# Chapter 3 : Verilog Fundamentals

- Verilog HDL
- Language Basics

## What is an HDL?

- A Hardware Description Language (HDL) is a software programming language used to model the intended operation of a piece of hardware.
- The difference between an HDL and “C”
  - Concurrency
  - Timing
- A powerful feature of the Verilog HDL is that we can use the same language for describing, testing and debugging the system.

## History of Verilog

- Verilog was developed by Gateway Design Automation as a simulation language in 1983. Cadence purchased Gateway in 1989 and placed the Verilog language in the public domain. Open Verilog International (OVI) was created to develop the Verilog language as an IEEE standard. The definitive reference guide to the Verilog language is now the Verilog LRM, IEEE Std 1364-1995 [1995].
- Verilog is a fairly simple language to learn, especially if you are familiar with the C programming language.

## Verilog Coding

Basic Verilog Concepts

Code Structure

## Basic Verilog Concepts

Comments
Identifiers
Logic Values
Data Types
Numbers
Strings

## Verilog Comments

- **Single line comments:**
  - **Begin with "//" and end with a carriage return**
  - **May begin anywhere on the line.**
- **Multiple line comments:**
  - **Begin with "/*" and end with a "*/"**
  - **May begin and end anywhere on the line**
  - **Everything in between is commented out**
- Coding style tip - Use single line comments for comments. Reserve multi-line comments for commenting out a section of code.

## An Example

```
module pound_one;
reg [7:0] a,a$b,b,c; // register declarations
reg clk;

initial
 begin
       clk=0;  // initialize the clock
       c = 1;
       forever  #25 clk = !clk;
 end
/*  This section of code implements
     a pipeline */
always @ (posedge clk)
 begin
       a = b;
       b = c;
 end
endmodule
```

## Identifiers

- Identifiers are names assigned by the user to Verilog objects such as modules, variables, tasks etc.
- An identifier may contain any sequence of letters, digits, a dollar sign '$' , and the underscore '_' symbol.
- The first character of an identifier must be a letter or underscore; it cannot be a dollar sign '$' , for example. We cannot use characters such as '-' (hyphen), brackets, or '#' in Verilog names (escaped identifiers are an exception).

## Escaped Identifiers

- The use of escaped identifiers allow any character to be used in an identifier.
  - Escaped identifiers start with a backslash (\) and end with white space (White space characters are space, tabs, carriage returns).
  - Gate level netlists generated by EDA tools (like DC) often have escaped identifiers
- Examples:
  - \/clock = 0;
  - \a*b = 0;
  - \5-6
  - \bus_a[0]
  - \bus_a[1]

```
module identifiers; /* Multiline comments in Verilog   look like C comments
and // is OK in here. */
// Single-line comment in Verilog.
  reg legal_identifier, two__underscores;
  reg _OK,OK_,OK_$,OK_123,CASE_SENSITIVE, case_sensitive;
  reg \/clock ,\a*b ; // Add white_space after escaped identifier.
   //reg $_BAD,123_BAD; // Bad names even if we declare them!
  initial begin
        legal_identifier = 0; // Embedded underscores are OK,
        two__underscores = 0; // even two underscores in a row.
        _OK = 0; // Identifiers can start with underscore
        OK_ = 0; // and end with underscore.
        OK$ = 0; // $ sign is OK.
        OK_123 =0; // Embedded digits are OK.
        CASE_SENSITIVE = 0; // Verilog is case-sensitive (unlike VHDL).
        case_sensitive = 1;
        \/clock = 0; // An escaped identifier with \ breaks rules
        \a*b = 0; // but be careful to watch the spaces!
        $display("Variable CASE_SENSITIVE= %d",CASE_SENSITIVE);
        $display("Variable case_sensitive= %d",case_sensitive);
        $display("Variable \/clock = %d",\/clock );
        $display("Variable \\a*b = %d",\a*b );
   end
endmodule
```

**An Example**

## Simulation Result of the Example

```
Variable CASE_SENSITIVE= 0
Variable case_sensitive= 1
Variable /clock = 0
Variable \a*b = 0
```

## Logic values

- **Verilog has 4 logic Values:**
  - **‘0’ represents zero, low, false, not asserted.**
  - **‘1’ represents one, high, true, asserted.**
  - **‘z’ or ‘Z’ represent a high-impedance value, which is usually treated as an 'x' value.**
  - **‘x’ or ‘X’ represent an uninitialized or an unknown logic value--an unknown value is either '1' , '0' , 'z' , or a value that is in a state of change.**

## Data Types

- **Three data type classes:**
  - **Nets**
    - **Physical connections between devices**
  - **Registers**
    - **Storage devices, variables.**
  - **Parameters**
    - **Constants**

## Nets

- Most common Net types
  - **wire** and **tri** (which are identical);
  - **supply1** and **supply0** (which are equivalent to the positive and negative power supplies respectively).
- The wire data type is analogous to a wire in an ASIC. A wire cannot store or hold a value. A wire must be continuously driven by an assignment statement. The default initial value for a wire is 'z'
- Coding style tip - Use "tri" instead of "wire" as a visual indicator for more than one driver on a net.
- Other net types: wand, wor (synthesizable); trior, triand, trireg, tri1, tri0(not synthesizable).

```
example:
     wire a,b;                          // scalar wires
```

## Registers

- A register data type is declared using the keyword **reg** and is comparable to a variable in a programming language.
- A storage device. But a **reg** is not always equivalent to a hardware register, flip-flop, or latch.
- On the LHS of an assignment a register data type is updated immediately and holds its value until changed again.
- The default initial value for a reg is 'x' .

```
reg a;                          // scalar reg variable
reg [7:0] in_bus;               // vectored reg variable
```

## Parameters

- **parameters:**
  - **run-time constant**
  - **used anywhere a literal may**
  - **for synthesis, must be integer and must be defined before being used**

```
syntax:
parameter <[msb:lsb]> identifier = value <, identifier = value ...> ;
```

```
examples:
     parameter [2:0] a = 1;          // 3-bit
     parameter
        depth = 32,                  // default depth
        width = 8;                   //default width
```

## Numbers

- Constant numbers are integer or real constants .
- Integers may be sized or unsized.
  - Syntax: <size>'<base><value>
    where:
    - <size> is the number of bits
    - <base is b or B (binary), o or O (octal), d or D (decimal), h or H (hex)
    - <value> is 0-9 a-f A-F x X z Z ? _
    - Examples: 2'b01, 6'o243, 78, 4'ha,
- Default radix is decimal, i.e. 1=1'd1
- underscores ( _ ) are ignored (use them as you would commas), e.g. 836_234_408_566_343
- a "?" is interpreted as Z (high impedance), 2'b??=2'bzz
- When <size> is less than <value> - the upper bits are truncated, e.g. 2'b101->2'b01, 4'hfcba->4'ha

## Points to Note

- When <size> is greater than <value>, and the left-most bit of <value> is 0 or 1, then zero's are extended to <size> bits.
  - 4'b01 -> 4'b0001, 16'h0 -> 16'h0000
  - 4'b11 -> 4'b0011, 16'h1 -> 16'h0001
- When <size> is greater than <value>, and the left-most bit of <value> is an x then the x value is extended to <size> bits
  - 4'bx1 -> 4'bxxx1, 16'hx -> 16'hxxxx
- When <size> is greater than <value>, and the left-most bit of <value> is a z then the z value is extended to <size> bits
  - 4'bz1 -> 4'bzzz1, 16'hz ->16'hzzzz
- Real numbers may be either in decimal or scientific notation
  - Syntax: <value>.<value> or <mantissa>e<exp>
    - 6.439 or 5.3e6

## Examples

| | |
|---|---|
| • 3.14 | decimal notation |
| • 6.4e3 | scientific notation for 6400.0 |
| • 16'bz | 16 bit z (z is extended to 16 bits) |
| • 83 | unsized decimal |
| • 8'h0 | 8 bits with 0 extended to 8 bits |
| • 2'ha5 | 2 bits with upper 6 bits truncated (binary equivalent = 01) |
| • 2_000_000 | 2 million |
| • 16'h0x0z | 16'b0000xxxx0000zzzz |

- Coding style tip - don't use " ? " in a number to indicate high impedance. It only adds confusion. If you want high impedance use " z "!!

## Strings

- Strings are enclosed in double quotes and are specified on one line.
- Verilog recognizes normal C escape Characters (\t, \n, \\, \",%%).

**examples:**
**parameter** A_String = "abc";
// string constant, must be on one line
**parameter** Say = "Say \"Hey!\"";
// use escape quote \" for an embedded quote
**parameter** Tab = "\t"; // tab character

## Code Structure

Design Entities

Verilog Module Basics

## Design Entities

- The **module** is the basic unit of code in the Verilog language.
- Example

```
module holiday_1(sat, sun,weekend);
  input sat, sun;
  output weekend;
  assign weekend = sat | sun;
endmodule
```

## Verilog Module

- **Modules contain**
  - **declarations**
  - **functionality**
  - **timing**

```
module name (port_names);
     module port declarations
     data type declarations
     procedural blocks
     continuous assignments
     user defined tasks & functions
     primitive instances
     module instances
     specify blocks
endmodule
```

syntax:

**module *module_name*** *(signal, signal,...  signal ) ;*

**.        ; //content of module**

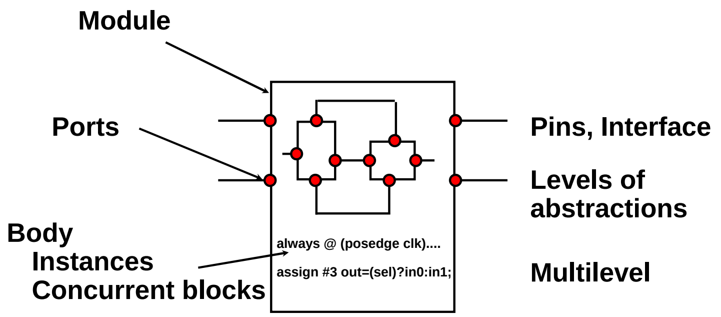
**.**

**..**

**.**

**endmodule**

## Basic Modeling Structure



## Module Port Declarations

- Scalar (1bit) port declarations:
  - *port_direction  port_name, port_name ... ;*
- Vector (Multiple bit) port declarations:
  - *port_direction  [port_size]  port_name, port_name ... ;*
- *port_direction* : input, inout (bi-directional) or output
- *port_name* : legal identifier
- *port_size* : is a range from [msb:lsb]

```
input a, into_here, george;// scalar ports
input [7:0] in_bus, data;     //vectored ports
output [31:0] out_bus;        //vectored port
inout [maxsize-1:0] a_bus;//parameterized port
```

## Using the ports of a module

Synthesizes

```
module incrementer(go, out, clk, rst);
input go,clk,rst;
output [11:0] out;

reg[11:0] out;

always @ (posedge clk or negedge rst)
  if (!rst)
     out = 12'b0;
  else if (go)
     out = out +1;

endmodule
```

**Driving an output port**
If an **output** port and a **reg** variable share the same name, Verilog infers a **wire** of that name connecting the two. This is called an implicit wire. So, the port reflects all changes in the reg value.

**Reading an input port**
An input port can be used directly in the code as Verilog infers a wire by that name also

## Port Connection Rules

- **Inputs:**
  - **Internally must be of net data type.**
  - **Externally the inputs may be connected to a reg or net data type.**
- **Inouts**
  - **Internally must be of net data type.**
  - **Externally must be connected to a net data type.**
- **Outputs**
  - **Internally may be of net or reg data type.**
  - **Externally must be connected to a net data type.**

## Module Instances

- **A module may be instantiated within another module.**
- **There may be multiple instances of the same module.**
- **Ports are either by order or by name.**
- **Use by order unless there are lots of ports**
- **Use by name for libraries and other peoples code**
- **Can not mix the two syntax's in one instantiation**

syntax for instantiation with port order:
***module_name instance_name*  (signal, signal,...);**

syntax for instantiation with port name:
***module_name instance_name*  (.port_name(signal), .port_name (signal),...** );

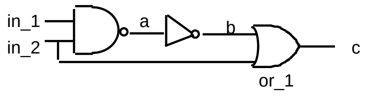
```
module example (a,b,c,d);
input a,b;
output c,d;
. . . .
endmodule

example ex_inst_1(in_1, in_2, w, z);
example ex_inst_2(in_1, in_2, , z);  // skip a port
example ex_inst_3 (.a(w), .d(x), .c(y), .b(z));
```

## Gate-level Primitives

- Verilog has pre-defined primitives that implement basic logic functions.
- Structural modeling with the primitives is similar to schematic level design.

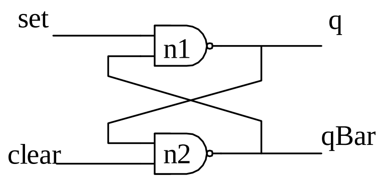
| | | | | | |
|---|---|---|---|---|---|
| **and** | **nand** | **or** | **nor** | **xor** | **xnor** |
| **buf** | **not** | **bufif0** | **bufif1** | **notif0** | **notif1** |

```
module
gate_level_ex(in_1,in_2,c);
output c;
input in_1,in_2;

nand (a, in_1, in_2);
not (b, a);
or or_1(c, in_2, b);

endmodule
```



## An Example

```
Module simple_latch (q, qBar, set, clear);
  input set, clear;
  output q, qBar;
  nand #2 n1(q,qBar,set);
  nand #2 n2(qBar,q,clear);
endmodule
```



## User-Defined Primitives

- We can define primitive gates (a **user-defined primitive** or **UDP**) using a truth-table specification. The first port of a UDP must be an output port, and this must be the only output port (we may not use vector or inout ports).
- An example

```
primitive Adder(Sum, InA, InB);
    output Sum;
    input InA, InB;
    table // inputs : output
    00 : 0;
    01 : 1;
    10 : 1;
    11 : 0;
    endtable
endprimitive
```

## User-Defined Functions

- Similar to functions in other programming languages. Functions are useful to model combinational logic (rather like a subroutine)

syntax:
***function* <[** size or type ***]>*** name_of_function;
input declarations
local variable declarations
statement or statement_group
***endfunction***

- size is optional and is of form [msb:lsb]
- type is optional and is either *integer* or *real*
- Returns the value assigned to the name of the function.
- Functions may *not* contain timing controls (incl. non-blocking procedural assignments).
- Functions must have at least one input.
- Looks local first then global to module for referenced variables.
- Functions may be called
  - within a continuous assignment e.g. assign b = func(a);
  - indirectly within an instantiation e.g. mod U1 (one, func (a, b) );
  - nested within another function

## Function - Example

```
`define FALSE 0
`define TRUE 1
module function_ex (clk);
input clk;
reg r1,r2,r3;

function error;  // the function definition
input[7:0] a,b,c;
  if ((a !=b) && (a !=c))
     error = `FALSE;  // assign value to the name of the function
  else error = `TRUE;
endfunction

always @ (posedge clk)
if (error(r1,r2,r3))  // call of the function
$display ("error in reg compare");

// another example call below
reg d;
always @ (posedge clk)
  d = error(r1,r2,r3);
endmodule
```

- **A function can be called where a value may be placed in your code**

## Operators

- **Verilog operators (in increasing order of precedence)**
  - ?: (conditional)
  - || (logical or)
  - && (logical and)
  - | (bitwise or)
  - ~| (bitwise nor)
  - ^ (bitwise xor)
  - ^~ ~^ (bitwise xnor, equivalence)
  - & (bitwise and)
  - ~& (bitwise nand)
  - == (logical) != (logical) === (case) !== (case)
  - < (lt)
  - <= (lt or equal)
  - > (gt)
  - >= (gt or equal)
  - << (shift left)
  - >> (shift right)
  - + (addition)
  - - (subtraction)
  - * (multiply)
  - / (divide)
  - % (modulus)

## Procedures and Assignments

Procedural Assignment
Continuous Assignment
Control Statement

## Procedures

- A Verilog **procedure** is an **always** or **initial** statement, a task , or a function .
- The statements within a sequential block (statements that appear between a **begin** and an **end** ) that is part of a procedure execute sequentially in the order in which they appear, but the procedure executes concurrently with other procedures.

## Procedural Blocks

- There are two types of procedural blocks:
  - initial blocks - executes only once
  - always blocks - executes in a loop
- Multiple Procedural blocks may be used, if so the multiple blocks are *concurrent*.
- Procedural blocks may have:
  - Timing controls - which delays when a statement may be executed
  - Procedural assignments
  - Programming statements

## Procedural Statement Groups

- When there is more than one statement within a procedural block the statements must be grouped.
- Sequential grouping: statements are enclosed within the keywords **begin** and **end.**
- An example

```
always
  begin
    a = 5;      // executed 1st
    c = 4;      // executed 2nd
    wake_up = 1;  // executed 3rd
  end
```

## Timing Controls (procedural delays)

Ignored

- ***#delay*** - simple delay
  - Delays execution for a specific number of time steps.

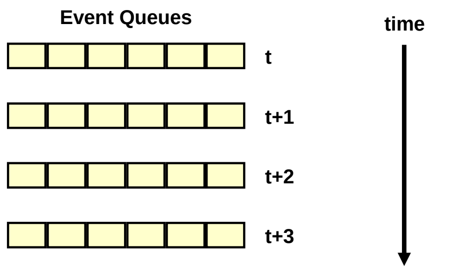
    **#5** reg_a = reg_b;

Synthesizes

- @ ***(edge signal)*** - edge-triggered timing control
  - Delays execution until a transition on ***signal*** occurs.
  - ***edge*** is optional and can be specified as either ***posedge*** or ***negedge.***
  - Several ***signal*** arguments can be specified using the keyword ***or.***
  - An example : always @ (posedge clk) reg_a = reg_b;
- ***wait (expression)*** - level-sensitive timing control
  - Delays execution until ***expression*** evaluates true.
  - **wait (cond_is_true)** reg_a = reg_b;

~~Synthesizes~~

## Time & Event Queues



- **Time can only advance forward.**
- **Time advances when every event scheduled at that time step is executed.**
- **Simulation completes when all event queues are empty**
- **An event at time t may schedule another event at time t or any other time t+n**

## Procedural assignments

- Assignments made within procedural blocks are called procedural assignments.
  - Value of the RHS of the equal sign is transferred to the LHS
  - LHS must be a register data type (reg, integer, real). NO NETS!
  - RHS may be any valid expression or signal

```
always @ (posedge clk)
    begin
        a = 5;          // procedural assignment
        c = 4*32/6;  // procedural assignment
        wake_up =$time;  // procedural assignment
    end
```

## Blocking Assignments

- Blocking assignments.
  - RHS expression evaluated and assignment is scheduled.
- Delayed Blocking assignments.
  - Evaluation of the assignment is delayed by the timing control.
  - RHS expression evaluated and assignment is scheduled.

```
Blocking assignment:
initial
        begin
            a = b;
            c = d;
        end
```

```
Delayed Blocking assignments:
initial
        begin
            #1 a = b;
            #1 c = d;
        end
```

## Blocking Assignments Example

- **RHS expression evaluated.**
- **Assignment is scheduled in sequence.**

```
initial
        begin
            a = b;
            c = d;
            e = f;
        end
```

| Event Queues | | | | Time |
|---|---|---|---|---|
| | e<-f(t) | c<-d(t) | a<-b(t) | t |
| | | | | t+1 |
| | | | | t+2 |
| | | | | t+3 |

<-- Execution order

## Non Blocking Assignments

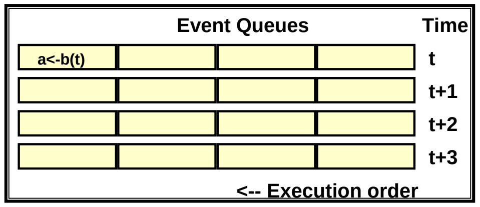
- The **nonblocking procedural assignment statement** allows execution in a sequential block to continue and registers are all updated together at the end of the current time step.
  - **RHS expression evaluated.**
  - **Assignment is scheduled at the end of the queue .**
  - **Assignment is made at end of the time step.**

```
initial
        begin
            a <= b;
        end
```

## Non-blocking Assignments Example

- RHS expression evaluated.
- Assignment is scheduled at the end of the queue .

```
initial
        begin
             a <= b;
        end
```



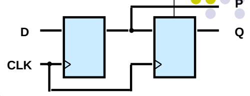
## Assignments and Synthesis (1)

*Synthesis Tip!*

```
module two_stage(Q, D, CLK);
input D, CLK;
output Q;

reg Q, P;

always @ (posedge CLK)
  begin
    Q = P;
    P = D;
 end
```

Q1. Does this <u>simulate</u> a pipe ?
A1. _____

Q2. Which does it synthesize into?
A2. ___________

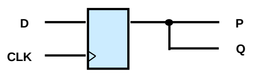


is_a_pipe

```
module two_stage(Q, D, CLK);
input D, CLK;
output Q;

reg Q, P;

always @ (posedge CLK)
  begin
    P = D;
    Q = P;
end
```

Q3. Does this <u>simulate</u> a pipe ?
A3. _____

Q4. Which does it synthesize into?
A4. ___________



not_a_pipe

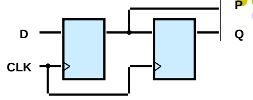
Conclusion: Blocking assignments are order dependent!
( for both simulation <u>and</u> synthesis )

## Assignments and Synthesis (2)

*Synthesis Tip!*

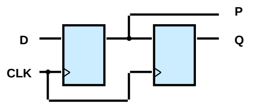
```
module two_stage(Q, D, CLK);
input D, CLK;
output Q;

reg Q, P;

always @ (posedge CLK)
  begin
    Q <= P;
    P <= D;
 end
```

Synthesizes into this...



```
module two_stage(Q, D, CLK);
input D, CLK;
output Q;

reg Q, P;

always @ (posedge CLK)
  begin
    P <= D;
    Q <= P;
end
```

Synthesizes into this...

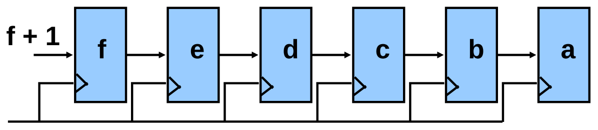


Conclusion: Non-blocking assignments are order <u>independent</u>!

## An Example

```
module pound;

reg [7:0] a,b,c,d,e,f;
reg clk;
initial
 begin
       clk=0;
       f = 1;
       forever
              #25 clk = !clk;
 end
/*** group 1 ***/
always @ (posedge clk) // group 1
 begin
       e = f;
 end
/*** group 2 ***/
always @ (posedge clk) // group 2
 begin
       c = d;
       d = e;
 end
/*** group 3 ***/
always @ (posedge clk) // group 3
 begin
        a = b;
        b = c;
 end
/*** group 4 ***/
always @ (posedge clk) // group 4
 begin
        f = f + 1;
 end
initial
$monitor (f,,e,,d,,c,,b,,a);
initial
   #700 $stop;
endmodule
```

## Procedural assignment Example



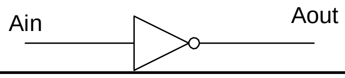
```
Expected output:
f e d c b a
1 x x x x x
2 1 x x x x
3 2 1 x x x
4 3 2 1 x x
5 4 3 2 1 x
6 5 4 3 2 1
7 6 5 4 3 2
and so on
```

## Continuous Assignment

- Continuous assignment assigns a value to a **wire** in a similar way that a real logic gate drives a real wire.
- The main use for continuous assignments is to model combinatorial logic.

**syntax**: Explicit continuous assignment:
***assign net_name = expression;***
where ***net_name*** is a ***net*** that has been previously declared

```
module continuous (Ain, Aout);
        input Ain;
        output Aout;
        assign Aout = ~Ain //continuous assignment.
endmodule
```



## Illustration of Assignment Statements

```
module assignments
   //... Continuous assignments go here.
   always // beginning of a procedure
        begin // beginning of sequential block
        //... Procedural assignments go here.
        end
endmodule
```

## Control Statements

- Two types of programming statements:
  - Conditional
  - Looping
- Programming statements only used in procedural blocks

## if and if-else

**syntax:**
***if(expression) statement***
If the expression evaluates to true then execute the statement (or statement group)

***if(expression) statement1***
***else statement2***
If the expression evaluates to true then execute statement1,
if false, then execute statement2 (or corresponding statement groups).

```
module if_ex(clk);
   input clk;
   reg red,blue,pink,yellow,orange,color,green;
   always @ (posedge clk)
   if (red || (blue && pink))
     begin
       $display ("color is mixed up");
        color <= 0;  // reset the color
     end
   else  if (blue && yellow)
     $display ("color is greenish");
   else  if (yellow && (green || orange))
     $display ("not sure what color is");
   else $display ("color is black");
endmodule
```

## case

**syntax:**
***case (expression)***
***case_item_1: statement or statement_group***
***case_item_2,***
***case_item_3: statement or statement_group***
***case_item_n: statement or statement_group***
***default: statement or statement_group***
***endcase***

- Does an identity comparison (But only simulation will match x, z)
- Compares expression with each case_item_(n) in turn.
- If none match, the default code is executed.
- default clause is ideal to catch unknown/unspecified values

```
reg [2:0] reg_a, reg_b;
always @ (posedge clk)
        case (reg_a)
              3'b000:      reg_b <= 0;
              3'b001:      reg_b <= 1;
              3'b010,
              3'b011:      reg_b <= 3;
              default:  reg_b <= 5;
        endcase
```

## casez, casex

- ***casez*** - special version of case that allows the Z logic value in the case-items (**z** or **?** treated as a don't care).
- ***casex*** - special version of case that allows the Z or X logic value in the case-items (**x** or **z** or **?** treated as don't cares).

```
reg [2:0] reg_a, reg_b;
always @ (posedge clk)
        casex (reg_a)
              3'b000:    reg_b <= 0;
              3'b001:    reg_b <= 1;
              3'b01?:    reg_b <= 2;
              3'b011:    reg_b <= 3;
              3'b1x0:    reg_b <= 4;
              default:   reg_b <= 5;
        endcase
```

**Coding style tip** - to save confusion use " ? " as the don't care indicator.

## Which to use: case or if-else ?

- Some general rules to remember:
  - Use **if-else** where you MUST have priority encoded logic
  - Use **case** for non-priority encoded logic
    - case items are mutually exclusive
    - Always specify a default clause in **case** statements

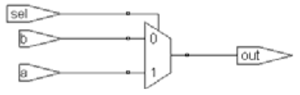
*Synthesis Tip!*

## Inferred latches in Synthesis

Latches can be accidentally inferred from Verilog RTL code

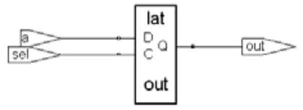
An example:
When using if - else and case all possible states and values must be specified <u>including</u> default or else storage devices are added

The following generates a mux:
```
reg out, sel, a, b;
always @ (sel or a or b)
if(sel)
        out = a;
else out = b;
```



The following <u>infers</u> a latch:
```
always @ (sel or a or b)
if(sel)
        out = a;
```

Can you see why?



*Synthesis Tip!*

## Inferred storage device report

When sequential logic is inferred by HDL Compiler, an *inference report* is produced.

- Each inferred memory element is identified
- Sequential characteristics are listed

```
===================================================================
| Register name |  Type    |  Width    |     Bus    | AR | AS | SR | SS | ST |
===================================================================
|       out     |  latch   |     8     |      -     |  - |  - |  - |  - |  - |
===================================================================
```

<u>For more information consult:</u>

HDL Compiler for Verilog Reference Manual, available on-line

*Synthesis Tip!*

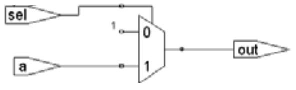
## Avoiding Inferred latches

Specify all case/if structures thoroughly!

-or-

Assign a default value to all outputs before the if/case structure:

```
module default (out, sel, a);
input sel; a;
output out;
reg out, sel, a;
always @(sel or a)
 begin
   out = 1'b1;
   if(sel)    // no else is no problem now!
        out = a;
 end
endmodule
```



## forever

Not recommended for synthesis

**syntax:**
***forever statement or statement_group***

- statement or statement_group is continuously executed.
- An infinite loop.

```
module clock_gen;
reg clk;
initial
        begin
            clk = 0;
            forever #25 clk = !clk; //50 time step clock
        end
endmodule
```

## while

**syntax:**
***while (expression) statement***
***or statement_group***

- statement or statement_group is continuously executed as long as expression evaluates true (or non zero).
- In synthesis, the loop must contain an edge-triggered timing control, i.e. @(posedge clk) or @ (negedge clk)

```
module while_ex (clk, a,b,c);
input clk;
input [1:0] a,b;
output [1:0] c;
reg [1:0] c;

always
begin
@ (posedge clk)
   while (c < b)
@ (posedge clk)
    c = c + a;
end
endmodule
```

## for

**syntax:**
***for (assignment_init; expression; assignment)***
***statement or statement_group***

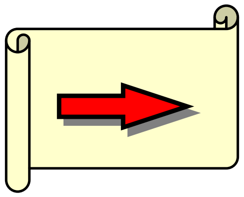
- The ***assignment_init*** is executed once at the start of the loop.
- Loop executes as long as ***expression*** is true.
- The ***assignment*** is executed at the completion of each loop.

```
module for_ex1 (clk);
input clk;
reg [31:0] mem [0:9]; // 10x32 memory
integer i;
always @ (posedge clk)
  for (i = 9; i >= 0; i = i-1)
          mem[i] = 0;  // init the memory to zeros
endmodule
```

## Ever see a hardware **for?**

**HDL Compiler® simply unrolls the loop...**

```
module for_ex2(start_cnt,cnt);
input start_cnt;
output [7:0] cnt;
integer i;
reg [7:0] vec,cnt;
always @ (start_cnt)
 for (i = 0; i <= 3; i = i+1)
          if (vec[i] == 1'b0)
           cnt = cnt + 1;
endmodule
```

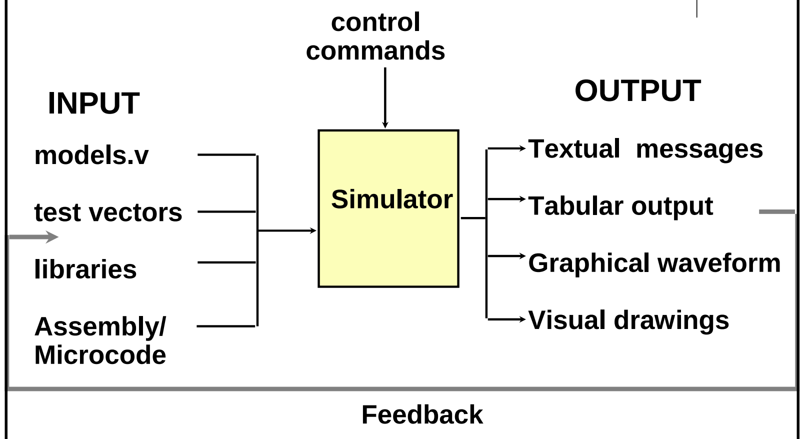


```
module for_ex3(start_cnt,cnt);
input start_cnt;
output[7:0] cnt;
integer i;
reg [7:0] vec,cnt;
reg start_cnt;
always @ (start_cnt)
begin
 if (vec[0] == 1'b0)
   cnt = cnt + 1;
 if (vec[1] == 1'b0)
   cnt = cnt + 1;
 if (vec[2] == 1'b0)
   cnt = cnt + 1;
 if (vec[3] == 1'b0)
   cnt = cnt + 1;
end
endmodule
```

- You can't re-assign the loop variable from within the for loop. It's supposed to be a constant!
- Beware using complex functions inside a for-loop. They can easily be replicated unnecessarily by the unrolling. The example here generates 4 adders!
- For synthesis you can't embed edge-triggered timing controls in **for loops**
- Must use constants in expression limit.

## Simulation

## Simulation Environment



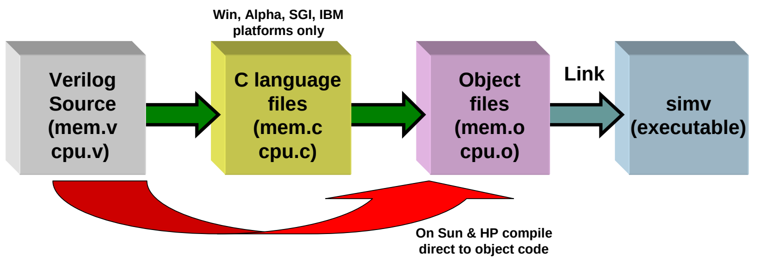
## VCS

VCS stands for Verilog Compiled Simulator

There are two different types of Verilog event based simulators:

- Interpreted - example is Verilog-XL ®
- Compiled - VCS

Depending upon platform, VCS first generates C code from the Verilog source, then it compiles and links the object files to the simulation engine to create an executable.


Win, Alpha, SGI, IBM platforms only
Verilog Source (mem.v cpu.v)
C language files (mem.c cpu.c)
Object files (mem.o cpu.o)
Link
simv (executable)
On Sun & HP compile direct to object code

## Running the Simulator - VCS

| | |
|---|---|
| vcs -help | lists compile options, runtime options, environment variables |

Basic invocation:

| | |
|---|---|
| vcs file1.v file2.v file3.v filen.v | |
| simv | Verilog executable generated by VCS |

Command line options (commonly used)

| | |
|---|---|
| -I | Compile for interactive simulation |
| -Mupdate | Incremental compilation (only changed files are compiled) |
| -R | Run after compilation |
| -RI | Run interactively (with GUI) after compilation |
| -line | Enable line, next and trace commands |
| -f <filename> | read host command arguments from file |
| -l <filename> | set log file name |
| -s | Stop simulation before it begins; enter interactive mode |

## System tasks & functions

Specific tasks and functions may be defined by EDA vendors and users to be used as part of the simulator.

- Begin with the dollar sign ( $ )
- The Verilog standard has a number of standard $ defined
- Users may define their own built in tasks using the Programming Language Interface (PLI)

List of most commonly used built in tasks and functions:

| | |
|---|---|
| **$monitor** | Continuously monitors listed signals |
| **$display** | Prints message to the screen |
| **$time** | function that returns the current simulation time (64-bits) |
| **$stime** | like above, but returns truncated lower 32-bits |
| **$stop** | Halts execution but does not exit |
| **$finish** | Halts execution and exits the simulation |

## Compiler directives

**Compiler directives cause the Verilog compiler to take special actions**

- **Indicated by the grave accent character ( ` )**
- **Directive remains in effect until it is overridden or modified. It is active across modules and files.**

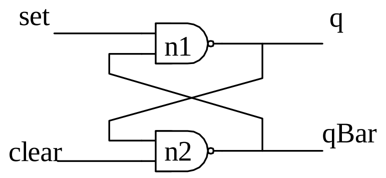
**List of most commonly used compiler directives:**

| | |
|---|---|
| **`define** *macro text_string* | **text substitution of *text_string* for *macro*** |
| **`include** *"file_name"* | **file inclusion. Another source file is substituted here** |
| **`ifdef** *macro*<br>*verilog source*<br>**`else**<br>*verilog source*<br>**`endif** | **Conditional Compilation** |

## Simulating the Verilog Code

- Verilog code of NAND Latch

```
Module simple_latch (q, qBar, set, clear);
   input set, clear;
   output q, qBar;
   nand #2 n1(q,qBar,set);
   nand #2 n2(qBar,q,clear);
endmodule
```



## Testbench

- A testbench generates a sequence of input values (we call these **input vectors** ) that test or **exercise** the verilog code.
- It provides stimulus to the statement that will monitor the changes in their outputs.
- Testbenchs do not have a port declaration but must have an instantiation of the circuit to be tested.

## A testbench for NAND Latch

```
Module test_simple_latch;
   wire q, qBar;
   reg set, clear;
   simple_latch SL1(q,qBar,set,clear);
   initial
         begin
          #10 set = 0; clear = 1;
          #10 set = 1;
          #10 clear = 0;
          #10 clear = 1;
          #10 $stop;
          #10 $finish;
         end
   initial
         begin
          $monitor ("%d set= %b clear= %b q=%b qBar=%b",$time,
                   set,clear,q,qBar);
         end
endmodule
```