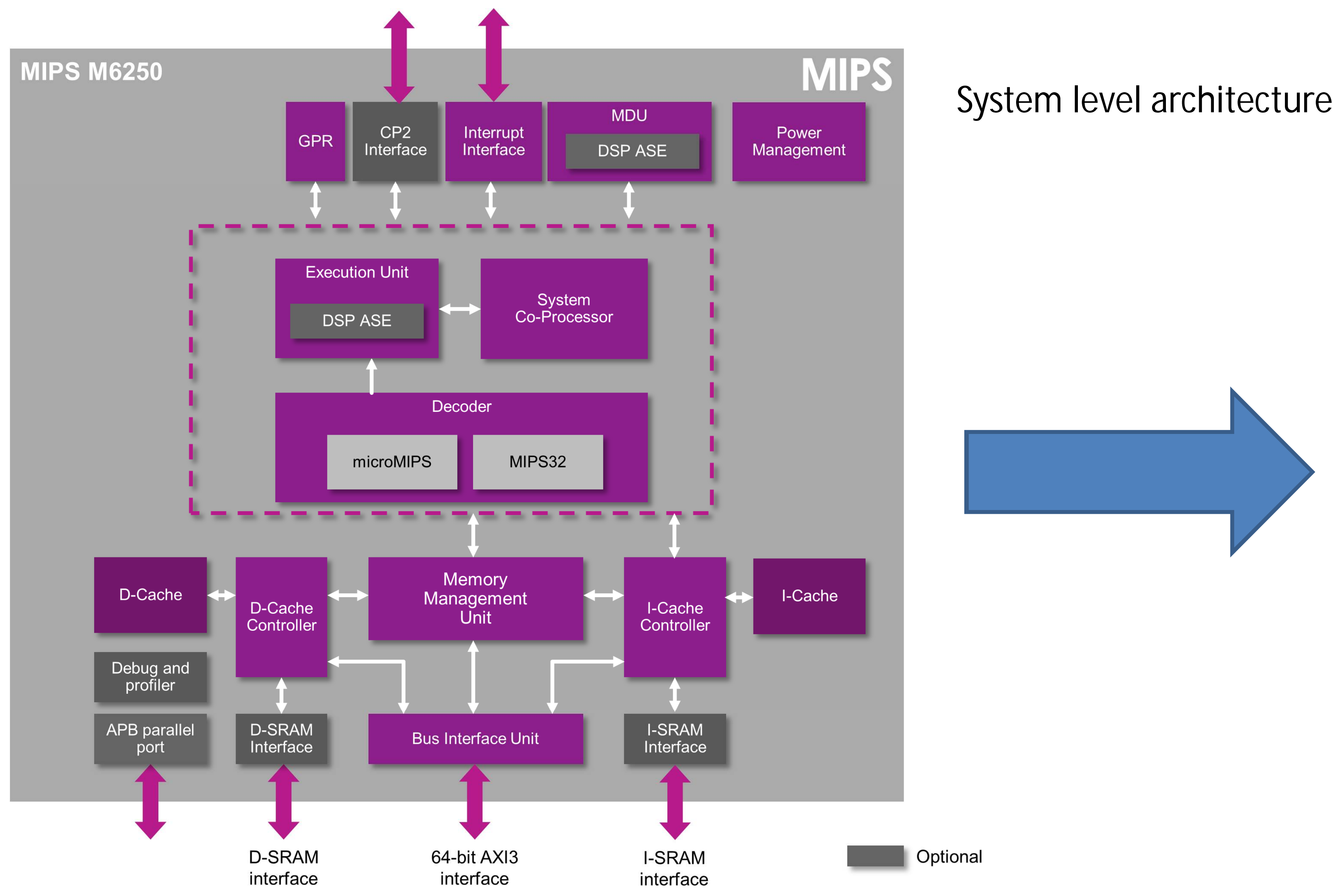
Verilog Programming

Prof. Sayan Chatterjee
Jadavpur University

Introduction

- Verilog is designed for circuit verification and simulation, for timing analysis, for test analysis (testability analysis and fault grading) and for logic synthesis.
- Verilog HDL is a Hardware Description Language that can be used to model a digital system at many levels of abstraction:
  - Algorithmic-level
  - Gate-level
  - Switch-level

Design Flow

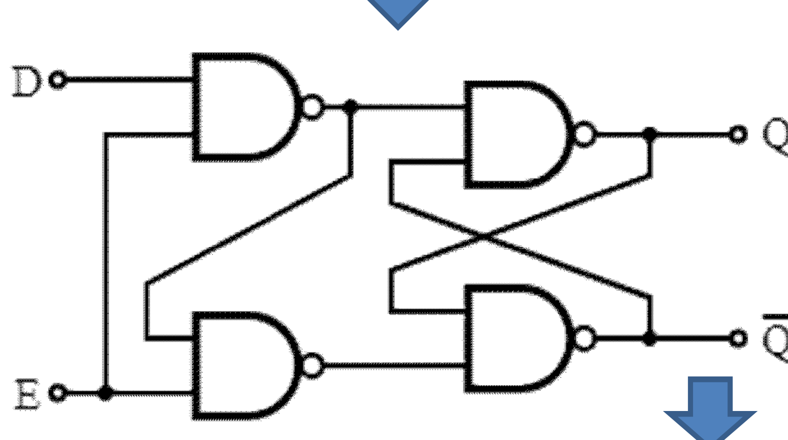


System level architecture

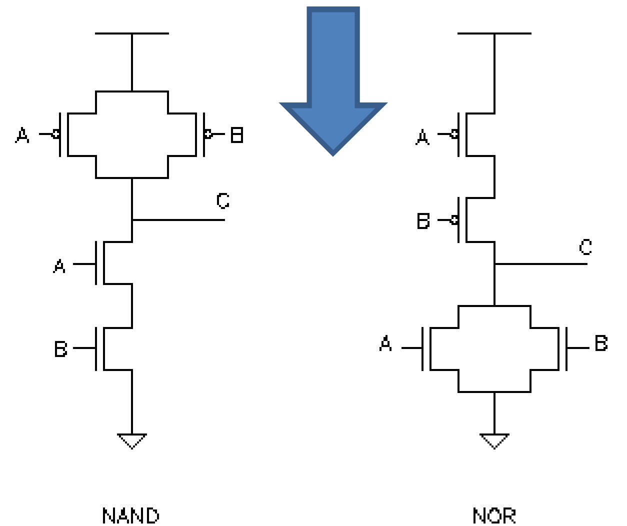
Tools Design Help

```
// Verilog HDL for "archi", "alu32" "behavioral"

`define ALULEN  31
module alu32(Result, ALUOp, A, B, Zero) ;
  output [`ALULEN:0] Result;
  reg    [`ALULEN:0] Result;
  output Zero;
  reg    Zero;
  input  [2:0]  ALUOp;
  input  [`ALULEN:0] A, B;

  always @(A or B or ALUOp)
  begin
        case (ALUOp)
           3'b000: Result = A & B ;//and
           3'b001: Result = A | B ;//or
//        add your code here for addition, subtraction and set on less than

        endcase

//      add your code here for Zero detect
  end
endmodule
```

- Sub blocks are defined by HDL to describe functionality
- Technology library are not considered in this level

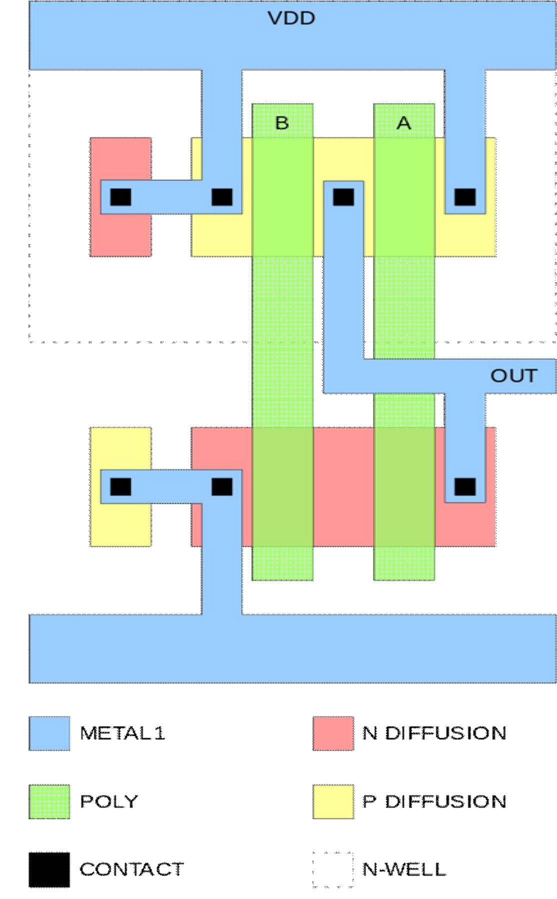


HDLs are then converted to gate level schematics that also involve technology libraries that characterize digital elements like flip-flops. For example, the digital circuit for a D latch contains NAND gates arranged in a certain fashion such that all combinations of D and E inputs produce an output Q given by the truth table.



Implementation of a NAND gate is done by the connection of CMOS transistors in a particular format. At this level, the transistor channel widths, Vdd and the ability to drive the output capacitive load are taken into account during the design process.

The final step is the layout of these transistors in silicon using EDA tools so that it can be fabricated.



History

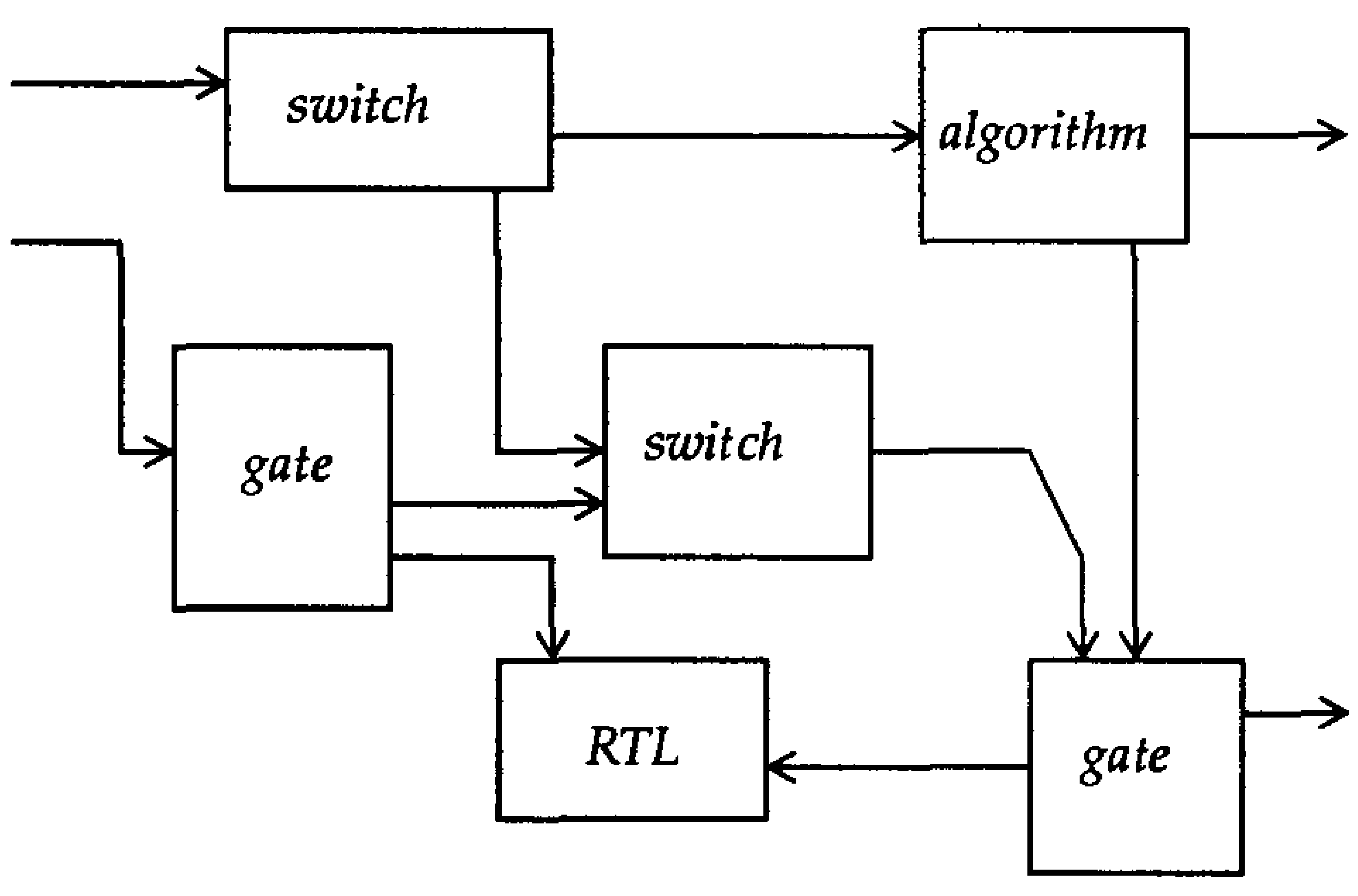
- Verilog HDL was first developed by Gateway Design Automation in 1983 for hardware modeling
- Cadence Design Systems acquired Gateway in 1989, and with it the rights to the language and the simulator. In 1990, Cadence put the language (but not the simulator) into the public domain, with the intention that it should become a standard, non-proprietary language.
- Open verilog International was formed and in 1995 Verilog HDL standardized through IEEE standard.
- The standard is called IEEE Std 1364-1995

- A significantly revised version was published in 2001: IEEE Std. 1364-2001.
- There was a further revision in 2005 but this only added a few minor changes.
- Accellera, which was formed from the merger of Open Verilog International (OVI) and VHDL International, also developed a new standard - systemVerilog, which extends Verilog. SystemVerilog became an IEEE standard (1800-2005) in 2005.

Salient points

- Primitive logic gates are built in the language
- User defined primitive where primitive could either be combinational or sequential logic.
- Switch level modeling primitive gate are also built in
- Language can provide pin to pin delay, path delay and timing check
- Hierarchical design can be described up to any level

Mixed level Modeling



Primitives

- Verilog provides a standard set of primitives, such as **and**, **nand**, **or**, **nor**, and **not**, as a part of the language. These are also commonly known as *built-in* primitives.
- 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.

Examples (Primitives)

- Multiple-input gates:
  - **and, nand, or, nor, xor, xnor**

    xor xor1(out, inA, inB, inC);

Multiple-output gates:

buf, not

Not inverter1(fanout1, fanout2, in);

SYMBOLS:	in —▷— out buf	in —▷o— out not
TRUTH TABLES:	buf in | out 0 | 0 1 | 1 X | X Z | X	not in | out 0 | 1 1 | 0 X | X Z | X
INSTANTIATIONS:	**buf buf1(out,in);**	**not not1(out,in);**

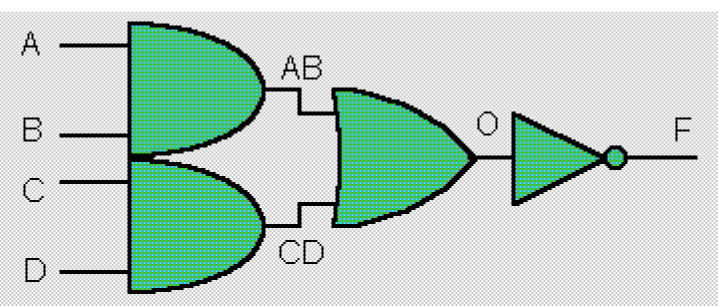
SYMBOLS:	in — bufif1 — out, ctrl bufif1	in — notif1 — out, ctrl notif1
	in — bufif0 — out, ctrl bufif0	in — notif0 — out, ctrl notif0
INSTANTIATIONS:	**bufif1 buf1(out,in,ctrl);** **bufif0 buf0(out,in,ctrl);**	

Modules

Module *module_name* (*port_list*);

<u>Declarations</u>:

input, output, wire, parameter…..



<u>System Modeling</u>:

describe the system in gate-level, data-flow, or behavioral style…

endmodule

- **Used to construct <u>design hierarchy</u>**
- **Cannot be <u>nested</u>**

Input Output

- Input Declaration
  - Scalar
    - input *list of input identifiers*;
    - Example: input A, B, c_in;
  - Vector
    - input[*range*] *list of input identifiers*;
    - Example: input[15:0] A, B, data;
- Output Declaration
  - Scalar Example: output c_out, OV, MINUS;
  - Vector Example: output[7:0] ACC, REG_IN, data_out;

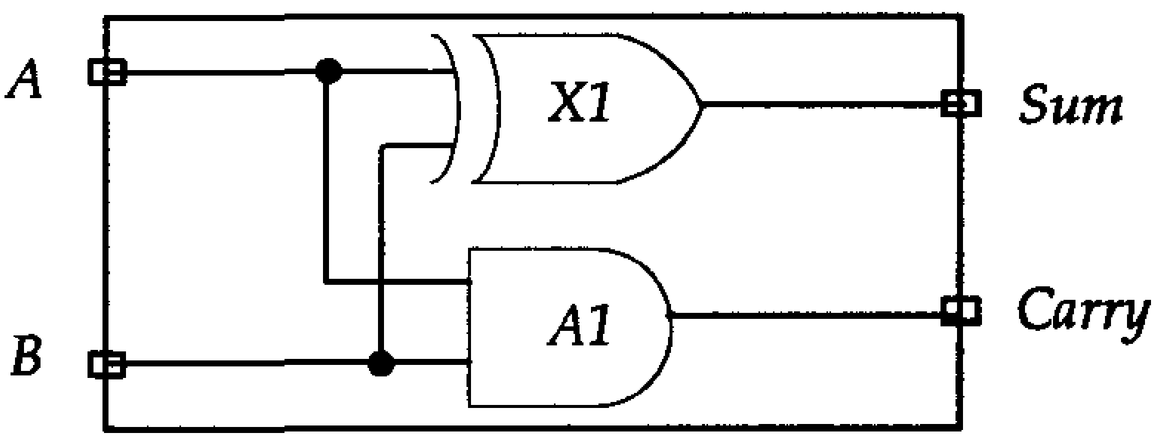
```
// Compute the logical AND and OR of
   inputs A and B.

   module AND_OR(andOut, orOut, A, B);
       output andOut, orOut;
        input A, B;
       and TheAndGate (andOut, A, B);
       or TheOrGate (orOut, A, B);
   endmodule
```

Operation/ primitives

User defined name

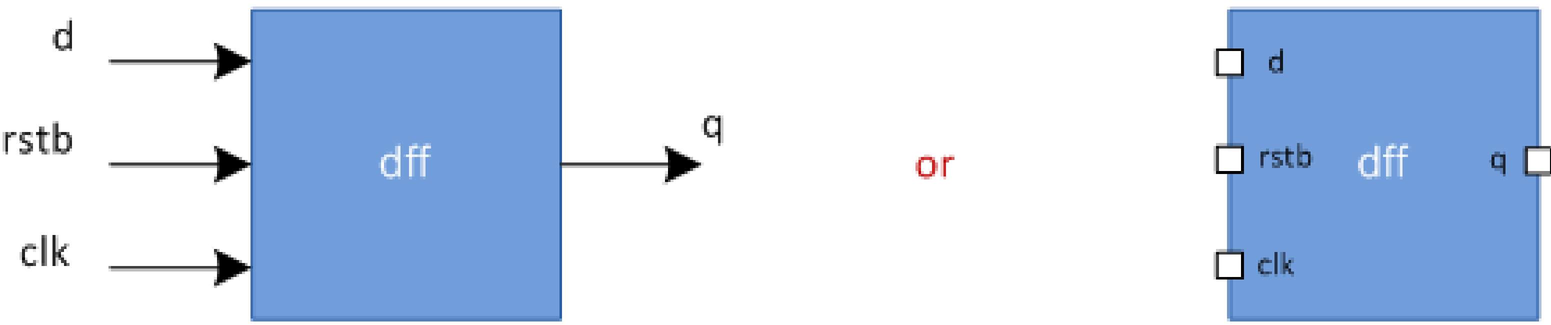
AND_OR
A
B
andOut
orOut
TheAndGate
TheOrGate

Half Adder



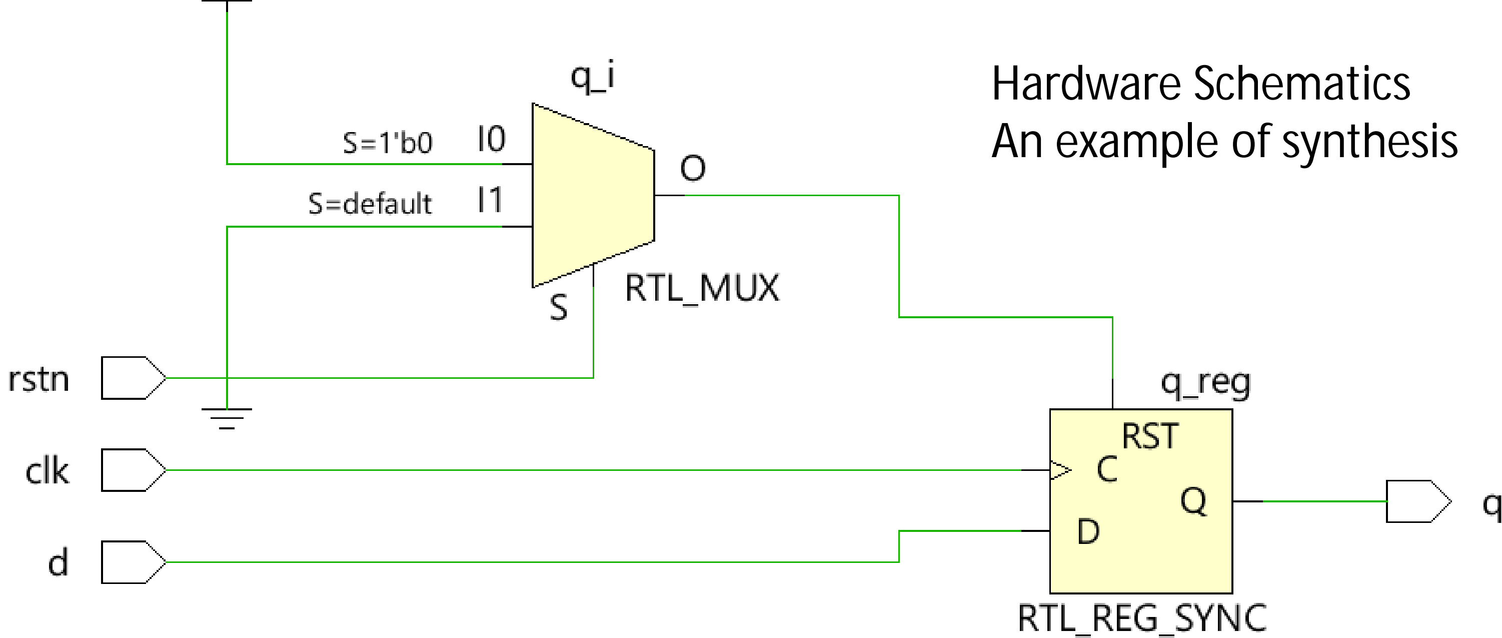

```
module HalfAdder (A, B, Sum, Carry);
  input A, B;
  output Sum, Carry;

  assign #2 Sum = A ^ B;
  assign #5 Carry = A & B;
endmodule
```

```
module <name> ([port_list]);
    // Contents of the module
endmodule

// A module can have an empty portlist
module name;
    // Contents of the module
endmodule
```



The module `dff` represents a D flip flop which has three input ports `d`, `clk`, `rstn` and one output port `q`. Contents of the module describe how a D flip flop should behave for different combinations of inputs. Here, input `d` is always assigned to output `q` at positive edge of clock if `rstn` is high because it is an active low reset.

```verilog
// Module called "dff" has 3 inputs and 1 output port
module dff (    input           d,
                            input           clk,
                            input           rstn,
                            output reg  q);

    // Contents of the module
    always @ (posedge clk) begin
        if (!rstn)
            q <= 0;
        else
            q <= d;
    end
endmodule
```



Constants

- **Constants:**
  - The format is: `W'Bval`
  - Examples:
    - `1'b0` – single bit binary 0 (or decimal 0)
    - `4'b0011` - 4 bit binary 0011 (or decimal 3)
    - `8'hff` = 8 bit hexadecimal ff (or decimal 255)
    - `8'd255`= 8 bit decimal 255

Output message

System Tasks

- System tasks are used to provide interface to simulation data
- Identified by a `$name` syntax
- Printing tasks:
  - `$display`, `$strobe`: Print once the statement is executed
  - `$monitor`: Print every time there is a change in one of the parameters
  - All take the "c" style `printf` format

```
$display("At %t Value of out is %b\n",$time,out);
```

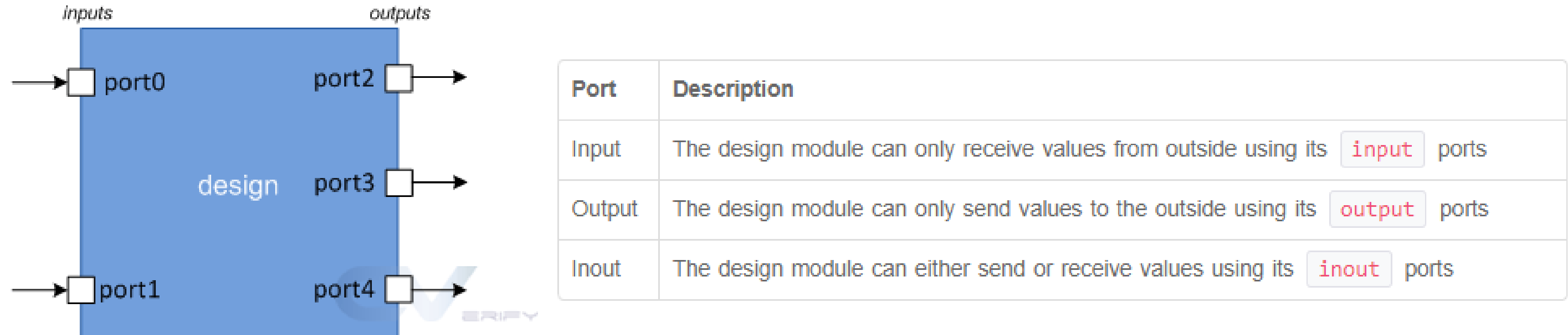
Operators

Operator Type	Symbol	Operation Performed
Arithmetic	+	Add
	−	Subtract
	*	Multiply
	/	Divide
	%	Modulus
Logical	!	Logical negation
	&&	Logical and
	||	Logical or

Operator Type	Symbol	Operation Performed
Relational	>	Greater than
	<	Less than
	>=	Greater than or equal
	<=	Less than or equal
Equality	==	Equality
	!=	Inequality
Bitwise	~	Bitwise negation
	&	Bitwise AND
	|	Bitwise OR
	^	Bitwise XOR

Operator Type	Symbol	Operation Performed
Shift	<<	Shift left logical
	>>	Shift right logical
	<<<	Arithmetic left shift
	>>>	Arithmetic left shift
Concatenation	{}	Join bits
Replication	{{}}	Duplicate bits
Indexing/Slicing	`[MSB:LSB]`	Select bits

Ports



Port	Description
Input	The design module can only receive values from outside using its `input` ports
Output	The design module can only send values to the outside using its `output` ports
Inout	The design module can either send or receive values using its `inout` ports

Ports are by default considered as nets of type `wire` .

```
1  input  [net_type] [range] list_of_names;    // Input port
2  inout  [net_type] [range] list_of_names;    // Input & Output port
3  output [net_type] [range] list_of_names;    // Output port driven by a wir
4  output [var_type] [range] list_of_names;    // Output port driven by a var
```

```
module  my_design ( input wire          clk,
                    input                    en,
                    input                    rw,
                    inout [15:0]      data,
                    output                   int );

    // Design behavior as Verilog code

endmodule
```

In, out and inout in a single module declaration

Signed ports

The `signed` attribute can be attached to a port declaration or a net/reg declaration or both. Implicit nets are by default **unsigned**.

```
module ( input      a,
                    b,
         output     c);

    // ports a, b, and c are by default unsigned
endmodule
```

If either the net/reg declaration has a `signed` attribute, then the other shall also be considered signed.

```
1  module ( input signed a, b,
2          output c);
3     wire a, b;          // a, b are signed from port declaration
4     reg signed c;       // c is signed from reg declaration
5  endmodule
```

Verilog reg, Verilog wire

Verilog data types are divided into two main groups: **nets and variables.** The distinction comes from how they are intended to represent different hardware structures.

A net data type represents a **physical connection** between structural entities (think a plain wire), such as between gates or between modules. **It does not store any value.** Its value is derived from what is being driven from its driver(s). Verilog wire is probably the most common net data type, although there are many other net data types such as tri, wand, supply0.

A variable data type generally represents a **piece of storage**. It holds a value assigned to it until the next assignment. Verilog **reg** is probably the most common variable data type. Verilog **reg** is generally used to model hardware registers (although it can also represent combinatorial logic, like inside an **always@(*)** block). Other variable data types include **integer**, **time**, **real**, **realtime**.

Almost all Verilog data types are 4-state, which means they can take on 4 values:

- 0 represents a logic zero, or a false condition
- 1 represents a logic one, or a true condition
- X represents an unknown logic value
- Z represents a high-impedance state

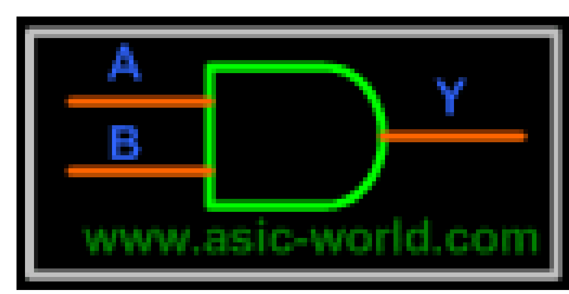
Verilog net data types can only be assigned values by continuous assignments. This means using constructs like continuous assignment statement (**assign statement**), or drive it from an output port.

Verilog **variable data** types can only be assigned values using procedural assignments. This means inside an **always block**, an initial block, a task, a function. The assignment occurs on some kind of trigger (like the posedge of a clock), after which the variable retains its value until the next assignment (at the next trigger). This makes variables ideal for modeling storage elements like flip-flops.

Wire vs Reg

wire data types can be used for connecting the output port to the actual driver. Below is the code which when synthesized gives a AND gate as output, as we know a AND gate can drive a load.

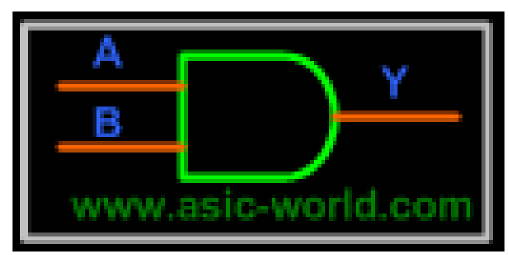
```
1 module wire_example( a, b, y);
2   input a, b;
3  output y;
4   wire a, b, y;
5   assign y = a & b;
6  endmodule
```



What this implies is that wire is used for designing combinational logic, as we all know that this kind of logic can not store a value. As you can see from the example above, a wire can be assigned a value by an assign statement. Default data type is wire: this means that if you declare a variable without specifying reg or wire, it will be a 1-bit wide wire.

Now, coming to reg data type, reg can store value and drive strength. Something that we need to know about reg is that it can be used for modeling both combinational and sequential logic. Reg data type can be driven from initial and always block.

Reg data type as Combinational element




```
 1 module reg_combo_example( a, b, y);
 2 input a, b;
 3 output y;
 4
 5 reg   y;
 6 wire a, b;
 7
 8 always @ ( a or b)
 9 begin
10   y = a & b;
11 end
12
13 endmodule
```

This gives the same output as that of the assign statement, with the only difference that y is declared as reg. There are distinct advantages to have reg modeled as combinational element; reg type is useful when a "case" statement is required.

Revision with Version

Verilog 1995

Verilog has undergone a few revisions and the original IEEE version in 1995 had the following way for port declaration. Here, module declaration had to first list the names of ports within the brackets and then direction of those ports defined later within the body of the module.

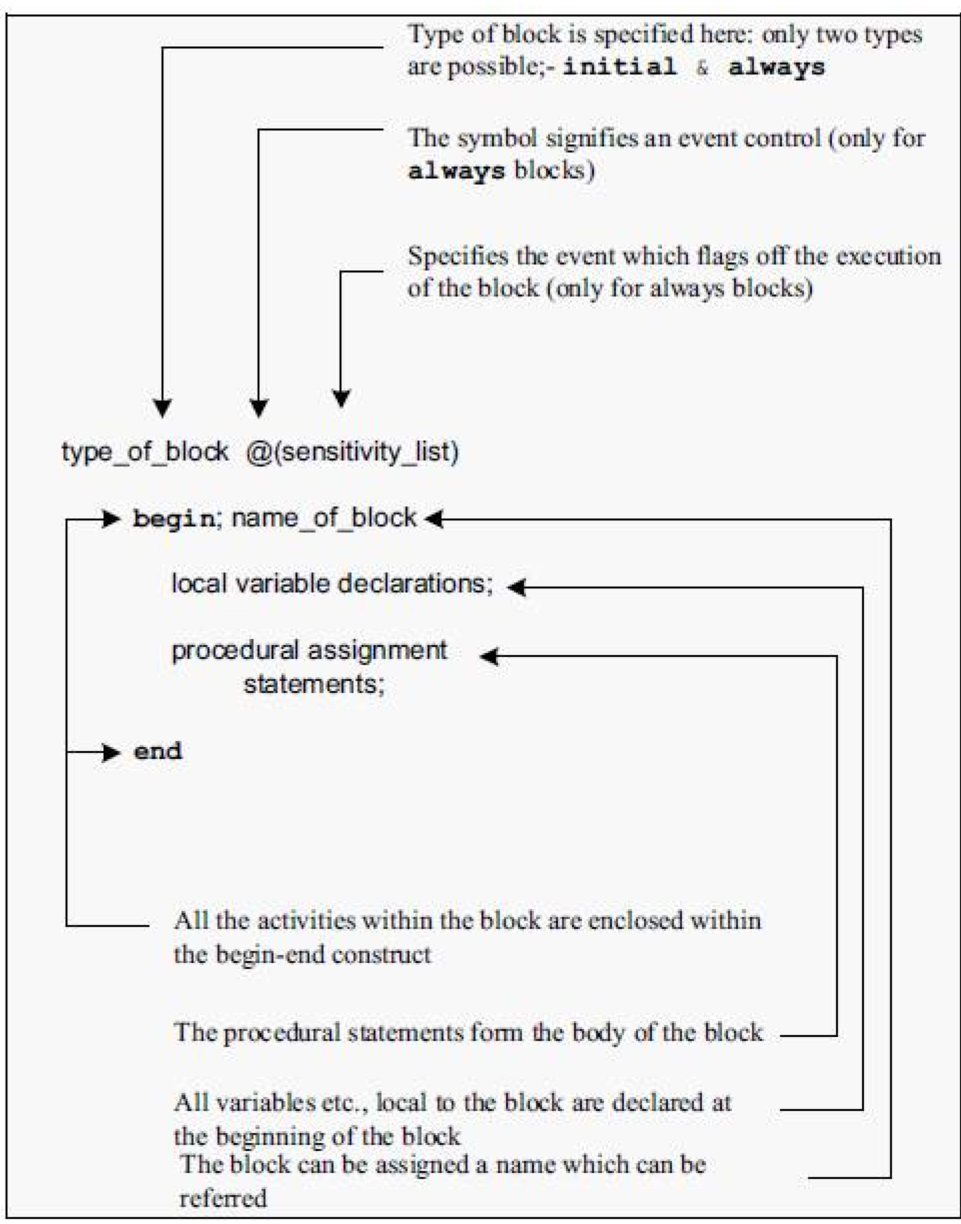
```
module test (a, b, c);

    input   [7:0] a;            // inputs "a" and "b" are wires
    input   [7:0] b;
    output  [7:0] c;            // output "c" by default is a wire

    // Still, you can declare them again as wires to avoid confusion
    wire    [7:0] a;
    wire    [7:0] b;
    wire    [7:0] c;
endmodule


module test (a, b, c);

    input  [7:0] a, b;
    output [7:0] c;           // By default c is of type wire

    // port "c" is changed to a reg type
    reg    [7:0] c;
endmodule
```

Verilog 2001 onwards

ANSI-C style port naming was introduced in 2001 and allowed the type to be specified inside the port list.

```
module test ( input [7:0]   a,
                            b,      // "b" is considered an 8-bit input
              output [7:0]  c);

    // Design content
endmodule

module test ( input wire [7:0]  a,
              input wire [7:0]  b,
              output reg [7:0]  c);

    // Design content
endmodule
```

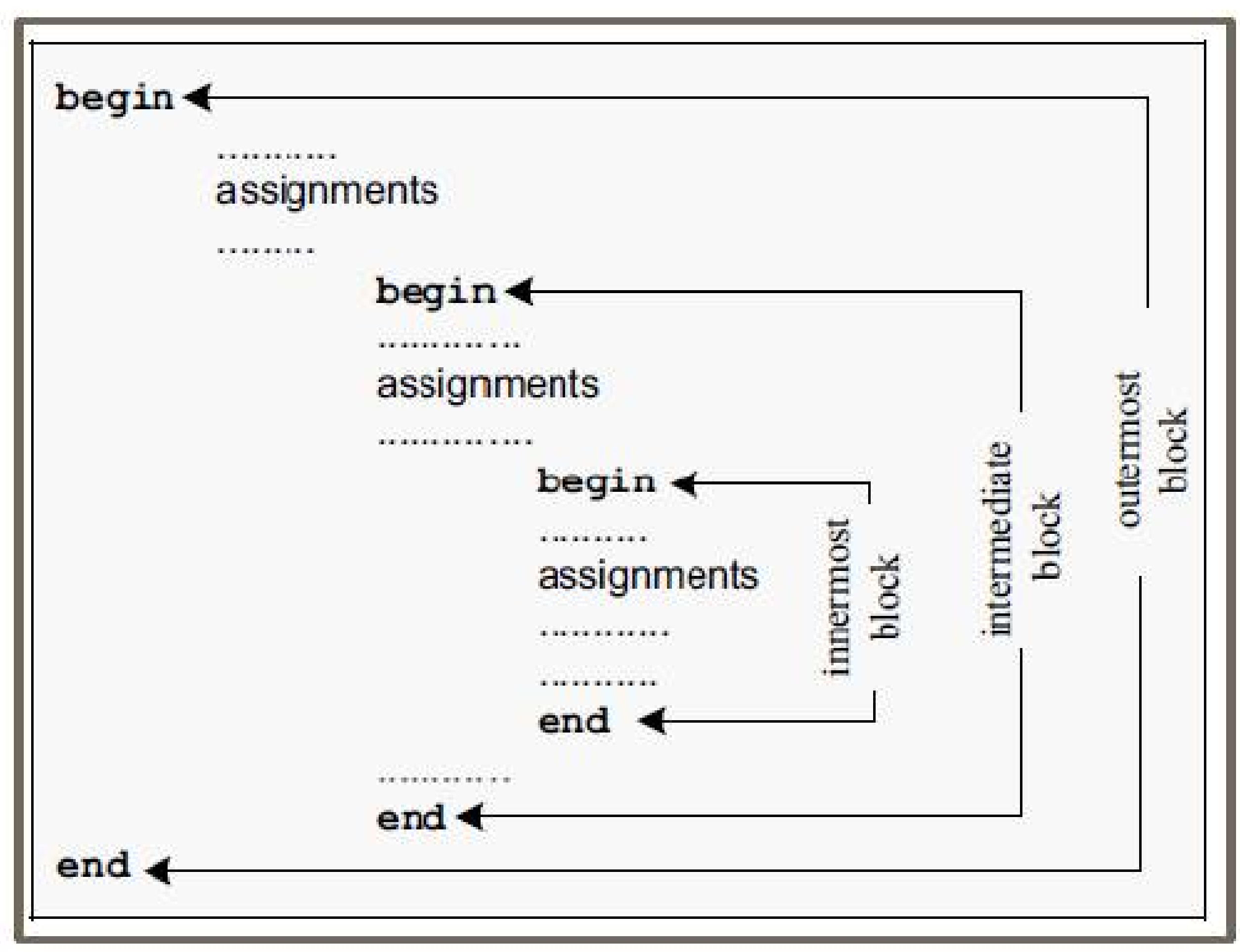
Procedural block



If a procedural block has only one assignment to be carried out, it can be specified

- as initial #2 a=0;
- More than one procedural assignment is to be carried out in an initial block. All such assignments are grouped together between "begin" and "end" declarations.
- Every begin declaration must have its associated end declaration.
- begin –end constructs can be nested as many times as desired.

NESTED **BEGIN – END** BLOCKS



- **Initial block**
  - Will be executed only once, at first time the unit is called (Only in testbench)

```
initial begin
  a = 1'b0;
  b = 1'b0;
end
```

- **Always block**
  - Statements will be evaluated when a change in sensitivity list occurs
  - **Example1** - sync reset, rising edge triggered flop:

```
always @(posedge clock)
  if (!nreset)
    q <= 1'b0;
  else
    q <= d;
```

  - **Example 2** - async reset, rising edge triggered, load enable flop:

```
always @(posedge clock or negedge nreset)
  if (!nreset)
    q <= 1'b0;
  else if (load_enable)
    q <= d;
```

i. Initial statement: This statement executes only once.

ii. Always statement: This statement always executes in a loop, that is, the statement is executed repeatedly.

Only a register data type can be assigned a value in either of these statements. Such a data type retains its value until a new value is assigned. All initial statements and always statements begin execution at time 0 concurrently.

- The block has three inputs and two outputs.
- Sum,Cout , T1 , T2 and T3 are declared to be as type reg
- They are assigned within always statement.
- Always statement has sequential block within begin end pair associated with event control.
- It means when an event occurs on A, B or Cin the sequential block executed
- Statements executed sequentially and execution suspended after the last statement in the sequential block has executed

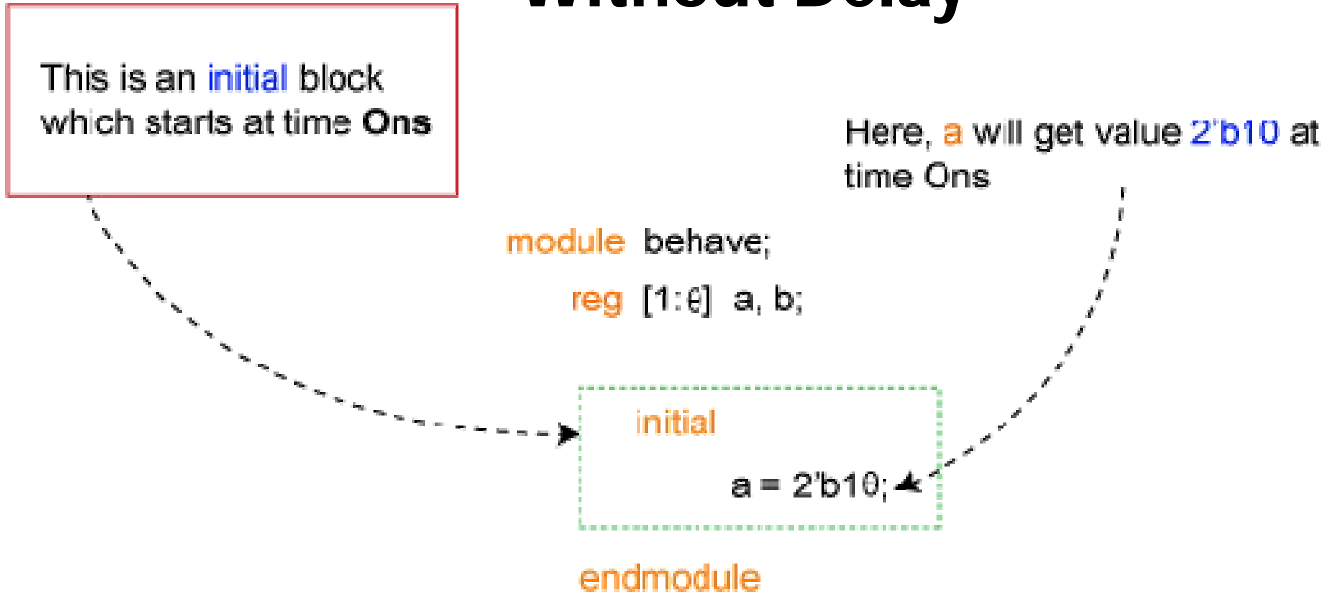
- The ***always*** block indicates a free-running process, but the ***initial*** block indicates a process executes exactly once. Both constructs begin execution at simulator time 0, and both exec
- Initial blocks can be used in either synthesizable or non-synthesizable blocks. They are commonly used in test.
- Initial blocks cause particular instructions to be performed at the beginning of the simulation before any other instructions operate. Initial blocks only operate once.

Initial

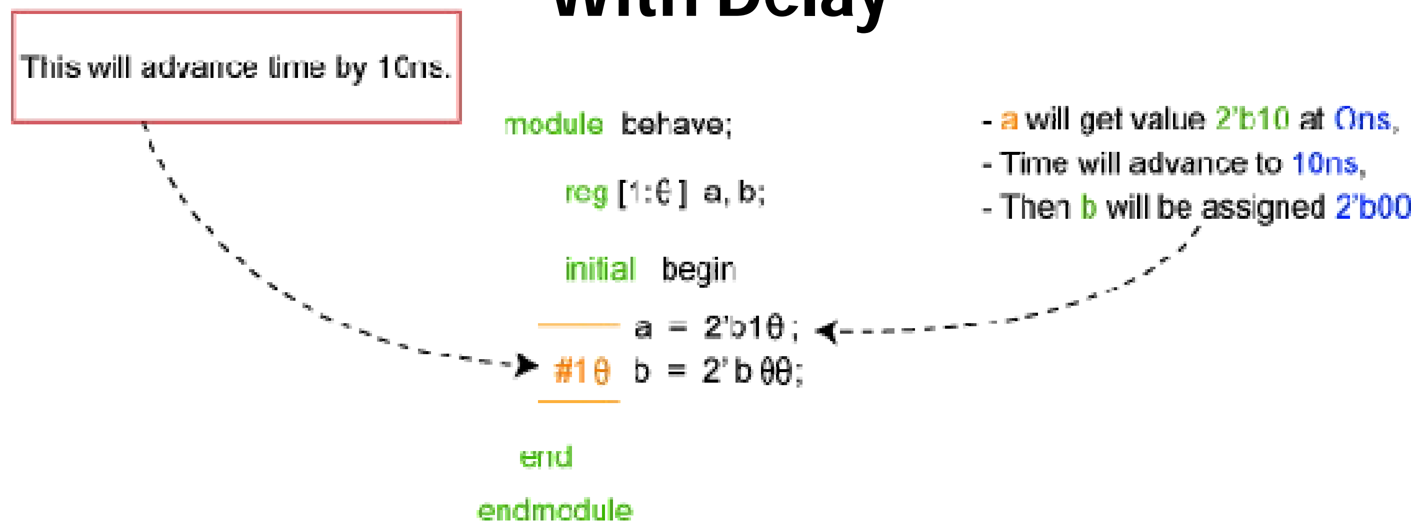
Example

- The register variables **a** and **b** are initialized to binary 1 and 0 respectively at simulation time zero.
- The initial statement is completed and not executed again during that simulation run. This initial statement is containing a begin-end block of statements. In this begin-end type block, **a** is initialized first, followed by **b**.

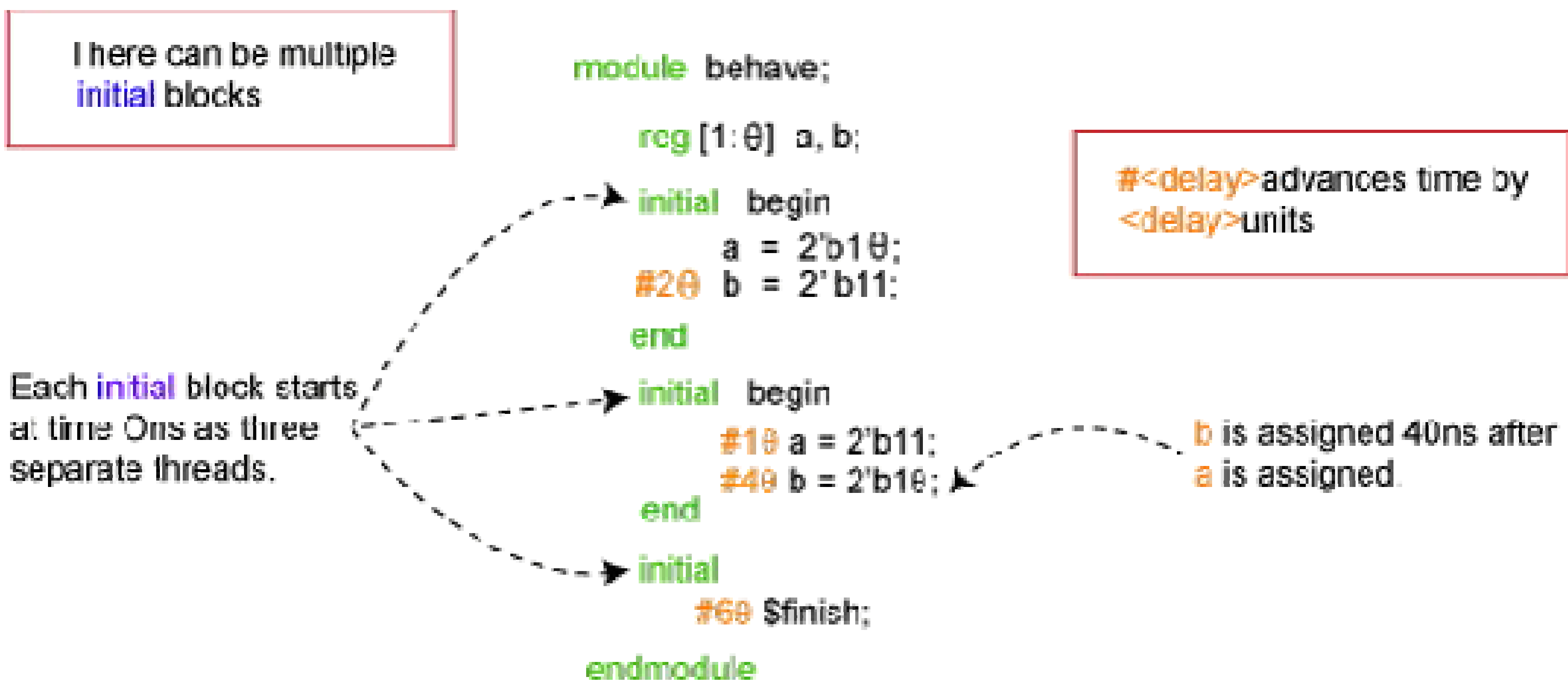
Without Delay



With Delay



There are no limits to the number of initial blocks that can be defined inside a module. The code shown below has three initial blocks, all of which are started at the same time and run in parallel.



$finish is a Verilog system task that tells the simulator to terminate the current simulation

In the above image, the first block has a delay of 20 units, while the second has a total delay of 50 units (10 + 40), and the last block has a delay of 60 units. Hence the simulation takes 60-time units to complete since there is atleast one initial block still running until 60-time units.

Always

- An always block always executes, unlike initial blocks that execute only once at the beginning of the simulation. The always block should have a sensitive list or a delay associated with it

```
always @ (event)
   [statement]

always @ (event) begin
   [multiple statements]
end
```

The symbol @ after reserved word *always*, indicates that the block will be triggered *at* the condition in parenthesis after symbol @.

```
always  @ (x or y or sel)
begin
    m = 0;
  if (sel == 0) begin
    m = x;
   end else begin
    m = y;
   end
end
```

```
always
   <single statement>

always @(<sensitivity_list>)
   <single statement>

always @(<sensitivity_list>) begin
   <statement 1>
   <statement 2>
   <statement 3>
   ...
   ...
end
```

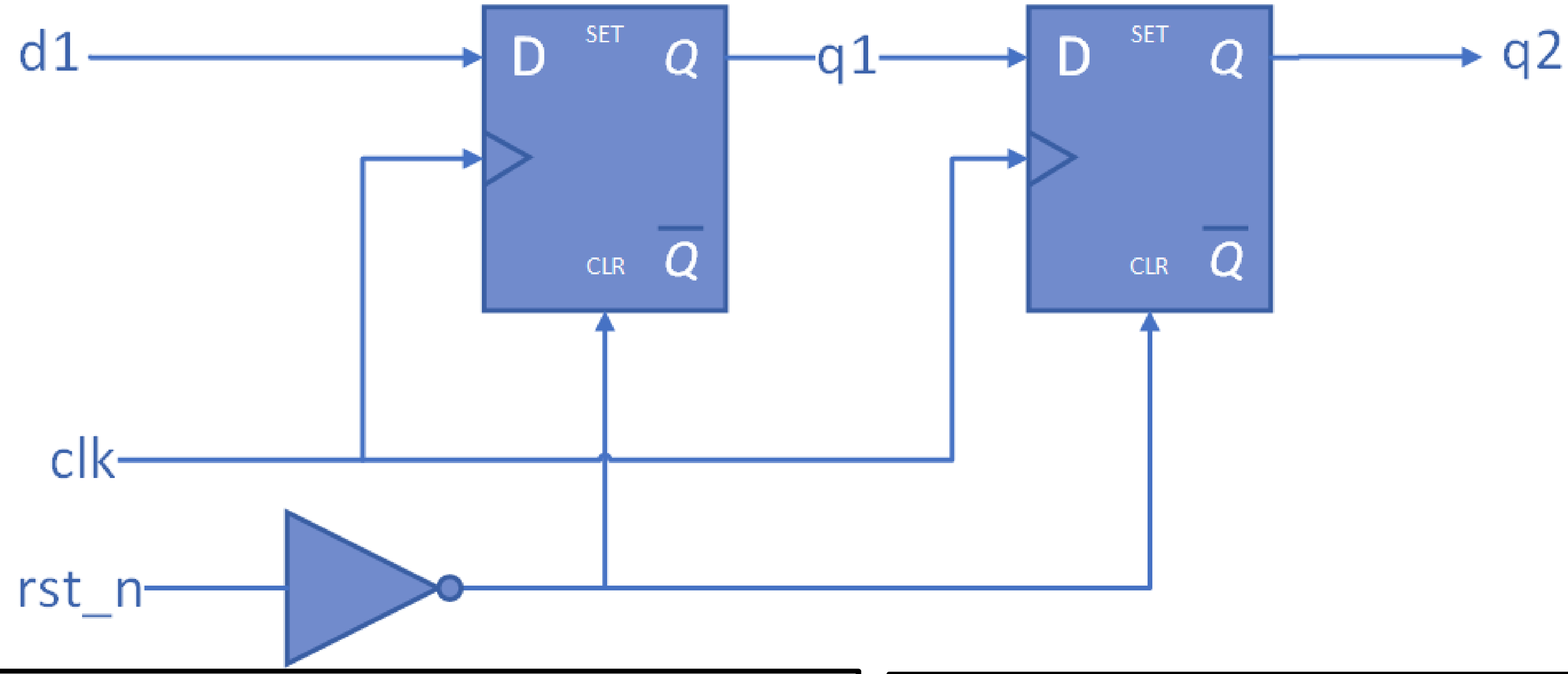
- In the above example, we describe a 2:1 mux, with input x and y. The ***sel*** is the select input, and ***m*** is the mux output.
- In any combinational logic, output changes whenever input changes. When this theory is applied to always blocks, then the code inside always blocks needs to be executed whenever the input or output variables change.

There are two types of sensitive list in the Verilog, such as:

- Level sensitive (for combinational circuits).
- Edge sensitive (for flip-flops).

```
always @ (posedge clk) begin
   [statements]
end
```

```
module tff (input  d, clk, rstn, output reg q);
always @ (posedge clk or negedge rstn) begin
if (!rstn)
   q <= 0;
else
   if (d)
      q <= ~q;
   else
      q <= q;
end
endmodule
```




```
always @(posedge clk or negedge rst_n)
   if (!rst_n)
      q1 <= 1'b0;
   else
      q1 <= d1;

always @(posedge clk or negedge rst_n)
   if (!rst_n)
      q2 <= 1'b0;
   else
      q2 <= q1;
```

```
always @(posedge clk or negedge rst_n)
   if (!rst_n) begin
      q1 <= 1'b0;
      q2 <= 1'b0;
   end
   else begin
      q1 <= d1;
      q2 <= q1;
   end
```

Modeling Combinational Logic

- The Verilog always block can also model combinational logic, but it is a bit less straight forward to understand.
- A physical implementation of a combinational circuit obviously operates continuously, sampling the inputs and calculating the resulting outputs.
- A simulator, however, cannot execute a logical statement "continuously", without causing the zero-delay infinite loop described at the beginning of the article.
- Therefore to simulate combinational circuit, we have to define specific events at which the simulator should execute procedures, while maintaining correct behavior.
- The simple answer is, to model combinational circuit, the sensitivity list needs to contain all the inputs to the circuit (all the variables on the right hand side of the assignment).

The following code describes a combinational AND gate using a Verilog always block.

```
always @(A or B)
   C = A & B;
```

Using Comma in Event Expression

Verilog-2001 standard introduced the use of comma "," to separate items in the event expression.

```
always @(posedge clk, negedge rst_n)
   if (!rst_n)
      q <= 1'b0;
   else
      q <= d;
```

Implicit Event Expression @* or @(*)

- Sensitivity list is a frequent source of bugs in a Verilog design/model.
- More precisely, the "@*" and "@(*)" syntax will add all nets and variables that appear in the (right hand side of a) statement to the event expression, with some exceptions.

```
always @*
   C = A & B;

always @(*)
   C = A & B;
```

always@(*) blocks are used to describe Combinational Logic, or Logic Gates. Only = (blocking) assignments should be used in an always@(*) block. Never use <= (non-blocking) assignments in always@(*) blocks. Only use always@(*) block when you want to infer an element(s) that changes its value as soon as one or more of its inputs change.

Verilog Block

- The block statements are the grouping of two or more statements together, which act syntactically like a single statement. There are two types of blocks
- Sequential block
- Parallel block
- These blocks can be used if more than one statement should be executed. All statements in the ***sequential*** blocks will be executed sequentially in the given order.
- All statements in the ***parallel*** blocks are executed at the same time or concurrently. It means that the next statement's execution will not be delayed even if the previous statement contains a timing control statement.

- **Sequential**
  - Asserted by a clock in the sensitivity list.
  - Translates into flip-flops/latches.

```
always @(posedge clock or negedge nreset)
  if (!nreset)
    q <= 1'b0;
  else if (load_enable)
    q <= d;
```

- **Combinational**
  - Describes purely combinational logic, and therefore, the sensitivity list has (non-clock) signals.

```
always @(a or b or c)
  out = a & b & c;
```

Blocking and Non-blocking

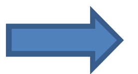
- Blocking and non-blocking assignments statements within the always block.
- The blocking assignment is similar to software assignment statements found in most popular programming languages.
- The non-blocking assignment is the more natural assignment statement to describe many hardware systems, especially for synthesis.
- The blocking assignments can only be used in a few situations, such as modeling combinational logic, defining functions, or implementing testbench algorithms.

- Non-blocking assignments happen in parallel. In other words, if an always@ block contains multiple <= assignments, which are literally written in Verilog sequentially, you should think of all of the assignments being set at exactly the same time.
- Blocking assignments happen sequentially. In other words, if an always@ block contains multiple = assignments, you should think of the assignments being set one after another.

Assignment operator

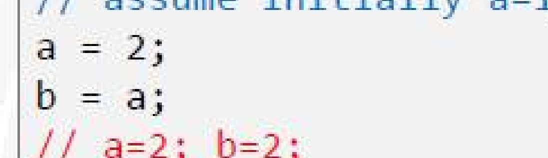
Verilog has three types of assignments:

- **Continuous assignment**
  - Outside of `always` blocks



```
assign muxout = (sel&in1) | (~sel&in0);
assign muxout = sel ? in1 : in0;
```

- **Blocking procedural assignment “= ”**
  - RHS is executed and assignment is completed before the next statement is executed.



```
// assume initially a=1;
a = 2;
b = a;
// a=2; b=2;
```

- **Non-blocking procedural assignment “<= ”**
  - RHS is executed and assignment takes place at the end of the current time step (not clock cycle)

```
// assume initially a=1;
a <= 2;
b <= a;
// a=2; b=1;
```

- Combinational always block: Use **blocking** assignments (=)
- Sequential always block: Use **non-blocking** assignments (<=)
- **Do not mix** blocking and non-blocking in the same always block
- **Do not assign to the same variable** from more than one always block

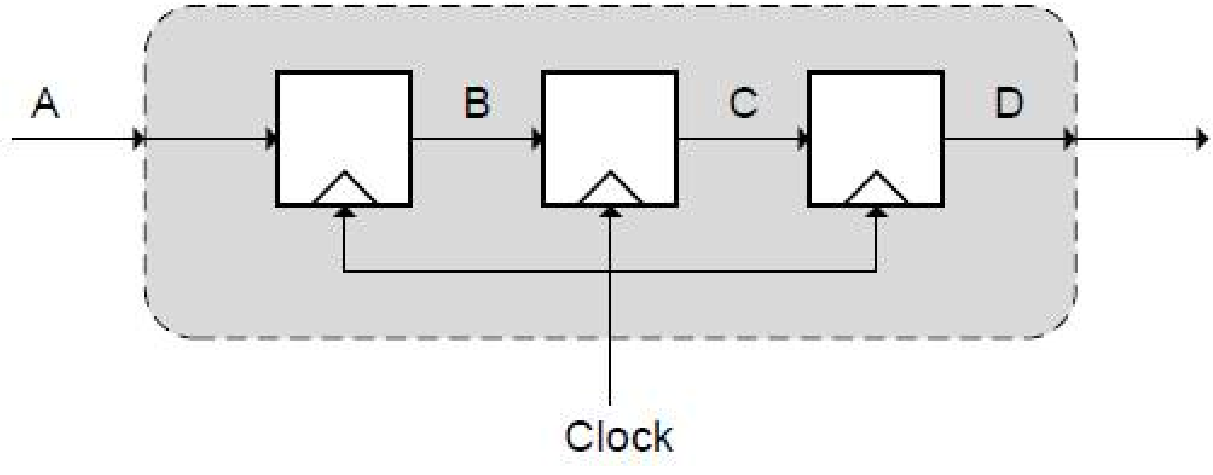
```
4 always @( ... sensitivity list ... ) begin
5       B <= A;
6       C <= B;
7       D <= C;
8 end
```

It species a circuit that reads “when the sensitivity list is satisfied”, B gets A's value, C gets B's old value, and D gets C's old value." The key here is that C gets B's old value, etc This ensures that C is not set to A, as A is B's new value, as of the always@ block's execution.

```
1 always @( ... sensitivity list ... ) begin
2       B = A;
3       C = B;
4       D = C;
5 end
```

It species a circuit that reads “when the sensitivity list is satisfied”, B gets A, C gets B, and D gets C." But, by the time C gets B, B has been set to A. Likewise, by the time D gets C, C has been set to B, which, as we stated above, has been set to A.

- always@(posedge Clock) ("always at the positive edge of the clock") or always@(negedge Clock) ("always at the negative edge of the clock") blocks are used to describe Sequential Logic, or Registers.
- Only <= (non-blocking) assignments should be used in an always@(posedge Clock) block.
- Never use =(blocking) assignments in always@(posedge Clock) blocks. Only use always@(posedge Clock) blocks when you want to infer an element(s) that changes its value at the positive or negative edge of the clock.

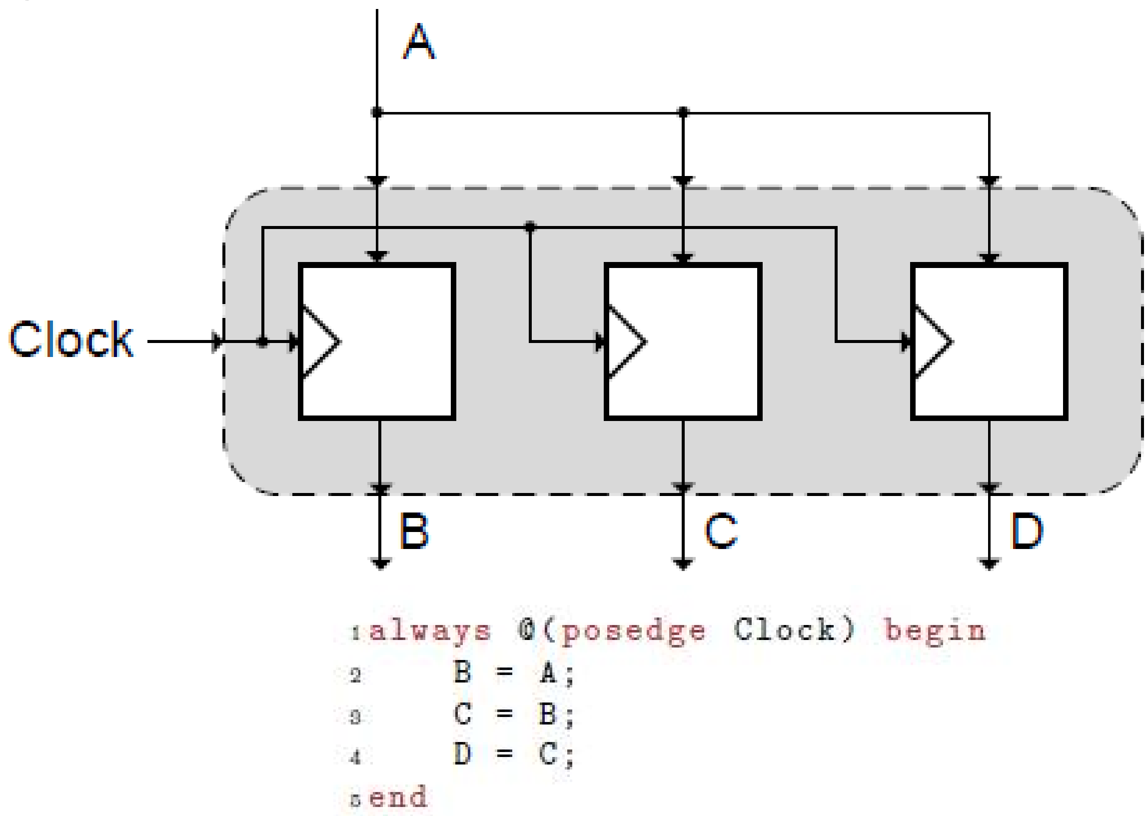



```
always @(posedge Clock) begin
    B <= A;
    C <= B;
    D <= C;
end
```

This is a shift register

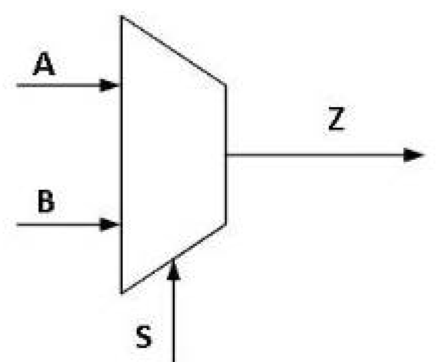
If we are not following this <= operator within always and use = operator then what will happen?

- Consider the shift register from prev slide. If you place = assignments inside of an always@(posedge Clock) block to produce the shift register, you instead get the parallel registers.

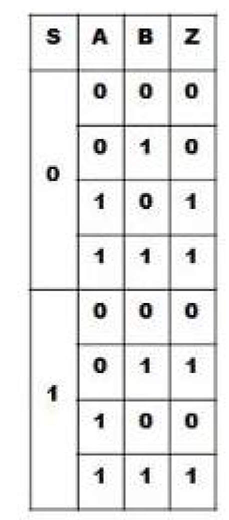


Comparative example: Combinational

Mux design



$$Z = (A.\bar{S}) + (B.S)$$



S	A	B	Z
0	0	0	0
	0	1	0
	1	0	1
	1	1	1
1	0	0	0
	0	1	1
	1	0	0
	1	1	1

- Using an `assign` Statement:

```
wire out;
assign out = sel ? a : b;
```

If conditional expression?
True expression : false expression;

- Using a `case` statement:

```
reg out;
always @ (a or b or sel)
  begin
    case (sel)
        1'b0: out=b;
        1'b1: out=a;
    endcase
  end
```

- Using an `always` Block:

```
reg out;
always @ (a or b or sel)
  if (sel)
    out=a;
  else
    out=b;
```

Comparative example: Sequential

- A simple D-Flip Flop:

```
reg q;
always @(posedge clk)
  q<= d;
```

- An asynch reset D-Flip Flop:

```
reg q;
always @(posedge clk or negedge reset_)
  if (~reset_)
    q<= 0;
  else
    q<= d;
```

- Be careful not to infer latches!!!:

```
reg q;
always @(en)
   if (en)
     q<= d;
```

```
// Blocking example
    x = 0;          // x changes at t = 0
    a = 1;          // a changes at t = 0
    #10 c = 3;      // delay until t=10, then c changes
    #15 d = 4;      // delay until t=25, then d changes
    e = 5;          // e changes at t = 25

//Non-blocking example
    x = 0;          //execute at t = 0
    a = 1;          //execute at t = 0
    c <= #15  3;    //evaluate at t = 0, schedule c to change at t = 15
    d <= #10  4;    //evaluate at t = 0, schedule d to change at t = 10
    c <= c + 1;     //evaluate at t = 0, schedule c to change at t = 0
```

```
initial begin
    a = 1; b = 0;   //block until after change at t=0
    #1  b = 1;      //delay until t=1; then block until b=1
    c = #1 1;       //block until t=2, c=val from t=1
    #1;             //delay to t=3
    d = 1;          //block until change at t=3
    e <= #1 1;      //non-blocking, update e at t=4
    #1  f <= 1;     //delay to t=4, non-blocking f update
    g <= 1;         //still t=4, non-blocking g update
end
```

Results:

t	a	b	c	d	e	f	g
0	**1**	**0**	x	x	x	x	x
1	1	**1**	x	x	x	x	x
2	1	1	**1**	x	x	x	x
3	1	1	1	**1**	x	x	x
4	1	1	1	1	1	1	1

wire vs reg in block

1. Inside **always** blocks (both sequential and combinational) only **reg** can be used as LHS.
2. For an **assign** statement, only **wire** can be used as LHS.
3. Inside an **initial** block (Testbench) only **reg** can be used on the LHS.
4. The **output** of an instantiated module can only connect to a **wire**.
5. **Inputs** of a **module** cannot be a **reg**.

```
reg r;
always @*
  r = a & b;
```

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

```
reg r;
initial
  begin
      r = 1'b0;
    #1
      r = 1'b1;
  end
```

```
module m1 (out)
  output out;
endmodule

reg r;
m1 m1_instance(.out(r));
```

Assign statement

All delays in a Verilog HDL model are specified in terms of time units. Here is an example of a continuous assignment with a delay.

```
assign #2 Sum = A ^ B;
```

The #2 refers to 2 time units.

The assignment syntax starts with the keyword assign, followed by the signal name, which can be either a signal or a combination of different signal nets.

The ***drive strength*** and ***delay*** are optional and mostly used for dataflow modeling than synthesizing into real hardware. The signal on the right-hand side is evaluated and assigned to the net or expression of nets on the left-hand side.

assign <net_expression> = [drive_strength] [delay] <expression of different signals or constant value>

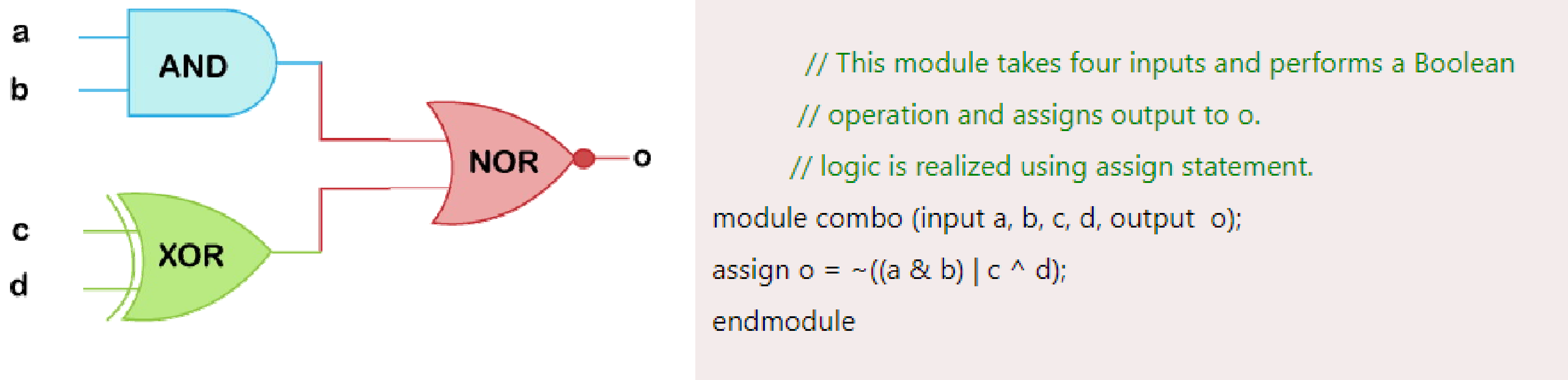
Rules: Assign

Some rules need to be followed during the use of an assign statement:

- LHS should always be a scalar, vector, or a combination of scalar and vector nets but never a scalar or vector register.
- RHS can contain scalar or vector registers and function calls.
- Whenever any operand on the RHS changes in value, LHS will be updated with the new value.
- **Assign statements are also called continuous assignments.**
- **Cannot drive or assign *reg* type variables with an assign statement because a reg variable is capable of storing data and is not driven continuously.**

Implicit vs Explicit

When an assign statement is used to assign the given net with some value, it is called an ***explicit*** assignment.
If an assignment to be done during the net is declared, it is called an ***implicit*** assignment.

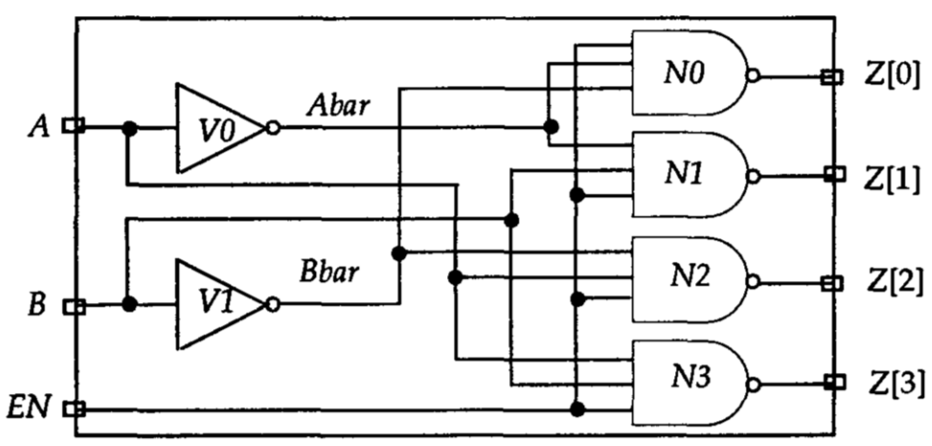


Combinational logic requires the inputs to be continuously driven to maintain the output, unlike sequential elements like flip flops where the value is captured and stored at the edge of a clock.

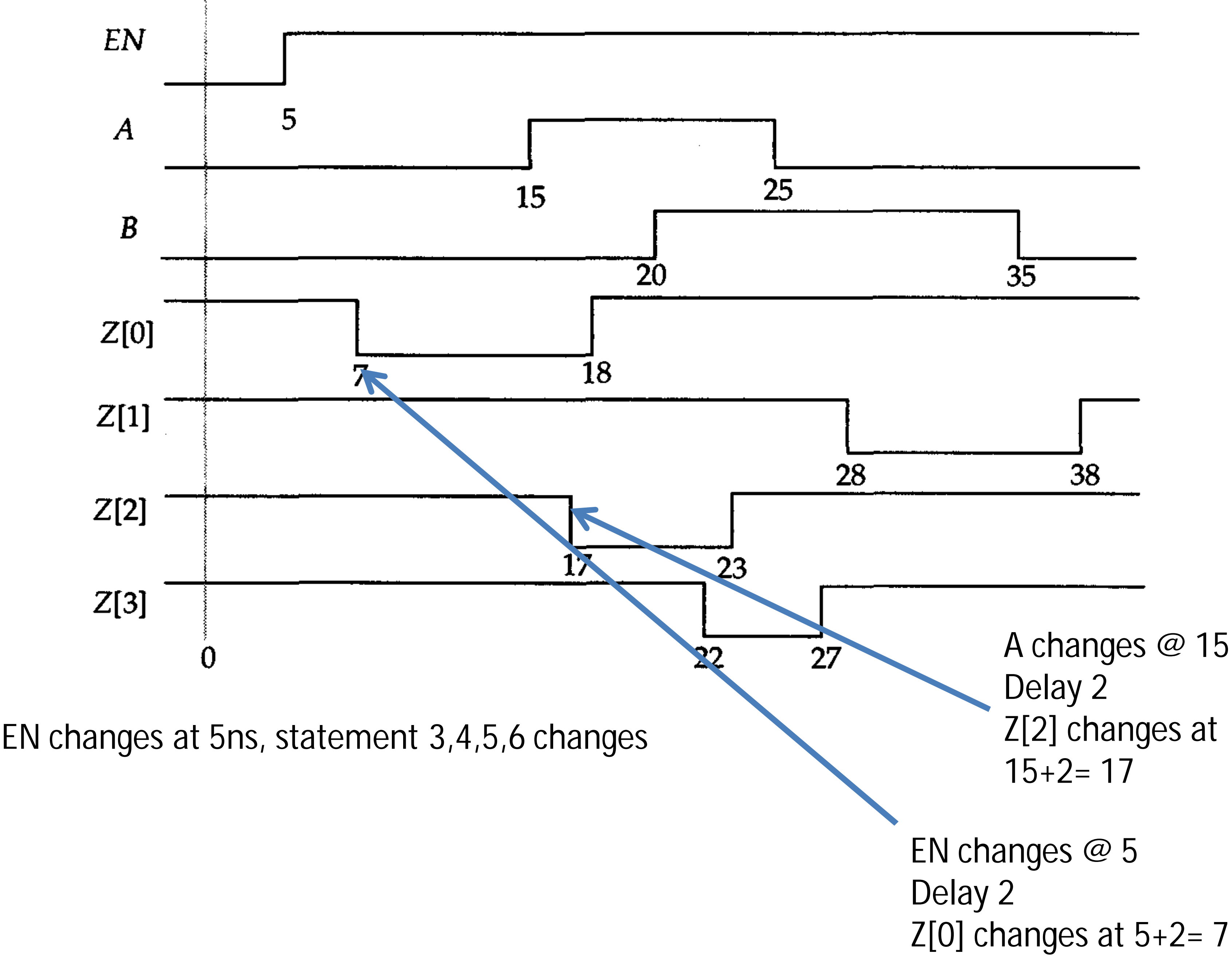
Delay , Timescale

- Verilog simulation depends on how time is defined because the simulator needs to know what a #1 means in terms of time. The `timescale compiler directive specifies the time unit and precision for the modules that follow it.
- The ***time_unit*** is the measurement of delays and simulation time, while the ***time_precision*** specifies how delay values are rounded before being used in the simulation.
- ***$printtimescale*** system task to display time unit and precision.

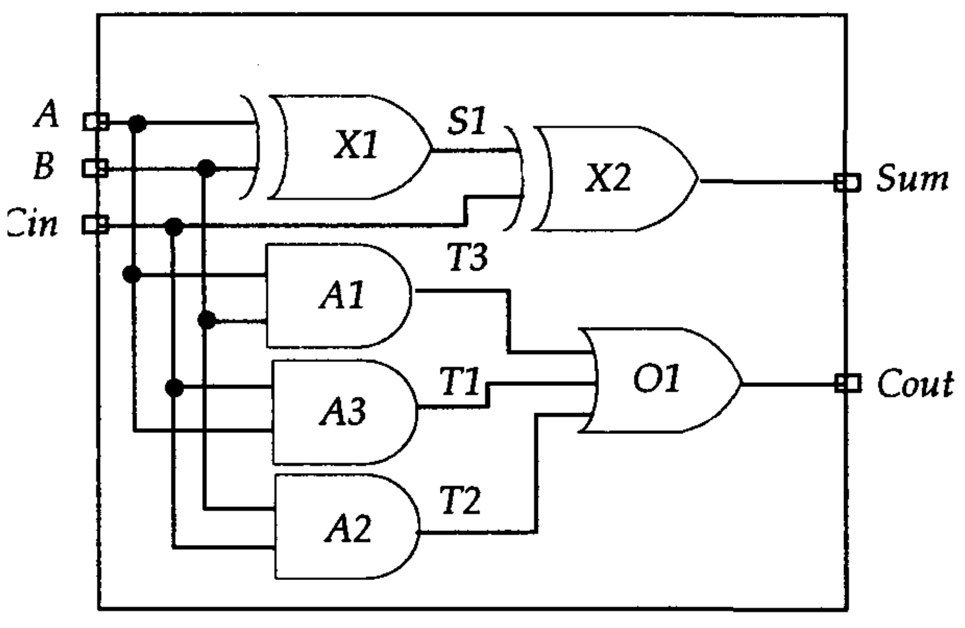
Dataflow style




```
module Decoder2x4 (A, B, EN, Z);
  input A, B, EN;
  output [0:3] Z;
  wire Abar, Bbar;

  assign #1 Abar = ~A;                          // Stmt 1.
  assign #1 Bbar = ~B;                          // Stmt 2.
  assign #2 Z[0] = ~ (Abar & Bbar & EN);        // Stmt 3.
  assign #2 Z[1] = ~ (Abar & B & EN);           // Stmt 4.
  assign #2 Z[2] = ~ (A & Bbar & EN);           // Stmt 5.
  assign #2 Z[3] = ~ (A & B & EN);              // Stmt 6.
endmodule
```


EN
A
B
Z[0]
Z[1]
Z[2]
Z[3]
0
5
15
25
20
35
7
18
28
38
17
23
22
27
A changes @ 15
Delay 2
Z[2] changes at
15+2= 17
EN changes at 5ns, statement 3,4,5,6 changes
EN changes @ 5
Delay 2
Z[0] changes at 5+2= 7

Behavioral Style




```
module FA_Seq (A, B, Cin, Sum, Cout)
  input A, B, Cin;
  output Sum, Cout;
```

```
reg Sum, Cout;
reg T1, T2, T3;

always
  @ (A or B or Cin) begin
    Sum = (A ^ B) ^ Cin;
    T1 = A & Cin;
    T2 = B & Cin;
    T3 = A & B;
    Cout = (T1 | T2) | T3;
  end
endmodule
```

Structural Modeling

Structure can be described in Verilog HDL using:

i. Built-in gate primitives (at the gate-level)

ii. Switch-level primitives (at the transistor-level)

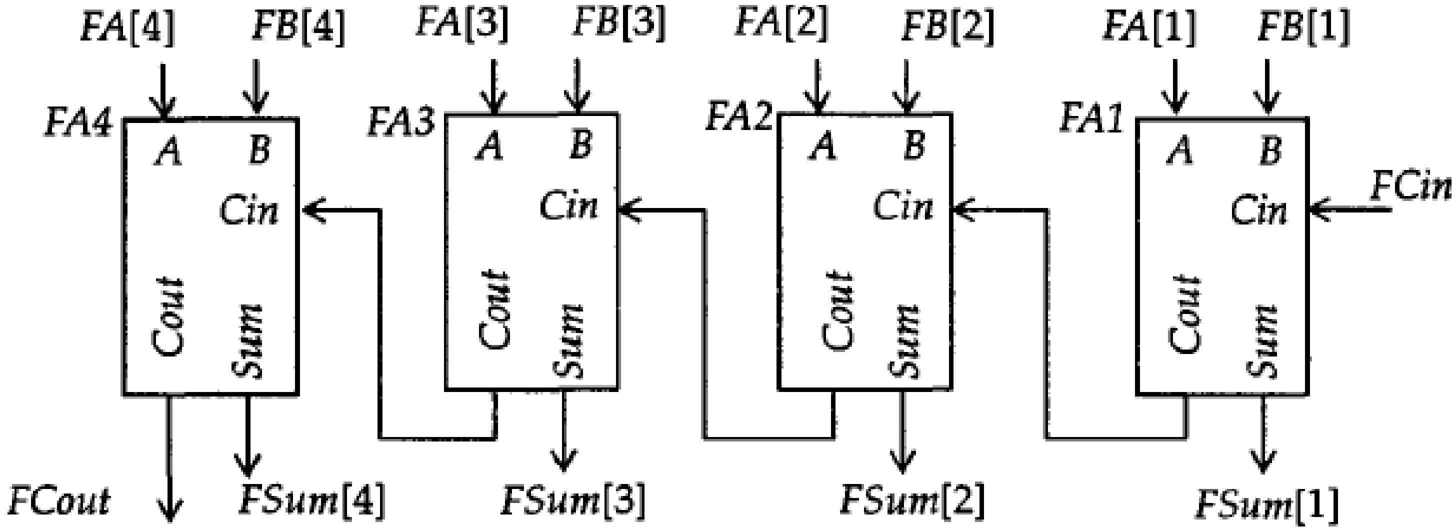
iii. User-defined primitives (at the gate-level)

iv. Module instances (to create hierarchy)

```
module FA_Str (A, B, Cin, Sum, Cout);
  input A, B, Cin;
  output Sum, Cout;
  wire S1, T1, T2, T3;

  xor
    X1 (S1, A, B),
    X2 (Sum, S1, Cin);

  and
    A1 (T3, A, B),
    A2 (T2, B, Cin),
    A3 (T1, A, Cin);

  or
    O1 (Cout, T1, T2, T3);
endmodule
```

In this example, the module contains gate instantiations, that is, instances of built-in gates **xor**, **and**, and **or**. The gate instances are interconnected by nets *S1*, *T1*, *T2*, and *T3*. The gate instantiations can appear in any order since no sequentiality is implied; pure structure is being shown; **xor**, **and** and **or** are built-in gate primitives; *X1*, *X2*, *A1*, etc. are the instance names. The list of signals following each gate are its interconnections; the first one is the output of the gate and the rest are its inputs. For example, *S1* is connected to the output of the `xor` gate instance *X1* while *A* and *B* are connected to its inputs.

Now use the same one bit FA to a 4 bit FA



```
module FourBitFA (FA, FB, FCin, FSum, FCout);
  parameter SIZE = 4;
  input [SIZE:1] FA, FB;
  output [SIZE:1] FSum;
  input FCin;
  input FCout;
  wire [1:SIZE-1] FTemp;
```

Original mapped with present

```
  FA_Str
    FA1 (.A(FA[1]), .B(FB[1]), .Cin(FCin),
         .Sum(FSum[1]), .Cout(FTemp[1])),
    FA2 (.A(FA[2]), .B(FB[2]), .Cin(FTemp[1]),
         .Sum(FSum[2]), .Cout(FTemp[2])),
    FA3 (FA[3], FB[3], FTemp[2], FSum[3], FTemp[3]),
    FA4 (FA[4], FB[4], FTemp[3],FSum[4], FCout);
endmodule
```

Port Mapping

Port mapping by order
Port mapping by name

SYNCHRO

DFF1

DFF2

C1_ASYNC

ASYNC

CLOCK

SYNC

Figure shows module "SYNCHRO" which consists of 2 'D' flip-flops and are connected in serial fashion. Module "SYNCHRO" has 2 input ports "ASYNC" and "CLOCK" and 1 output port "SYNC".

Verilog Programming for DFF instantiated in the SYNCHRO module:

```
1 module DFF (Q, D, CLK);
2   input D, CLK;
3   output reg Q;
4   always @ (posedge CLK)
5     Q <= D;
6 endmodule
```

MODULE PORT MAPPING BY ORDER

```
module SYNCHRO (ASYNC,SYNC,CLOCK);
 input ASYNC;
 input CLOCK;
 output SYNC;
  wire C1_ASYNC;

 DFF DFF1 (C1_ASYNC, ASYNC, CLOCK);
 DFF DFF2 (SYNC, C1_ASYNC, CLOCK);

endmodule
```

Here first instance name of 'D' flip-flop is "DFF1" and second instance name is "DFF2". In this **module ports are connected by order**. Order of ports in instantiation of DFF1 and DFF2 is same as order of ports in DFF. If the number of ports increased, then it is very difficult to do "module ports connection by order".

MODULE PORT MAPPING BY NAME

```
Module SYNCHRO (ASYNC, SYNC, CLOCK);
 input ASYNC;
 input CLOCK;
 output SYNC;
  wire C1_ASYNC;
  DFF DFF1 (.D (ASYNC), .CLK (CLOCK), .Q (C1_ASYNC));
 DFF DFF2 (.D (C1_ASYNC), .Q (SYNC), .CLK (CLOCK));
endmodule
```

In this module, ports are connected by Name. Order of ports in instantiation of DFF1 and DFF2 is different from order of ports in DFF. In this '.' is used to represent port name followed by associated port name in small brackets i.e. "()".

Mixed Design

Here is an example of a 1-bit full-adder in a mixed-design style.

```
module FA_Mix (A, B, Cin, Sum, Cout);
  input A, B, Cin;
  output Sum, Cout;
  reg Cout;
  reg T1, T2, T3;
  wire S1;

  xor X1 (S1, A, B);          // Gate instantiation.

  always
    @ (A or B or Cin) begin   // Always statement.
      T1 = A & Cin;
      T2 = B & Cin;
      T3 = A & B;
      Cout = (T1 | T2) | T3;
    end

  assign Sum = S1 ^ Cin;      // Continuous assignment.
endmodule
```

Identifier & Comments

An *identifier* in Verilog HDL is any sequence of letters, digits, the $ character, and the _ (underscore) character, with the restriction that the first character must be a letter or an underscore. In addition, *identifiers are case-sensitive*. Here are some examples of identifiers.

```
Count
COUNT          // Distinct from Count.
_R2_D2
R56_68
FIVE$
```

Comments

There are two forms of comments in Verilog HDL.

```
/* First form: Can
  extend across
  many
  lines */

// Second form: Ends at the end of this line.
```

Format

Verilog HDL is case-sensitive. That is, identifiers differing only in their case are distinct. In addition, Verilog HDL is free-format, that is, constructs may be written across multiple lines, or on one line. White space (newline, tab, and space characters) have no special significance. Here is an example that illustrates this.

```
initial begin Top = 3'b001; #2 Top = 3'b011; end
```

is same as:

```
initial
  begin
    Top = 3'b001;
    #2 Top = 3'b011;
  end
```

Compiler directives

- The compiler directives are similar to C language preprocessor directives that may be used to specify certain information and ask the compiler to process it.
- The character ` is used prior to the specific keyword. Its declaration can be put outside of the module and its scope may not be limited to a single module.

Certain identifiers that start with the ` (backquote) character are compiler directives. A compiler directive, when compiled, remains in effect through the entire compilation process (which could span multiple files) until a different compiler directive specifies otherwise. Here is a complete list of standard compiler directives.

- `define, `undef
- `ifdef, `else, `endif
- `default_nettype
- `include
- `resetall
- `timescale
- `unconnected_drive, `nounconnected_drive
- `celldefine, `endcelldefine

Compiler directives	Description
`define	To define text macros. (Similar to #define in C language)
`include	To include entire content from another Verilog file into the existing file during compilation. (Similar to #include in C language).
`ifdef…`endif `ifdef..`else..`endif	Conditional compiler directives that behave as if..else conditional statements.
`timescale	To specify time units and precision for the module

The **`define** directive is used for text substitution and is very much like the #define in the C programming language. Here is an example of this directive.

```
`define MAX_BUS_SIZE 32
. . .
reg [ `MAX_BUS_SIZE - 1 : 0 ] AddReg;
```

Once the **`define** directive is compiled, the definition stays in effect through the entire compilation. For example the usage of *MAX_BUS_SIZE* could be across many different files with the **`define** directive in another file.

The **`undef** directive removes the definition of a previously defined text macro. Here is an example.

```
`define WORD 16   // Creates a macro for text substitution.
. . .
wire [ `WORD : 1 ] Bus;
. . .
`undef WORD

        // The definition of WORD is no longer available
        // after this `undef directive.
```

`include

The **`include** compiler directive can be used to include the contents of any file in-line. The file can be specified either with a relative path name or with a full path name.

```
`include "../../primitives.v"
```

`resetall

This compiler directive resets all compiler directives to their default value.

```
`resetall
```

For example, this directive causes the default net type to be wire.

`timescale

In a Verilog HDL model, all delays are expressed in terms of time units. The association of time units with actual time is done using the **`timescale** compiler directive. This directive is used to specify the time unit and time precision. The directive is of the form:

```
`timescale time_unit / time_precision
```

where the *time_unit* and *time_precision* is made up of values from 1, 10, and 100 and units from s, ms, us, ns, ps and fs. Here is an example.

```
`timescale 1ns / 100ps
```

indicates a time unit of 1ns and a time precision of 100ps. The **`timescale** directive appears outside of a module declaration and affects all delay values that follow it. Here is an example.

```
`timescale 1ns / 100ps
module AndFunc (Z, A, B);
  output Z;
  input A, B;

  and #(5.22, 6.17) A1 (Z, A, B);
    // Rise and fall delay specified.
endmodule
```

The directive specifies all delays to be in ns and delays are rounded to one-tenth of a ns (100ps). Therefore, the delay value 5.22 becomes 5.2ns and the delay value 6.17 becomes 6.2ns. If instead the following **`timescale** directive is used in the above module,

```
`timescale 10ns / 1ns
```

then 5.22 becomes 52ns, and 6.17 becomes 62ns.

Multiple Driver

```
wire Reset;
wire [3:2] Cla, Pla, Sla;
tri [MSB-1 : LSB+1] Art;
```

```
assign Cla = Pla & Sla;
. . .
assign Cla = Pla ^ Sla;
```

tri net used for multiple driver

If multiple drivers drive a wire then following table determine the stages

Wire	0	1	x	z
0	0	x	x	0
1	x	1	x	1
x	x	x	x	x
z	0	1	x	z

In this example, *Cla* has two drivers. The values of the two drivers (the values of the right-hand side expressions) is used to index in the above table to determine the effective value of *Cla*. Since *Cla* is a vector, each bit position is evaluated independently. For example, if the first right-hand side expression has the value **01x** and the second right-hand side expression has the value **11z**, the effective value of *Cla* is **x1x** (the first bits 0 and 1 index into the table to give an **x**, the second bits 1 and 1 index into the table to give a 1, the third bits **x** and **z** index into the table to give an **x**).

Wor and Trior

This is a wired-or net, that is, if any one of the drivers is a 1, the value on the net is also a 1. Both wor and trior nets are identical in their syntax and functionality.

```
wor [MSB : LSB] Art;
trior [MAX-1 : MIN-1] Rdx, Sdx, Bdx;
```

Wire	0	1	x	z
0	0	1	x	0
1	1	1	1	1
x	x	1	x	x
z	0	1	x	z

Wand and Triand

This net is a wired-and net, that is, if any of the drivers is a 0, the value of the net is a 0. Both wand and triand nets are identical in their syntax and functionality.

```
wand [-7:0] Dbus;
triand Reset, Clk;
```

Wire	0	1	x	z
0	0	0	0	0
1	0	1	x	1
x	0	x	x	x
z	0	1	x	z

Trireg net

This net stores a value (like a register) and is used to model a capacitive node. When all drivers to a trireg net are at high-impedance, that is, have the value z, the trireg net retains the last value on the net. In addition, the default initial value for a trireg net is an x.

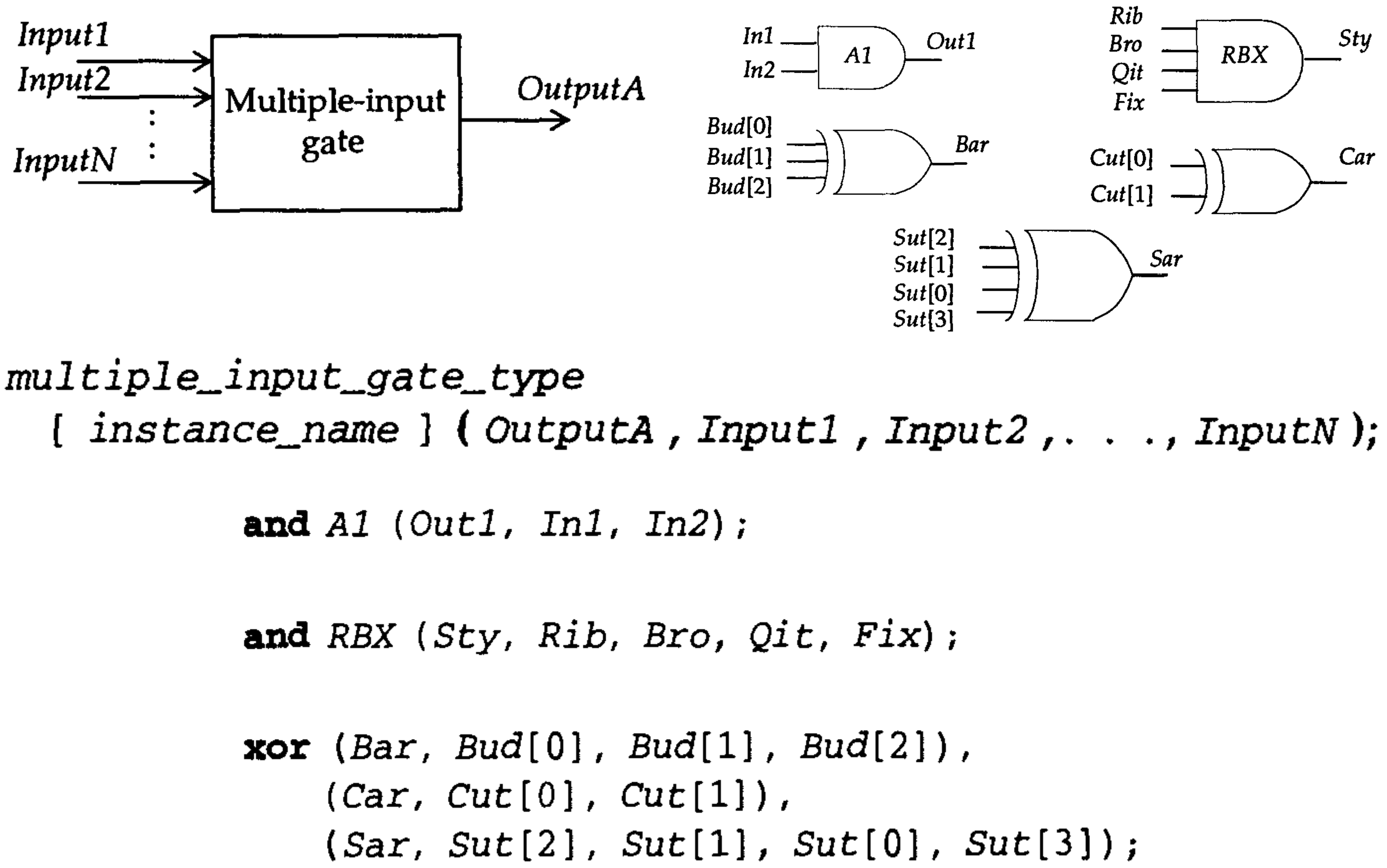
```
trireg [1:8] Dbus, Abus;
```

Tri0 and Tri1 Nets

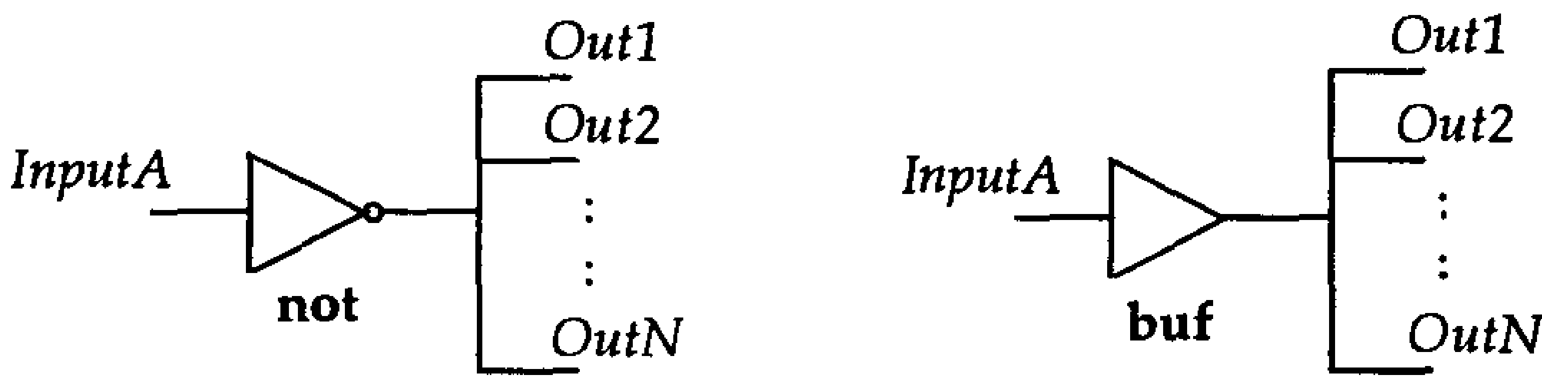
These nets also model wired-logic nets, that is, a net with more than one driver. The particular characteristic of a tri0 (tri1) net is that if no driver is driving this net, its value is 0 (1 for tri1).

```
tri0 [-3:3] GndBus;
tri1 [0:-5] OtBus, ItBus;
```

Multiple input gate



Multiple output gate




```
multiple_output_gate_type
        [ instance_name ] (Out1, Out2, . . . , OutN, InputA);
```

```
buf  B1  (Fan[0], Fan[1], Fan[2], Fan[3], Clk);
not  N1  (PhA, PhB, Ready);
```

buf	0	1	x	z
(output)	0	1	x	x

not	0	1	x	z
(output)	1	0	x	x

Conditional Statement (Behv model)

```
if ( condition_1 )
  procedural_statement_1
{ else if ( condition_2 )
        procedural_statement_2 }
      [ else
        procedural_statement_3 ]
```

```
if (Sum < 60)
  begin
    Grade = C;
    Total_C = Total_C + 1;
  end
else if (Sum < 75)
  begin
    Grade = B;
    Total_B = Total_B + 1;
  end
else
  begin
    Grade = A;
    Total_A = Total_A + 1;
  end
```

Case Statement

```
case ( case_expr )
  case_item_expr {, case_item_expr } : procedural_statement
  . .
        [ default: procedural_statement ]
  . .
     endcase
```

```
case (Day)
  TUE     : Pocket_Money = 6;     // Branch 1
  MON,
  WED     : Pocket_Money = 2;     // Branch 2
  FRI,
  SAT,
  SUN     : Pocket_Money = 7;     // Branch 3
  default : Pocket_Money = 0;     // Branch 4
endcase
```

Example

```
module ALU (A, B, OpCode, Z);
  input [3:0] A, B;
  input [1:2] OpCode;
  output [7:0] Z;
  reg [7:0] Z;
          parameter
            ADD_INSTR = 2'b10,
            SUB_INSTR = 2'b11,
            MULT_INSTR = 2'b01,
            DIV_INSTR = 2'b00;
                        always
                          @ (A or B or OpCode)
                            case (OpCode)
                              ADD_INSTR:     Z = A + B;
                              SUB_INSTR:     Z = A - B;
                              MULT_INSTR:    Z = A * B;
                              DIV_INSTR:     Z = A / B;
                            endcase
                        endmodule
```

Loop Statement

There are four kinds of loop statements. These are:

i. Forever-loop

ii. Repeat-loop

iii. While-loop

iv. For-loop

Forever-loop Statement

The syntax for this form of loop statement is:

```
forever
  procedural_statement
```

This loop statement continuously executes the procedural statement. Thus to get out of such a loop, a disable statement may be used with the procedural statement. Also, some form of timing controls must be used in the procedural statement, otherwise the forever-loop will loop forever in zero delay.

```
initial
  begin
    Clock = 0;
    #5 forever
      # 10 Clock = ~ Clock;
  end
```

Repeat-loop Statement

This form of loop statement has the form:

```
repeat ( loop_count )
  procedural_statement
```

It executes the procedural statement the specified number of times. If loop count expression is an **x** or a **z**, then the loop count is treated as a 0. Here are some examples.

```
repeat (Count)
  Sum = Sum + 10;

repeat (ShiftBy)
  P_Reg = P_Reg << 1;
```

What does the following mean?

```
repeat (NUM_OF_TIMES) @(negedge ClockZ);
```

It means to wait for *NUM_OF_TIMES* negative clock edges before executing the statement following the repeat statement.

While-loop Statement

The syntax of this form of loop statement is:

```
while ( condition )
  procedural_statement
```

```
while (By > 0)
  begin
    Acc = Acc << 1;
    By = By - 1;
  end
```

This loop executes the procedural statement until the specified condition becomes false. If the expression is false to begin with, then the procedural statement is never executed. If the condition is an **x** or a **z**, it is treated as a 0 (false). Here are some examples.

For-loop Statement

This loop statement is of the form:

```
for ( initial_assignment ; condition ; step_assignment )
  procedural_statement
```

A for-loop statement repeats the execution of the procedural statement a certain number of times. The *initial_assignment* specifies the initial value of the loop index. The *condition* specifies the condition when loop execution must stop. As long as the condition is true, the statements in the loop are executed. The *step_assignment* specifies the assignment to modify, typically to increment or decrement, the step count.

```
integer K;

for (K = 0; K < MAX_RANGE; K = K + 1)
  begin
    if (Abus[K] == 0)
      Abus[K] = 1;
    else if (Abus[K] == 1)
      Abus[K] = 0;
    else
      $display ("Abus[K] is an x or a z");
  end
```

User Defined primitive

- user can virtually argument predefined gate primitives by designing and specifying new primitive elements called user-defined primitives (UDPs).
- ***AND, NAND,*** NOT, ***OR,*** and ***NOR***, as a part of the language. Part of built-in primitive.
- Instances of these new UDPs can be used in the same manner as the gate primitives to represent the circuit being modeled.
- Each UDP has exactly one output, which can be in one of these states: 0, 1, or x.

- These two types of behavior can be represented in user-defined primitives:
  - Combinational UDP
  - Sequential UDP
- A sequential UDP uses the value of its inputs and the current value of its output to determine the next value of output. ***latches and flip-flops***.
- A sequential UDP can model both level-sensitive and edge-sensitive behavior.
- The maximum number of inputs to a sequential UDP is limited to 9 because the internal state counts as an input.

```
primitive UDP_name (output, input, …);
  port_declaration
  [ reg output; ]
  [ initial output = initial_value; ]
  table
    truth_table
  endtable
endprimitive
```

UDPs should be defined outside the ***module*** and ***endmodule***.

Hardware behavior is described as a primitive state table that lists out a different possible combination of inputs and their corresponding output within the ***table*** and ***endtable***.

Combinational UDP

- In combinational UDPs, the output state is determined solely as a function of the current input states. Whenever an input changes state, the UDP is evaluated, and one of the state table rows is matched. The output state is set to the value indicated by that row. The maximum number of inputs to a Combinational UDP is 10.

```
// Output should always be the first signal in the port list

primitive mux (out, sel, a, b);
   output out;
   input sel, a, b;

   table
         //  sel   a   b          out
         0   1   ?   :   1;
         0   0   ?   :   0;
         1   ?   0   :   0;
         1   ?   1   :   1;
         x   0   0   :   0;
         x   1   1   :   1;
   endtable

endprimitive
```

A ? indicates that the signal can be either 0, 1 or x and does not matter in deciding the final output.

Sequential UDP

- Sequential UDP allows the mixing of the level-sensitive and edge-sensitive constructs in the same description. The output port should also be declared as ***reg*** type within the UDP definition and can be optionally initialized within an ***initial*** statement.

1. Level-Sensitive UDPs

- Level-sensitive sequential behavior is represented the same way as combinational behavior, except that the output is declared to be ***reg*** type, and there is an additional field in each table entry.

```
primitive d_latch (q, clk, d);
   output  q;
   input   clk, d;
   reg     q;

   table
           // clk     d        q   q+
         1   1   :   ? : 1;
         1   0   :   ? : 0;
         0   ?   :   ? : -;
   endtable

endprimitive
```

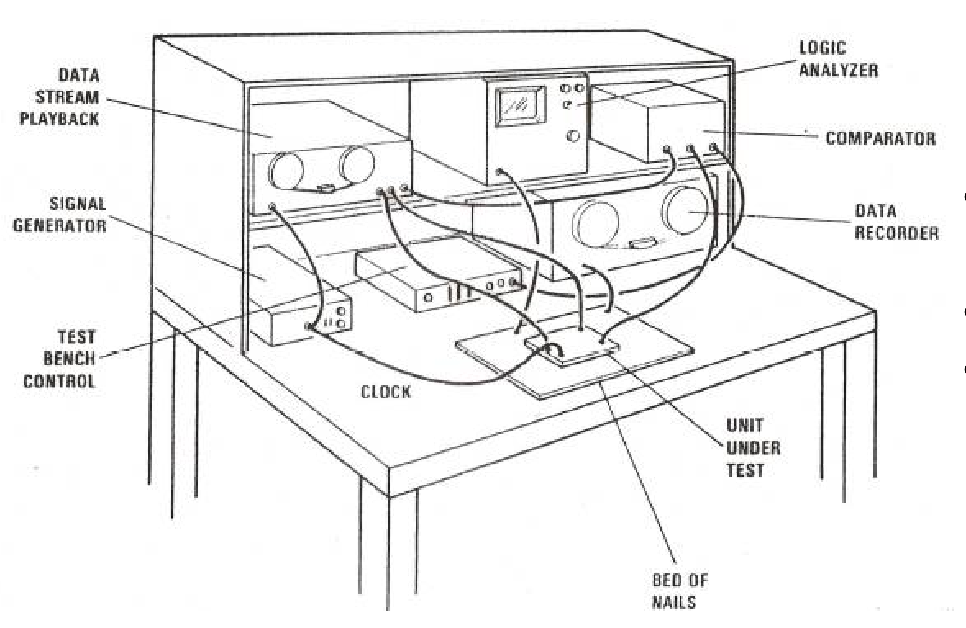
In the above code, a hyphen "-" on the last row of the table indicates no change in value for q+

2. Edge-Sensitive UDPs

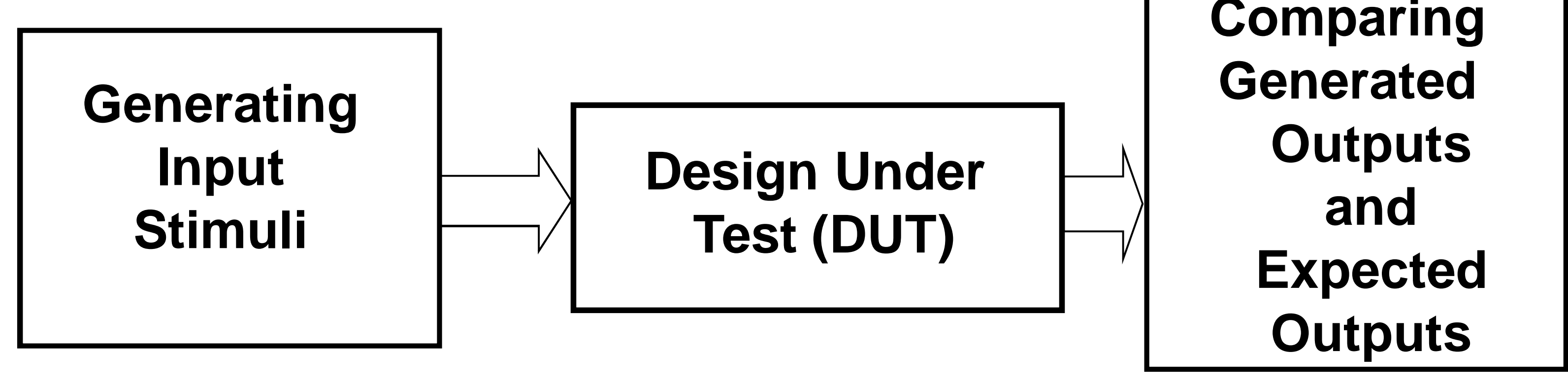
- In level-sensitive behavior, the inputs and the current state's values are sufficient to determine the output value.
- Edge sensitive behavior differs in that changes in the output are triggered by specific transitions of the inputs.

```
primitive d_flop (q, clk, d);
  output  q;
  input   clk, d;
  reg     q;
   table
                    // clk       d       q      q+
        // obtain output on rising edge of clk
        (01)   0  :  ?  :  0;
        (01)   1  :  ?  :  1;
        (0?)   1  :  1  :  1;
        (0?)   0  :  0  :  0;
         // ignore negative edge of clk
        (?0)   ?  :  ?  :  -;
         // ignore data changes on steady clk
        ?      (??):  ?  :  -;
  endtable
 endprimitive
```

Test Bench



- Generate stimulus for testing the hardware block.
- Apply the stimulus.
- Compare the generated outputs against the expected outputs.



How Do You Know That A Circuit Works?

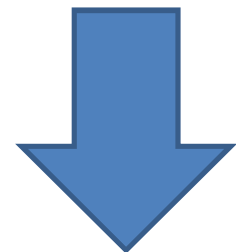
- **You have written the Verilog code of a circuit**
  - Does it work correctly?
  - Even if the syntax is correct, it might do what you want?
  - What exactly it is that you want anyway?
- **Trial and error can be costly**
  - You need to 'test' your circuit in advance
- **In modern digital designs, functional verification is the most time consuming design stage.**

Code for TestBench by example

Write Verilog code to implement the following function in hardware:

$$y = (\overline{b} \cdot \overline{c}) + (a \cdot \overline{b})$$

```
module sillyfunction(input  a, b, c,
                     output y);

  assign y = ~b & ~c | a & ~b;
endmodule
```



Simple Testbench

```
module testbench1(); // Testbench has no inputs, outputs
  reg  a, b, c;      // Will be assigned in initial block
  wire y;

  // instantiate device under test
  sillyfunction dut (.a(a), .b(b), .c(c), .y(y) );d

  // apply inputs one at a time
  initial begin                  // sequential block
    a = 0; b = 0; c = 0; #10; // apply inputs, wait 10ns
    c = 1; #10;                  // apply inputs, wait 10ns
    b = 1; c = 0; #10;           // etc .. etc..
    c = 1; #10;
    a = 1; b = 0; c = 0; #10;
  end
endmodule
```

Summary

- Simple testbench instantiates the design under test
- It applies a series of inputs
- The outputs have to be observed and compared using a simulator program.
  - This type of testbench does not help with the outputs
- `initial` statement is similar to `always`, it just starts once at the beginning, and does not repeat.
- The statements have to be blocking.