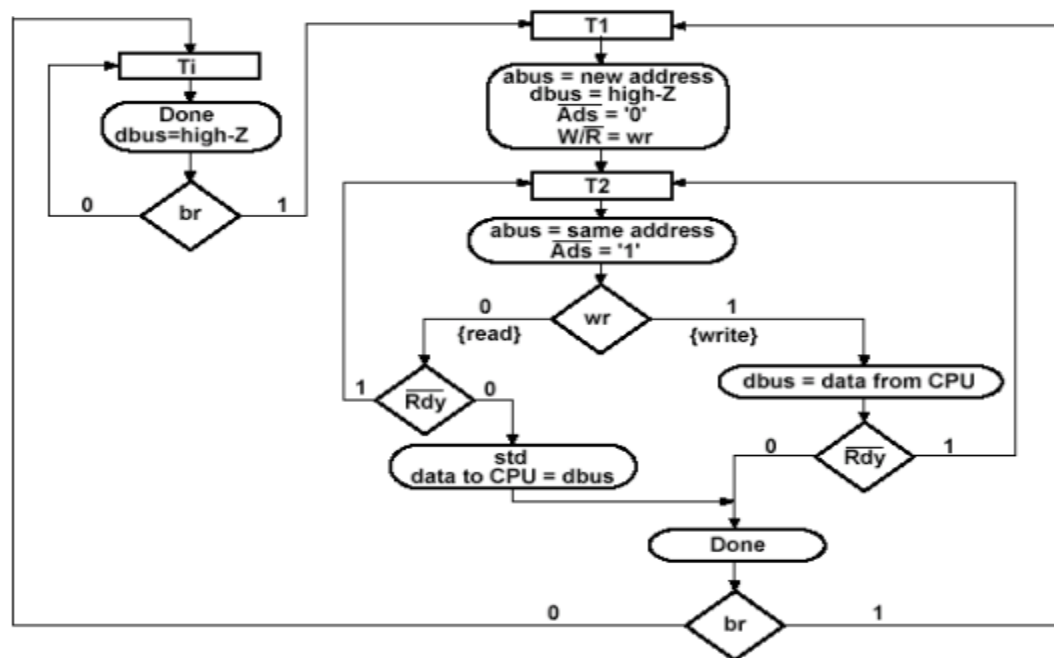


Figure 18 SM Chart for Simplified 486 Bus Interface

UART DESIGN

Universal Asynchronous Receiver Transmitter is an integrated circuit, which is used for transmitting and receiving data asynchronously via the serial port on the computer. It contains a parallel-to-serial converter for data transmitted from the computer and a serial-to-parallel converter for data coming in via the serial line. The UART also has a buffer for temporarily storing data from high-speed transmissions. In addition to the basic job of converting data from parallel to serial for transmission and from serial to parallel on reception. The UART serial module is divided into three sub-modules:

- The baud rate generator,
- receiver module and
- transmitter module.

The baud rate generator is used to produce a local clock signal. In data transmission through the UART, once the baud-rate has been established, both the transmitter and the receiver's internal clock are set to the same frequency.

The UART transmit module converts the data bytes into serial bits according to the frame format and transmits those bits through TXD.

UART frame format consist of a start bit, data bit, parity bit and stop bit. After the StartBit the data bits are sent, with the Least Significant Bit (LSB) sent first. The start bit is always low and the stop bit is always high. When the complete data word has been sent, it adds a parity bit this parity bit may be used by the receiver to perform error checking. Then at least one Stop Bit is sent by the transmitter. Because asynchronous data are “selfsynchronizing”, if there is no data to transmit, the transmission line will be idle

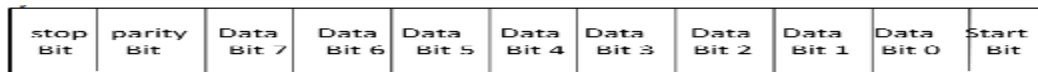


Fig 1: Data format for ASCII text transmitted by a UART

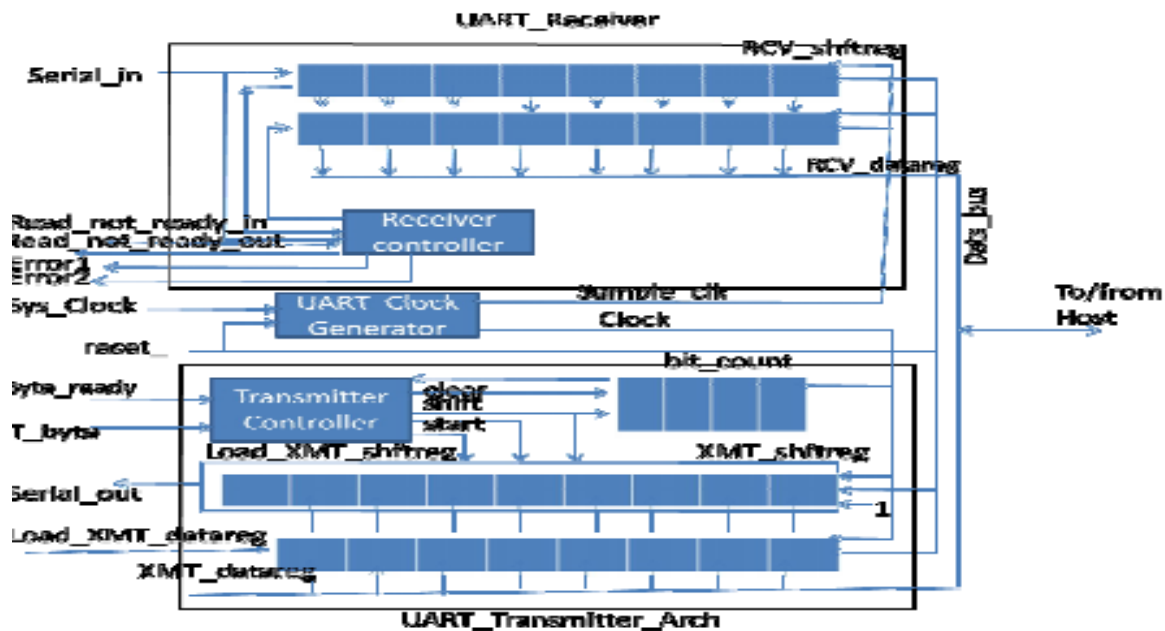


Fig 2: Architecture of UART

UART transmitter

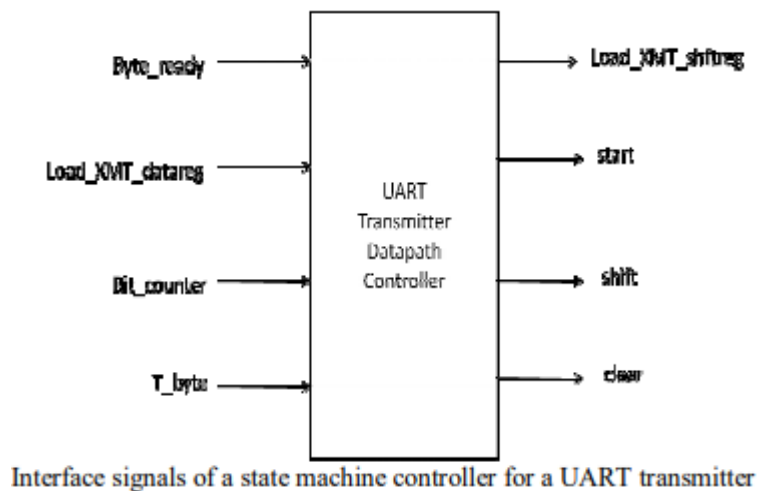
The UART transmitter is always part of larger environment in which a host processor controls transmission by fetching a data word in parallel format and directing the UART to transmit it in a serial format . likewise, receiver must detect transmission, receive the data in serial format, and strip off start- and stop-bits, and store the data word in a parallel format. The receiver's job is more complex because the clock used to send the inbound data .

The input –output signals of the transmitter are shown in the high-level block diagram in figure 3 the input signals are provider by the host processor, and the output signals control the movement of data in the UART .

The architecture of the transmitter will consist of a controller, a data register (XMT-datareg) , a data shift register (XMT-shftreg) , and a status register (bit-count) to count the bits that are transmitted. The status register will be included with in the data path unit.

The ASM chart state machine controlling the transmitter is shown figure 4. The machine as three states : **idle, waiting, and sending.**

When reset-is asserted, the machine a synchronously enters idle, bit-count is cleared ,XMTshftreg is loaded with 1 s ,and the control signal clear, load-XMT- shftreg , shift ,and start are driven to 0. In idle, if an active edge of clock occurs while load-XMT-data-ref=g is asserted by external host the contents of data-bus while to transfer to XMT-data-reg (this action is not part of ASM chart because it occurs independently of the state of the machine) the machine remains in idle until start is asserted



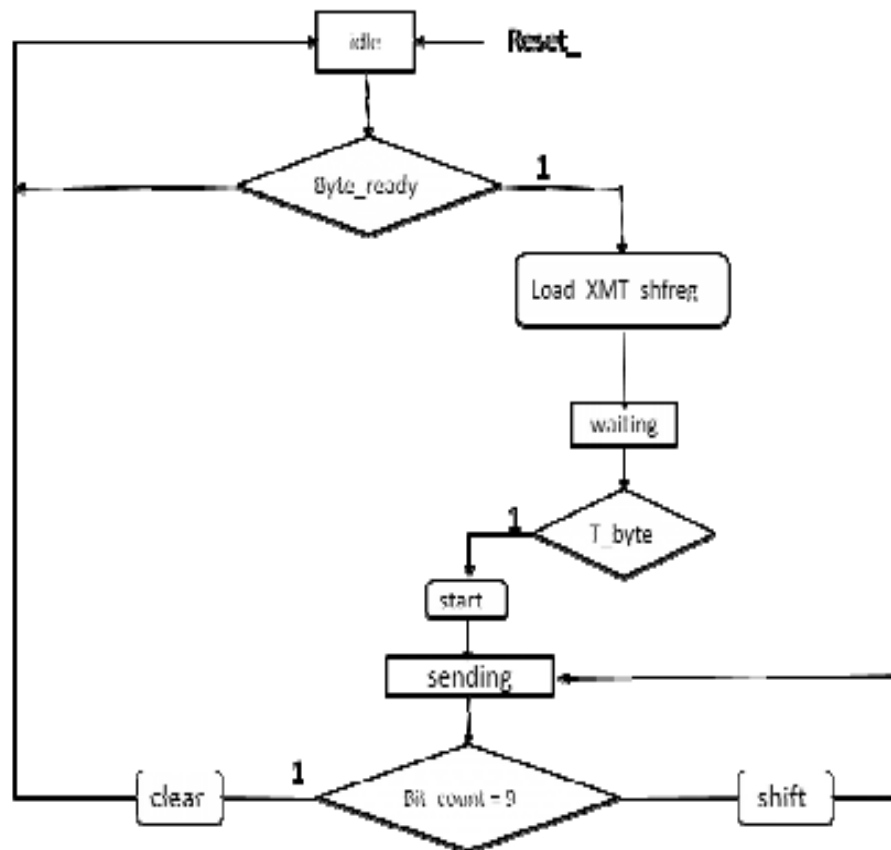


fig 4: ASM chart for the state machine controller for the UART transmitter

When Byte- ready is asserted, Load-XMT-shftreg is asserted and next –state is driven to waiting. The assertion of load-XMT-shftreg indicates that XMT-datareg now contains data that can be transferred to the internal shift register.

At the next active edge of clock, with load-XMT-shftreg asserted, three activities occur:

- (1) State transfer from idle to waiting,
- (2) The contents of XMT-datareg are loaded in to the left mode bits of XMT-shftreg a (word-size+1)-bit shift reg whose LSB signal the start and stop of transmission, and The LSB of XMT- shftreg is reloaded with 1, the stop-bit. The machine remains in waiting until the external processor asserted T-byte.

As the next active edge of clock, with T-byte asserted state enters sending and LSB of XMT-shftreg is set to 0 to signal the start of transmission at the same time shift is driven to 1, and next-state retains the state code corresponding to sending. At sub sequences active edges of clock, with shift asserted state remains in sending and the contents of XMT-shftreg are shifted towards the LSB the machine increments bit-count after each movement of data, and when bit-count reaches 9 clear asserts, indicating that all of bits of augmented word have been shifted to serial output. At the next active edge of the clock, the machine returns to idle.

UART receiver

The UART receiver has the task of receiving the serial bit stream of data, removing the start-bit, and transferring the data in a parallel format to a storage register connected to the host data bus.

The cycles of Sample _ clock will be counted to ensure that the data are sampled in the middle of a bit time, as shown in the figure 6 the sampling algorithm must

- (1) Verify that a start bit has been received,
- (2) Generate samples from 8 bits of the data and
- (3) Load the data on to the local bus.

Three additional samples will be taken to confirm that a valid start –bit has arrived. Thereafter, 8 successive bits will be sampled at approximately the centre of their bit times. Under worst-case conditions of misalignment, the sample is taken a fully cycle of Sample_clock ahead of actual centre of the bit time, which is a tolerable skew.

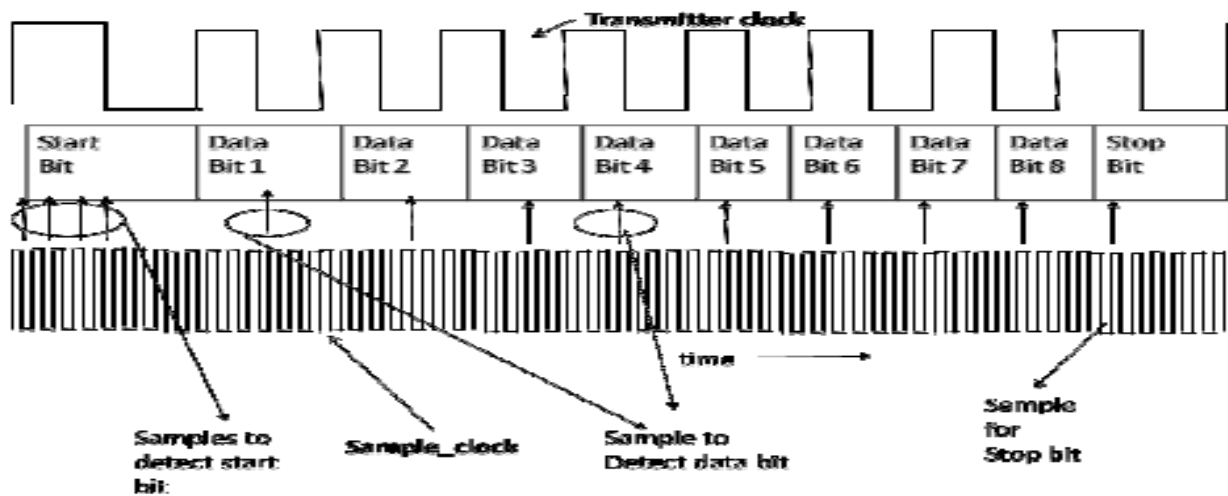


Fig 5: UART receiver sampling format for clock regeneration

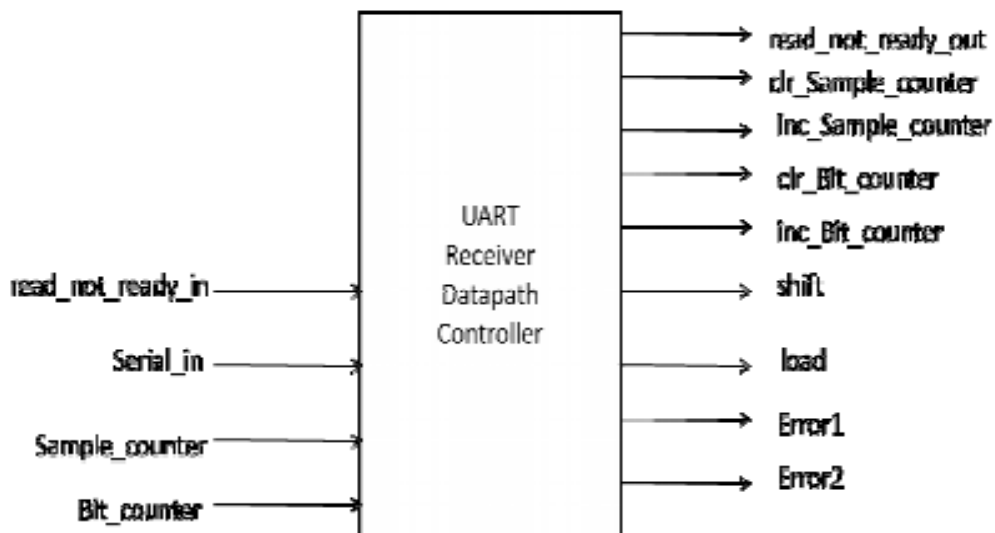


Fig 6: Interface signals of a state-machine controller for the UART receiver

The ASM chart of a state machine controller for the receiver is shown in figure 7 .

The machine has three states: **idle, starting, and receiving.**

- Transitions between states are synchronized by Sample _ clk .
- Assertion of an asynchronous active –low reset puts the machine in the idle state.
- It remains there until Serial _ in is low, and then makes a transition to starting.
- In starting, the machine samples Serial _ in to determine whether the first bit is a valid start – bit (it must be 0).
- Depending on the sampled values , inc _ Sample _ counter and clr _ Sample _ counter may be asserted to increment or clear the counter at the next active edge of Sample _ clock. If the next three samples of Serial _ in are 0, the machine concludes that the start – bit is valid and goes to the state receiving.
- Sample _ counter is cleared on the transition to receiving. In this state, eight successive samples are taken (one for each bit of the byte, at each active edge of Sample _ clk),with inc _ Sample _ counter asserted. Then Bit _ counter is incremented.
- If the sampled bit is not the last (parity) bit , inc _ Bit _ counter and shift are asserted. The assertion of shift will cause the sample value to be loaded into the MSB of RCV _ shftreg, the receiver shift register, and will shift the 7 leftmost bits of the register towards the LSB. After the last bit has been sampled, the machine will assert read _ not _ ready _ out, a handshake output signal to the processor, and clear the bit counter.
- If read _ not _ ready _ in is asserted, the host processor is not ready to receive the data (Error1). If a stop –bit is not the next bit (detected by Serial _ in =0), there is an error in the format of the received data (Error2). Otherwise, load is asserted to cause the contents of the shift register to be transferred as a parallel word to RCV _ datareg, a data register in the host machine, with a direct connection to data _ bus.

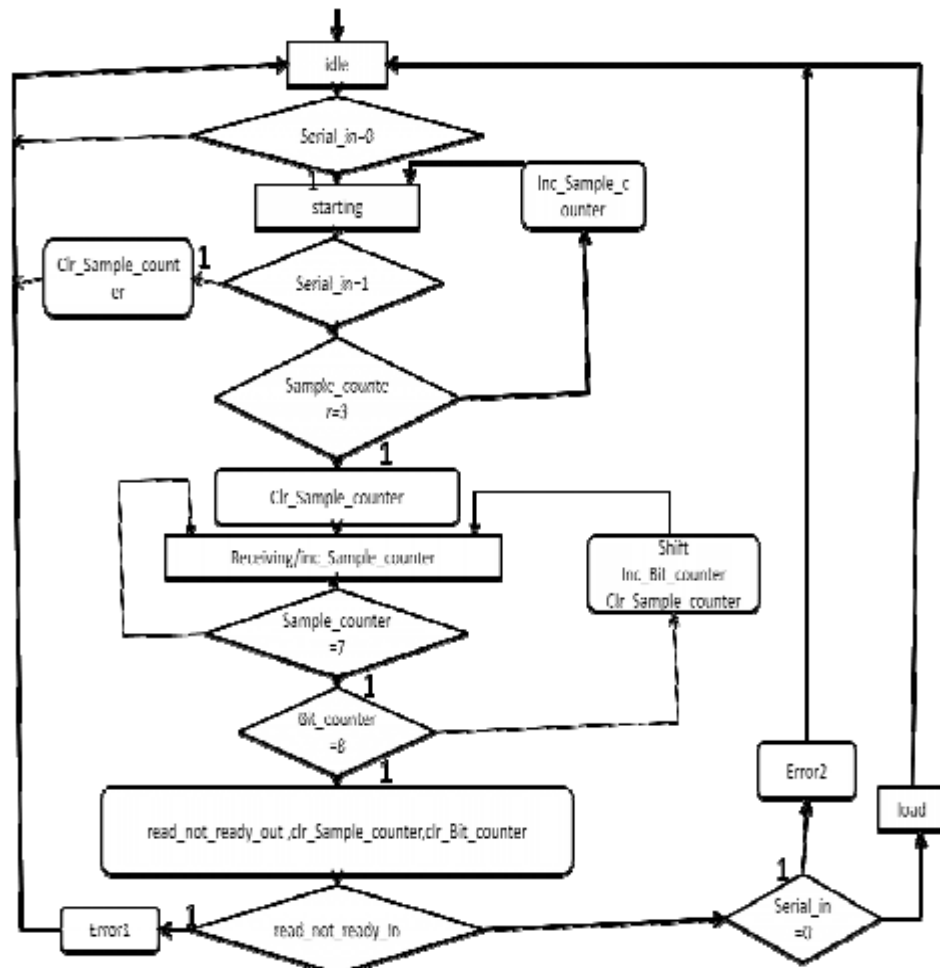


Fig 7: UART Receiver ASM chart

APPLICATIONS:UARTs are used for devices including GPS units

- Modems
- wireless communication
- Bluetooth modules, amongst many other applications
- low-cost home computers or embedded systems dispense with a UART

Design of microcontroller to CPU:

Instruction Set Architecture:

Each instruction is 12 bits. There are 3 types of instructions by encoding, shown as following:

- **M type:** one operand is accumulator (sometimes ignored) and the other operand is from data memory; the result can be stored into accumulator or the data memory entry (same entry as the second operand).
- **I type:** one operand is accumulator and the other operand is immediate number encoded in instruction; the result is stored into accumulator.
- **S type:** special instruction, no operand required. (e.g. NOP)

These instructions can be grouped into 4 categories by function.

1. ALU instruction: using ALU to compute result;
2. Unconditional branch: the GOTO instruction;
3. Conditional branch: the JZ, JC, JS, JO instruction;
4. Special instruction: the NOP.

#	instruction (binary)	instruction (assembly)	instruction (meaning)
0	0000_0000_0000	NOP	(no operation)
1	1011_0000_0001	MOVIA Acc, 1	Acc = 1
2	0010_0010_0000	MOVAM DMem[0], Acc	DMem[0] = Acc = 1
3	1011_0000_0000	MOVIA Acc, 0	Acc = 0
4	0011_0011_0000	MOVMA Acc, DMem[0]	Acc = DMem[0] = 1
5	0001_0000_0101	GOTO 5	(jump to itself, i.e. infinite loop)

The above table contains the detailed information of each M type instruction. Note that “aaaa” encodes the 4 bit address of data memory, and the “d” bit means destination of the result, i.e. if d = 1, result is written to Acc, otherwise the result is written to the same memory location as the operand

Note that all M type instructions are ALU instructions.

I type instructions

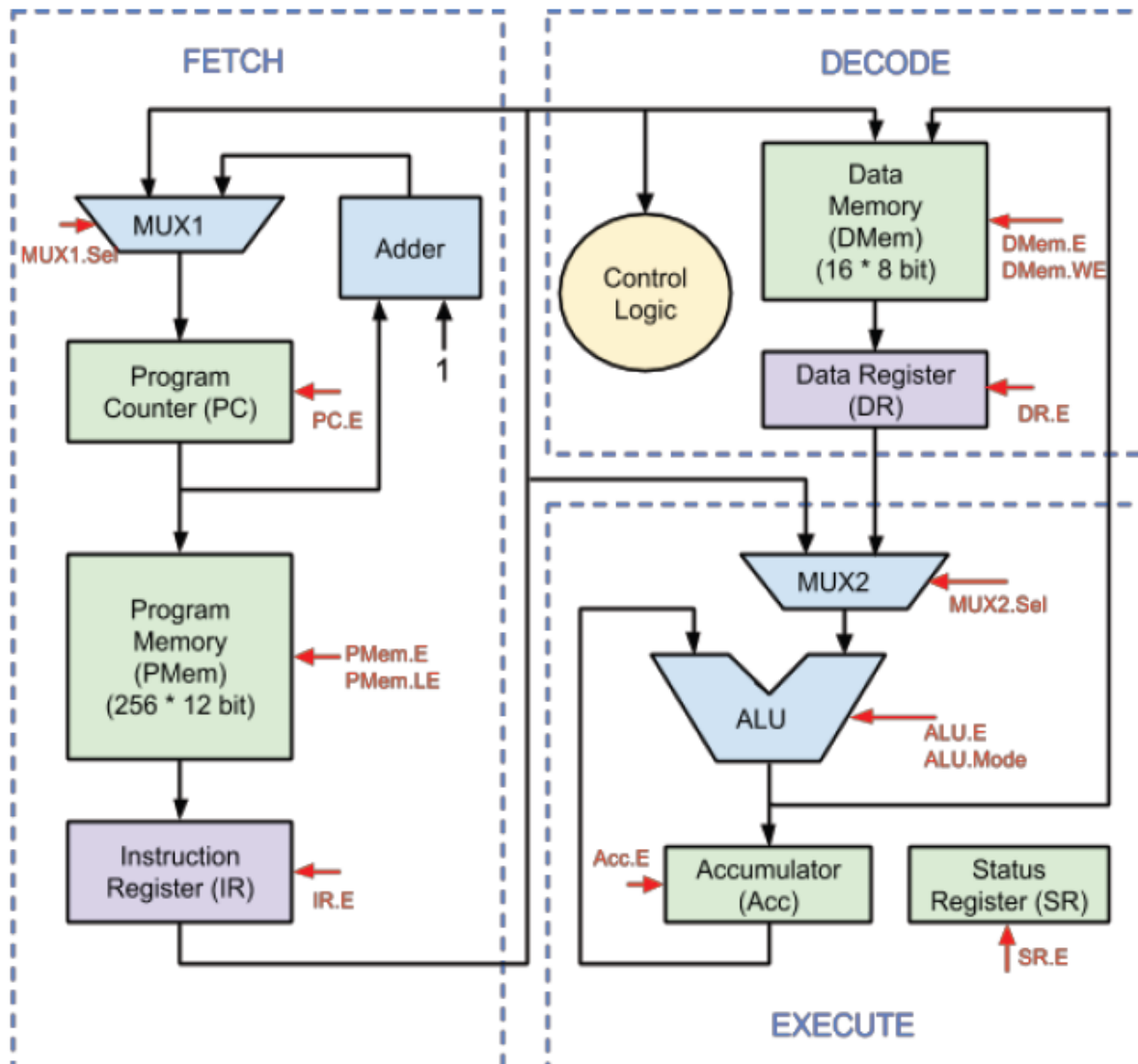
Note that I type instructions contains unconditional branch, conditional branch, and ALU instructions.

S type instructions

There is only one S type instruction, i.e. the NOP instruction.

instruction mnemonics	function	encoding	status affected	example
NOP	no operation	0000_0000_0000	none	NOP

Architecture of Microcontroller unit:



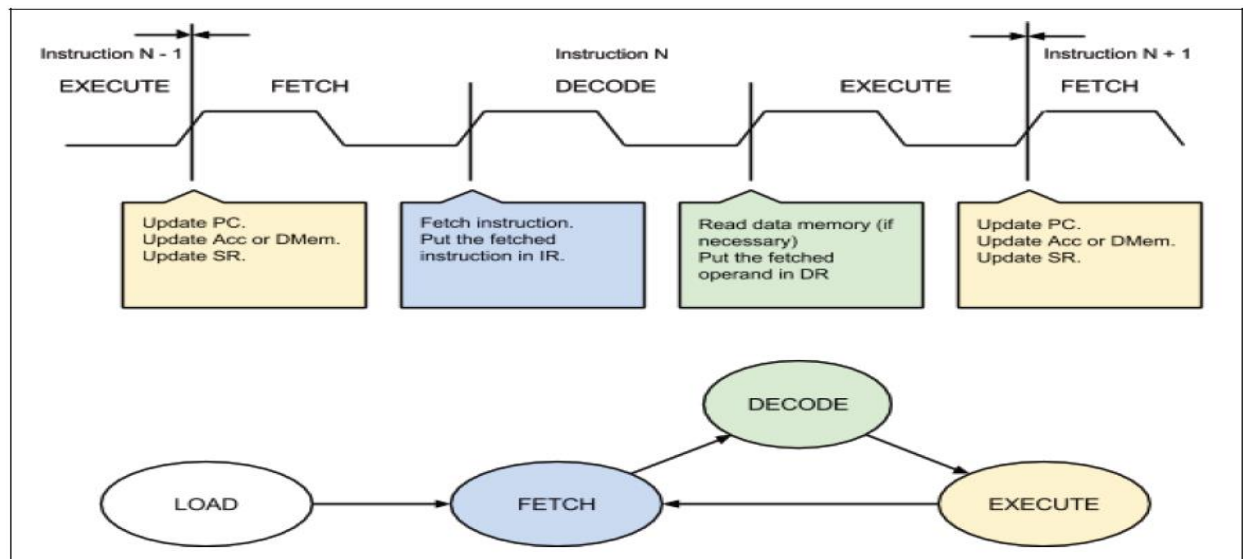
The following two type of components holds programming context.

- Program counter, program memory, data memory, accumulator, status register (green boxes). They are programmer visible registers and memories.
- Instruction register and data register (purple boxes). They are programmer invisible registers.

The following two type of components is Boolean logics that do the actual computation work. They are stateless

ALU, MUX1, MUX2, Adder (blue boxes), used as a functional unit.

- Control Logic (yellow box), used to denote all control signals (red signal)



Each instruction needs 3 clock cycles to finish, i.e. **FETCH stage, DECODE stage, and EXECUTE stage**. Note that it is not pipelined. Together with the initial LOAD state, it can be considered as an FSM of 3 states (technically 4 states).

States:

1. **LOAD (initial state)**: load program to program memory, which takes 1 cycle per instruction loaded;
2. **FETCH (first cycle)**: fetch current instruction from program memory;
3. **DECODE (second cycle)**: decode instruction to generate control logic, read data memory for operand;
4. **EXECUTE (of the third cycle)**: execute instruction

Registers

The microcontroller has 3 programmer visible register:

1. **Program Counter (8 bit, denoted as PC)**: contains the index of current executing instruction.
2. **Accumulator (8 bit, denoted as Acc)**: holds result and 1 operand of the arithmetic or logic calculation.
3. **Status Register (4 bit, denoted as SR)**: holds 4 status bit, i.e. Z, C, S, O.
 1. **Z (zero flag, SR[3])**: 1 if result is zero, 0 otherwise.
 2. **C (carry flag, SR[2])**: 1 if carry is generated, 0 otherwise.
 3. **S (sign flag, SR[1])**: 1 if result is negative (as 2's complement), 0 otherwise.
 4. **O (overflow flag, SR[0])**: 1 if result generates overflow, 0 otherwise.

The microcontroller has 2 programmer invisible registers (i.e. they can not be manipulated by programmer):

1. **Instruction Register (12 bit, denoted as IR)**: contains the current executing instruction.
2. **Data Register (8 bit, denoted as DR)**: contains the operand read from data memory.

Similarly, each of these registers has an enable port as a flag for whether the value of the register should be updated in state transition. They are denoted as IR.E and DR.E.

Program memory

The microcontroller has a 256 entry program memory that stores program instructions, denoted as PMem. Each entry is 12 bits, the i th entry is denoted as PMem[i]. The program memory has the following input/output ports.

- **Enable port** (1 bit, input, denoted as PMem.E): enable the device, i.e. if it is 1, then the entry specified by the address port will be read out, otherwise, nothing is read out.
- **Address port** (8 bit, input, denoted as PMem.Addr): specify which instruction entry is read out, connected to PC.
- **Instruction port** (12 bit, output, denoted as PMem.I): the instruction entry that is read out, connected to IR.
- 3 special ports are used to load program to the memory, not used for executing instructions.
- **Load enable port** (1 bit, input, denoted as PMem.LE): enable the load, i.e. if it is 1, then the entry specified by the address port will be load with the value specified by the load instruction input port and the instruction port is supplied with the same value; otherwise, the entry specified by the address port will be read out on instruction port, and value on instruction load port is ignored.
- **Load address port** (8 bit, input, denoted as PMem.LA): specify which instruction entry is loaded.
- **Load instruction port** (12 bit, input, denoted as PMem.LI): the instruction that is loaded.

For example, if the address point is supplied with 8'b0000_0011 and enable is set to 1, the fourth entry is read out on instruction port.

Data memory

The microcontroller has a 16 entry data memory, denoted as DMem. Each entry is 8 bits, the i th entry is denoted as DMem[i]. The program memory has the following input/output ports.

- **Enable port** (1 bit, input, denoted as DMem.E): enable the device, i.e. if it is 1, then the entry specified by the address port will be read out or written in; otherwise nothing is read out or written in.
- **Write enable port(1 bit, input, denoted as DMem.WE)**: enable the write, i.e. if it is 1, then the entry specified by the address port will be written with the value specified by the data input port and the data output port is supplied with the same value; otherwise, the entry specified by the address port will be read out on data output port, and value on data input port is ignored.
- **Address port (4 bit, input, denoted as DMem.Addr)**: specify which data entry is read out, connected to IR[3:0].
- **Data input port (8 bit, input, denoted as DMem.DI)**: the value that is written in, connected to ALU.Out.

- **Data output port (8 bit, output, denoted as DMem.DO):** the data entry that is read out, connected to MUX2.In1.

For example, if the address port is supplied with 8'0000_0011, data input port is supplied with 8'0000_0000, enable is set to 1, and write enable is set to 1, the fourth entry of the data memory is written with value 0 and the data output port shows 8'0000_0000.

PC adder

PC adder is used to add PC by 1, i.e. move to the next instruction. This component is pure combinational. It has the following port.

- **Adder input port** (8 bit, input, denoted as Adder.In): connected to PC.
- **Adder output port** (8 bit, output, denoted as Adder.Out): connected to MUX1.In2.

MUX1

MUX1 is used to choose the source for updating PC. If the current instruction is not a branch or it is a branch but the branch is not taken, PC is incremented by 1; otherwise PC is set to the jumping target, i.e. IR [7:0]. It has the following port.

- **MUX1 input 1 port** (8 bit, input, denoted as MUX1.In1): connected to IR [7:0].
- **MUX1 input 2 port** (8 bit, input, denoted as MUX1.In2): connected to Adder.Out.
- **MUX1 selection port** (1 bit, input, denoted as MUX1.Sel): connected to control logic.
- **MUX1 output port** (8 bit, output, denoted as MUX1.Out): connected to PC.

ALU

ALU is used to do the actual computation for the current instruction. This component is pure combinational. It has the following port. The mode of ALU is listed in the following table.

- **ALU operand 1 port** (8 bit, input, denoted as ALU.Operand1): connected to Acc.
- **ALU operand 2 port** (8 bit, input, denoted as ALU.Operand2): connected to MUX2.Out.
- **ALU enable port** (1 bit, input, denoted as ALU.E): connected to control logic.
- **ALU mode port** (4 bit, input, denoted as ALU.Mode): connected to control logic.
- **Current flags port** (4 bit, input, denoted as ALU.CFlags): connected to SR.
- **ALU output port** (8 bit, output, denoted as ALU.Out): connected to DMem.DI.
- **ALU flags port** (4 bit, output, denoted as ALU.Flags): the Z (zero), C (carry), S (sign), O (overflow) bits, from MSB to LSB, connected to status register.

MUX2

MUX2 is used to choose the source for operand 2 of ALU. If the current instruction is M type, operand 2 of ALU comes from data memory; if the current instruction is I type, operand 2 of ALU comes from the instruction, i.e. IR [7:0]. It has the following port.

- **MUX2 input 1 port** (8 bit, input, denoted as MUX2.In1): connected to IR [7:0].

- **MUX2 input 2 port** (8 bit, input, denoted as MUX2.In2): connected to DR.
- **MUX2 selection port** (1 bit, input, denoted as MUX2.Sel): connected to control logic.
- **MUX2 output port** (8 bit, output, denoted as MUX2.Out): connected to ALU.Operand2.

Control unit design

Control signal is derived from the current state and current instruction. The control logic component is purely combinational. There are in total 12 control signals, listed as follows.

- **PC.E: enable port of program counter (PC);**
- **Acc.E: enable port of accumulator (Acc);**
- **SR.E: enable port of status register (SR);**
- **IR.E: enable port of instruction register (IR);**
- **DR.E: enable port of data register (DR);**
- **PMem.E: enable port of program memory (PMem);**
- **DMem.E: enable port of data memory (DMem);**
- **DMem.WE: write enable port of data memory (DMem);**
- **ALU.E: enable port of ALU;**
- **ALU.Mode: mode selection port of ALU;**
- **MUX1.Sel: selection port of MUX1;**
- **MUX2.Sel: selection port of MUX2;**

VERILOG CODE FOR ALU:

```
module ALU(  input  [7:0] Operand1,Operand2,
            input  E,
            input  [3:0] Mode,
            input  [3:0] CFlags,
            output  [7:0] Out,
            output  [3:0] Flags
// the Z (zero), C (carry), S (sign),O (overflow) bits,
// from MSB to LSB, connected to status register
);
wire Z,S,O;
reg CarryOut;
reg [7:0] Out_ALU;
always @ (*)
begin
```

```

case (Mode)
4'b0000: {CarryOut, Out_ALU} = Operand1 + Operand2;
4'b0001: begin Out_ALU = Operand1 - Operand2;
CarryOut = !Out_ALU[7];
end
4'b0010: Out_ALU = Operand1;
4'b0011: Out_ALU = Operand2;
4'b0100: Out_ALU = Operand1 & Operand2;
4'b0101: Out_ALU = Operand1 | Operand2;
4'b0110: Out_ALU = Operand1 ^ Operand2;
4'b0111: begin
Out_ALU = Operand2 - Operand1;
CarryOut = !Out_ALU[7];
end
4'b1000: {CarryOut, Out_ALU} = Operand2 + 8'h1;
4'b1001: begin
Out_ALU = Operand2 - 8'h1;
CarryOut = !Out_ALU[7];
end
4'b1010: Out_ALU = (Operand2 << Operand1[2:0]) | (
Operand2 >> Operand1[2:0]);
4'b1011: Out_ALU = (Operand2 >> Operand1[2:0]) | (
Operand2 << Operand1[2:0]);
4'b1100: Out_ALU = Operand2 << Operand1[2:0];
4'b1101: Out_ALU = Operand2 >> Operand1[2:0];
4'b1110: Out_ALU = Operand2 >>> Operand1[2:0];
4'b1111: begin
Out_ALU = 8'h0 - Operand2;
CarryOut = !Out_ALU[7];
end
default: Out_ALU = Operand2;
endcase
end
assign O = Out_ALU[7] ^ Out_ALU[6];
assign Z = (Out_ALU == 0) ? 1'b1 : 1'b0;
assign S = Out_ALU[7];
assign Flags = {Z, CarryOut, S, O};

```

```
assign Out = Out_ALU;  
endmodule
```