# Verilog Basics

**Teemu Pitkänen**
**Teemu.pitkanen@tut.fi**
**TH318**
**(03) 3115 4778**

# Outline

- Modules
- Logic level modelling
- Design hierarchy
- Behavioural modelling
- Concurrent processes
- Parameters
- Switch level modelling
- ...

- Source: The Verilog Hardware Description Language, 2nd Edition. D. E. Thomas and P. R. Moorby. 1995.

# Introduction

- Verilog was developed around 1983 at Gateway Design Automation (later a part of Cadence) by Phil Moorby.
- Was first used in a simulator.
- Language was opened to the public in 1980.
- Much like C
- Case sensitive
- Key elements:
  - *modules*
  - behavioural modelling
  - continuous assignments
  - hierarchy
  - component instantiation
  - ...

# Module

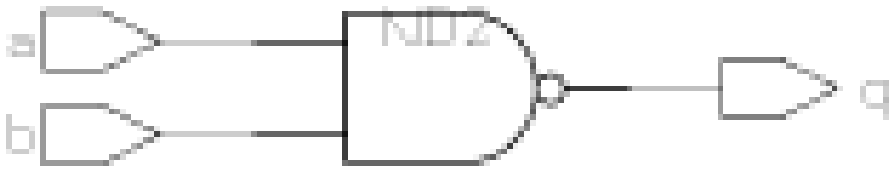
- **Verilog describes a digital system as a set of *modules*.**
- **Each module has an interface and a description off its contents.**
- **There can be an module without inputs or outputs.**

```
module <module_name>(<port_names>);
endmodule
```

- **Example:**

```
module Nand(q, a, b);
  output q;
  input a, b;

  nand (q, a, b);
endmodule
```



# Gate Level Modelling

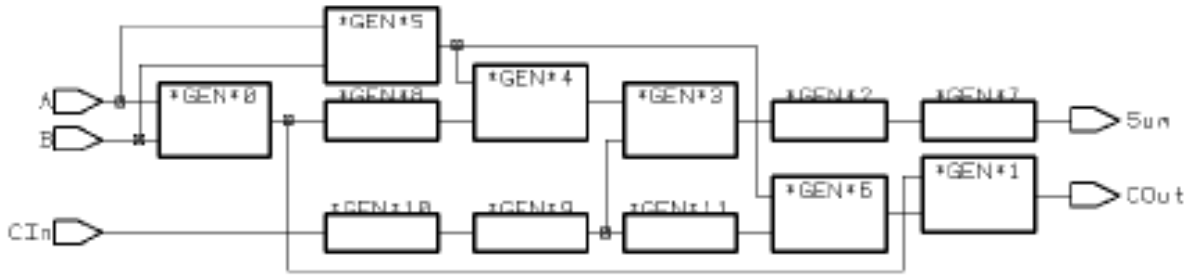
- Describes the circuit in terms of interconnections of logic primitives (ANDs, ORs, ...).
- Verilog provides *gate level primitives* for the standard logic functions.
  - *and, nand, nor, or, xor, xnor* (standard logic functions)
  - *buf* (buffer)
  - *not* (inverter)
  - *bufif0, bufif1, notif0, notif1* (buf and not with a tristate enable input)
  - *nmos, pmos, cmos, rnmos, rpmos, rcmos, tran, tranif0, tanif1, rtran, rtranif0, rtranif1, pullup, pulldown* (transistor switch level models)
- Standard logic functions has a single output or bidirectional port and any number of inputs.
- The output is the first one in the gate instantiations.

# Gate Level Modelling (2)

- ***Buf*** **and** ***not*** **may have any number of outputs. The only inout is listed last.**
- **Example:**

```
module GateLevel(COut, Sum, A, B, CIn);
    output COut, Sum;
    input A, B, CIn;
    wire x2;

    nand   (x2, A, B),
           (COut, x2, x8);
    xnor   (x9, x5, x6);
    nor    (x5, x1, x3),
           (x1, A, B);
    or     (x8, x1, x7);
    not    (Sum, x9),
           (x3, x2),
           (x6, x4),
           (x4, CIn),
           (x7, x6);
endmodule
```



# Gate Level Modelling (3)

- **The general syntax for gate instantiation:**

```
<GATETYPE> <drive_strength> <delay> <gate_instance>,
<gate_instance>;
```

- **where**
  - **<GATETYPE> is one the gate level primitives**
  - **<gate_instance> ::= <name_of_gate_instance> (<terminal>, <terminal> )**
  - **<name_of_gate_instance> ::= identifier**

# Gate Level Modelling (4)

- **Example:**

```
nand    Name1(Out1, AIn, BIn),
        OtherName(COut, x12, x1);
```

- **Nand gates with drive strenght and simulation delay:**

```
nand (strong0, strong1) #3
        Name1(Out1, AIn, BIn),
        OtherName(COut, x12, x1);
```

- **When drive strength and/or delay is given, it applies to all the defined instances in the comma-separated list.**
- **To change one or both these qualifiers, the gate instantiation must be ended (with a ";") and restarted.**

# Nets

- Nets do not store values.
- Net of type wire which have a delay:

```
wire #3 x2;
```

- # indicates delay.
- Delay could include both rise fall time specifications.
- Example transition to 1 has a delay of 3 units and the fall to 0 has a delay of 5:

```
wire #(3,5) x2;
```

- Nets can also be declared implicit!
  - when modules are connected with nets

# Nets (2)

- **Other net types than wires can be used.**

| Net Type | Modeling Usage |
|---|---|
| wire, tri | Used to model connections with no logic function.<br>Only difference is the name. |
| wand, wor, triand, trior | Used to model the wired logic functios. |
| tri0, tri1 | Used to model connections with a resistive pull to given supply. |
| supply0, supply1 | Used to model the connection to a power supply. |
| trireg | Used to model charge storage on a net. |

- **Example, AND-port and wired-AND (wand):**
  - **AND-port treats a z on its inputs as an X.**
  - **wand will pass z on its input**

# Registers

- Registers are abstractions of storage devices found in real system.
- Registers are defined with the reg keyword.
- Size is optionally given. The default size is one.
- Example:

```
reg TempBit;
```

  - defines a single bit register named TempBit

```
reg [15:0] TempNumber;
```

  - defines a 16-bit register

```
reg [0:15] TempNumber2;
```

  - defines also a 16-bit register

# Registers (2)

- **Both bit-select and part-select can be used.**

```
reg [11:0] counter;
reg        a;
reg [2:0]  b;

a = counter[7]; // bit seven is loaded into a
b = counter[4:2] // bits 4, 3, and 2 are loaded into b
```

- **Notice the comment style: // starts a comment**

# Port Specifications

- An input port specifies the internal name for a vector or scalar that is driven by external entity.
- An output port specifies the internal name for a vector or scalar which is driven by an internal entity is available external to the module.
- An inout port specifies the internal name for a vector or scalar that can be driven either by an internal or external entity.
- Input or inout port cannot be declared to be of type register.
- These port types may be read into register using a procedural assignment, used on the right-hand side of a continuous assignment, or used as input to instantiated modules or gates.

# Port Specifications (2)

- **A module's ports are normally connected at the in the order in which they are defined.**
- **Connection can be done also by naming the port and giving its connection.**
- **Example:**

```
module AndOfComplements(a, b, c, d);
  input a, b;
  output c, d;
  wand c;
  wand d;

  not(c, a);
  not(c, b);
  not(d, a);
  not(d, b);
endmodule
```

# Port Specifications (3)

```
module Ace;
   wire r, t;
   reg q, s;
   AndOfComplements m1(.b(s), .a(q), .c(r), .d(t));
endmodule
```

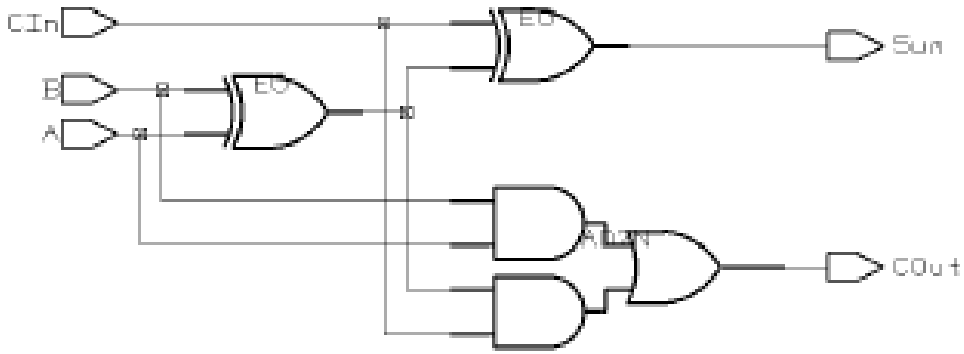
- **Port b of instance m1 of module AndOfComplements will be connected to the output of register s, port a to the output of register q, and so on.**
- **The connections may be listed in any order.**

# Continuous Assignment

- **Abstractly model combinational HW driving values onto nets.**
- **assign statement**

```
module GateLevel2(COut, Sum, A, B, CIn);
   output COut, Sum;
   input A, B, CIn;

   assign Sum = A ^ B ^ CIn,
          COut = (A & B) | (B & CIn) | (A & CIn);
endmodule
```



# Continuous Assignment (2)

- **Continuous assignment is always active. If any input to the assign statement changes at any time, the assign statement will be re-evaluated and the output will be propagated.**
- **The general form of the assign statement is:**

```
assign <drive_strength> <delay> <list_of_assignments>;
```

- **drive_strength and delay are optional**
- **Example:**

```
assign #5
  Sum = A ^ B ^ CIn,
  COut = (A & B) | (B & CIn) | (A & CIn);
```

- **The final gate level implementation is left to a synthesis program.**

# Logical Operators

| Operation | | |
|---|---|---|
| ~ | bitwise negation | Complements each bit in the operand |
| & | bitwise AND | Produces the bitwise AND of two operands |
| ~& | bitwise NAND | |
| \| | bitwise OR | |
| ~\| | bitwise NOR | |
| ^ | bitwise XOR | |
| ^~ or ~^ | equivalence | Produces the bitwise exclusive NOR |
| & | unary reduction AND | Produces the single bit AND of all of the bits of the operand |
| ~& | unary reduction NAND | |
| \| | unary reduction OR | |

# Logical Operators (2)

| Operation | | |
|---|---|---|
| ~\| | unary reduction NOR | Produces the single bit NOR f all of the bits of the operand |
| ^ | unary reduction XOR | |
| ~^ or ^~ | unary reduction XNOR | |

- **Unary reduction and binary bitwise operators are distinguished by syntax.**

# Logical Operators (3)

**Table 1: Bitwise AND**

| & | 0 | 1 | x |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | x |
| x | 0 | x | x |

**Table 2: Bitwise XNOR**

| ~^ | 0 | 1 | x |
|---|---|---|---|
| 0 | 1 | 0 | x |
| 1 | 0 | 1 | x |
| x | x | x | x |

# Operator Precedence

highest precedence: ! & ~& | ~| ^ ~^ + - (unary operators)

* / %

+ -

<< >>

< <= > >=

== != === !==

& ~& ^ ~^

| ~|

&&

||

lowest precedence: ?:

# Behavioural Modelling of Combinational Circuits

- **The right-hand side expression in the assign statement may contain a function call.**
- **Within a function, procedural statements such as case and looping statements may occur. However, function may not contain control statements such as event checks. => no sequential structures**
- **Example:**

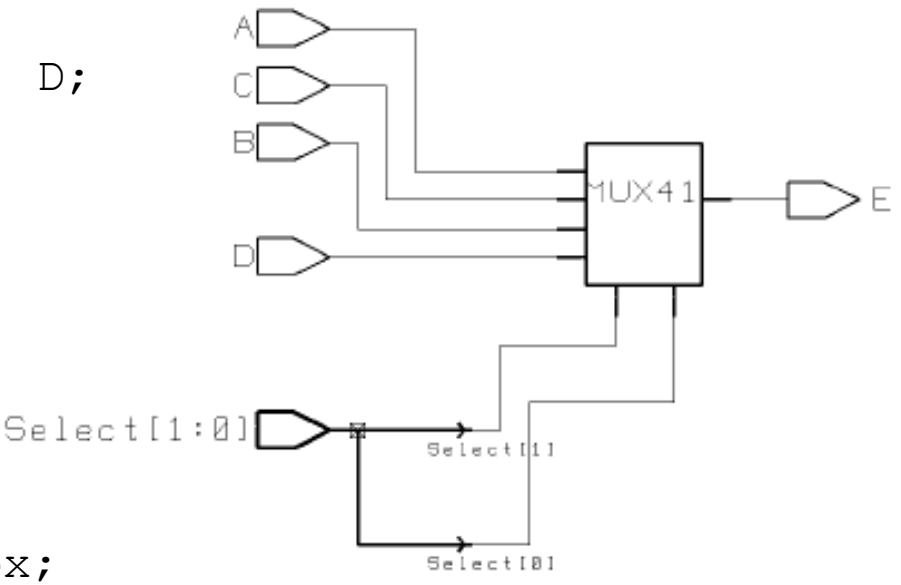
```
module Multiplexor(A, B, C, D, Select, E);
  input       A, B, C, D;
  input [1:0]Select;
  output      E;

  assign E = mux(A, B, C, D, Select);
```

# Behavioural Modelling of Combinational Circuits (2)

- **Example continues...**

```
function mux;
  input       A, B, C, D;
  input [1:0]Select;
    case(Select)
      2'b00: mux = A;
      2'b01: mux = B;
      2'b10: mux = C;
      2'b11: mux = D;
      default: mux ='bx;
  endcase
endfunction
endmodule
```



# Net and Continuous Assignment

- **A shorthand way to specify a value to be driven onto a net is combine the net and assign definition statements.**

```
module ModXor(AXorB, A, B);
  output [7:0] AXorB;
  input  [7:0] A, B;
  wire [7:0] #5 AXorB = A ^ B;
endmodule
```

- **The delay (#5) specifies the delay involved in the XOR-port, not in the wire drivers.**
- **If the wire and XOR had been declared separately, a separate delay could gave assigned to the wire drivers:**

```
wire    [7:0]  #10 AXorB;
assign #5     AXorB = A ^ B;
```

# Parameters

- **Parameters allow us to define generic module.**
- **Keyword parameter**
- **Example:**

```
module XorX(XOut, XIn1, XIn2);
  parameter width = 4,
  delay = 10;
  output [1:width] XOut;
  input  [1:width] XIn1, XIn2;

  assign #delay XOut = XIn1 ^ XIn2;
endmodule
```

- **Instantiation of the parameterized module:**
  - **XorX #(8, 0) x1(D, A, B);**

# Parameters (2)

- The #(8, 0) specifies that the value of the first parameter (width) is 8 for this case, and the value of the second parameter (delay ) is 0.
- If the #(8, 0) was omitted in the instantiation, the values specified in the module definition would be used as a default values.
- The general form is specifying parameter values is:

```
<name_of_module> <parameter_value_assignments>
<module_instance> <module_instance>;
```

- The order of the overriding values follows the order the parameter specification in the module's definition.
- It is not possible to skip over some parameters in a module definition and respecify the rest.

# Parameters (3)

- Another approach to overriding the parameters in a module definition is to use the *defparam* statement.
- Not supported, e.g. by Synopsys synthesis.
- Using that statement, all of the respecifications of parameters can be grouped into one place within the description.

```
module Xors(A1, A2);
  output [3:0] A1, A2;
  reg    [3:0] B1, C1, B2, C2;

  XorX A(A1, B1, C1),
       B(A2, B2, C2);
endmodule

module Annotate;
  defparam
    Xors.B.width = 8;
endmodule
```

# Logic Delay Modelling

- **Example circuit:**

```
module TriStateLatch(QOut, NQOut, Clock, Data, Enable);
  output QOut, NQOut;
  input Clock, Data, Enable;
  tri QOut, NQOut;

  not #5          (NData, Data);
  nand #(3,5)     N1(WA, Data, Clock),
                  N2(WB, NData, Clock);
  nand #(12,15)   N3(Q, NQ, WA),
                  N4(NQ, Q, WB);
  bufif1 #(3, 7, 13) QDrive(QOut, Q, Enable),
                     NQDrive(NQOut, NQ, Enable);
endmodule
```

bufif1

| Input Data | Control Input (enable) | | | |
|---|---|---|---|---|
| | 0 | 1 | x | z |
| 0 | z | 0 | 0/z | 0/z |
| 1 | z | 1 | 1/z | 1/z |
| x | z | x | x | x |
| z | z | x | x | x |

# Logic Delay Modelling (2)

- **Gate delays:**
  - **delays are specified in terms of the transition to 1, the transition to 0, and the transition to z**
  - **the default delay is zero**
  - **e.g., *bufif1 #(3,7,13)* has a rising delay of 3, falling delay of 7, and a delay to the high impedance value of 13**
- **Generally the delay specification takes the form of:**
  - **#(d1, d2, d3), where d1 is the raising delay, d2 the failing delay and d3 the delay to the high impedance value.**
- **If one delay parameter is specified, then the value is used for all delays**

# Verilog Numbers

- Constant numbers can be specified in decimal, hexadecimal, octal, or binary
- A number can be given one of two forms.
- The 1st form is an unsized decimal number specified using the digits from the sequence 0 to 9.
  - Verilog calculates a size and the appropriate number of bits, starting from the least significant bit, are selected for use.
- The 2nd form specifies the size of the constant and takes the form:

```
ss...s 'f nn...n
```

- where:

# Verilog Numbers (2)

`ss...s` is the size in bits of the constant. The size is specified as a decimal number.

`'f` is the base format. The f is replaced by one of the single letters: d (decimal), h (hexadecimal), o (octal), or b (binary). The letters may also be capitalized.

`nn...n` is the value of the constant with allowable digits. For the hexadecimal base, the letters a through f may also be capitalized.

- An underline character may be inserted into a number (of any base) to improve readability. It must not be the first character of a number.

  => *0x0x_1101_ 0zx1* is the same than *0x0x1101 0zx1*

# Verilog Numbers (3)

- **Examples (unsized):**

```
792 // a decimal number
7d9 // ILLEGAL, hexadecimal must be specified with 'h
'h 7d9 // an unsized hexadecimal number
'o 7746 // an unsized octal number
```

- **Examples (sized):**

```
12 'h x // a 12 bit unknown number
10 'd 17 // a 10 bit constant with the value 17
4 'b 110z // a 4 bit binary number
```

# Initial Statement

- The initial statement describes a process.
- Is executed only once.
- The initial statement provides a means of initiating input waveforms and initializing simulation variables before the actual description begins.
- Once the statements in the initial are exhausted, statement becomes inactive

```
module ffNandSim;
  reg preset, clear;
  ...
  initial
    begin
      #10 preset = 0; clear = 1;
      #10 preset = 1;
      #10 clear = 0;
      #10 clear = 1;
    end
endmodule
```

# Always Statement

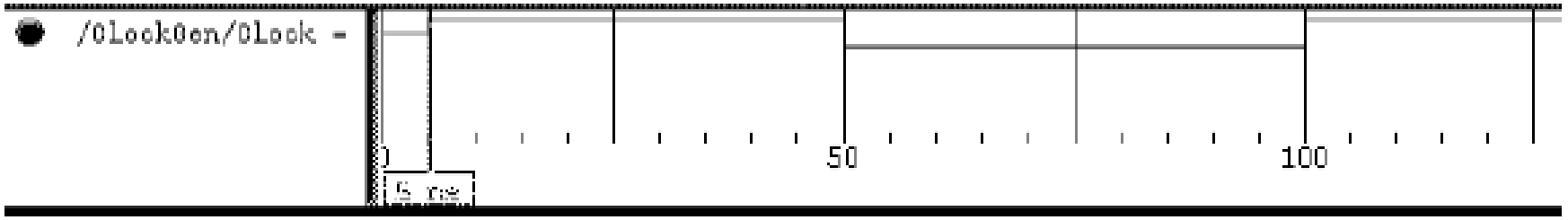
- The basic Verilog statement for a process is the *always* statement.
- It continuously repeats its statement, never exiting or stopping
- A behavioural Verilog model can contain one or more always statements.
- A module without always statement is purely a specification of hierarchical structure.
- Example:

```
module ClockGen(Clock);
  output Clock;
  reg Clock;

  initial
   #5 Clock = 1;
  always
   #50 Clock = ~Clock;
endmodule
```

# Always Statement (2)

- At the start, the output has the value x.
- After 5 time units have passed, the output is initialized to be one.
- After the 1st 50 time units have passed, the always statement executes.



# Edge Sensitive Process

- **Event control construct: @**
- **The event control watch for a situation that is generated by an external process.**
- **Watch for a change in a value -> edge sensitive**

```
module DFlipFlop(Q, Clk, Data);
  output Q;
  reg    Q;
  input  Clk, Data;

  initial Q = 0;
  always @(negedge Clk) Q = Data;
endmodule
```

- **Q will be loaded with the value on the Data when there is a negative edge on the Clk-port.**

# Edge Sensitive Process (2)

- **In addition to specifying a negative edge to trigger on, also a positive edge can be specified:**

```
always @(posedge Foo) Out1 = In2;
```

- **Or we can watch for any change:**

```
@(ControlSignal) Out2 = In1;
```

- **The watched instance can be a gate output, a wire or a register whose value is generated as a activity in another process.**
- **Any number of events can be expressed in the event control statement such that the occurence of any one of them will trigger the execution of the statement.**

```
@(posedge InputA or posedge Timer) ...
```

# Wait Statement

- ***Wait*** **statement waits for its conditional expression to become true.**
- **Level sensitive**
- **Primary for synchronizing two concurrent processes.**

```
module WaitModule(DataOut, DataIn, Ready);
  output [7:0] DataOut;
  input  [7:0] DataIn;
  input        Ready;
  reg          Internal;

  always
    begin
      wait(Ready)
      Internal = DataIn;
      ...
    end
endmodule
```

# ‘define Directive

- **Compiler directive *‘define*** **defines a value and gives a constant textual value to it.**
- **On compilation the text value will be substituted.**

```
`define DvLen 15
`define DdLen 31
`define QLen 15
`define hiDdMin 16

module Divide(DdInput, DvInput, Quotient, Go, Done);
  input [`DdLen:0] DdInput;
  input [`DvLen:0] DvInput;
  output [`QLen:0] Quotient;
```

# If-Then-Else

- **Purely procedural operator (for behavioural modelling). It may appear only in the body of an initial or always statement, or in a subprogram.**
- ***If* statement can be with or without an *else* clause.**

```
if (<expression>)
  <statement_or_null>
else
  <statement_or_null>
```

- **Example (no else):**

```
if (NegDivisor)
  Divisor = -Divisor;
```

- **Example (with else):**

```
if (!Divident[`DdLen])
  Quotient = Quotient+1;
else
  Divident[`DdLen:`HiDdMin] = Divident[`DdLen:`HiDdMin] + Divisor;
```

# If-Then-Else (2)

- **The else clause is paired with the nearest, unfinished if statement.**
- **The begin-end block following the if statement allows all of the encompassed statements to be considered as part of the then statement of the if.**

```
if (Divisor)
  begin
    NegDivisor = Divisor[`DvLen];
    if (NegDivisor)
      Divisor = -Divisor;
    if (!Divident[`DdLen])
      Quotient = Quotient+1;
    else
      Divident[`DdLen:`HiDdMin] = Divident[`DdLen:`HiDdMin] +
Divisor;
  end
```

# Relational Operators

| | | |
|---|---|---|
| > | Determines relative value | Registers and net operands are treated as unsigned. Real and integer operands may be signed. If any bit is unknown, the result will be unknown. |
| >= | Determines relative value | Registers and net operands are treated as unsigned. Real and integer operands may be signed. If any bit is unknown, the result will be unknown. |
| < | Determines relative value | Registers and net operands are treated as unsigned. Real and integer operands may be signed. If any bit is unknown, the result will be unknown. |
| <= | Determines relative value | Registers and net operands are treated as unsigned. Real and integer operands may be signed. If any bit is unknown, the result will be unknown. |

# Relational Operators (2)

| | | |
|---|---|---|
| == | Logical equality | Registers and net operands are treated as unsigned. Real and integer operands may be signed. If any bit is unknown (or high impedance), the result will be unknown |
| != | Logical equality | Registers and net operands are treated as unsigned. Real and integer operands may be signed. If any bit is unknown (or high impedance), the result will be unknown |
| === | Case equality | The bitwise comparison includes comparison of **x** and **z** values. All bits must match for equality. The result is either TRUE or FALSE. |
| !== | Case equality | The bitwise comparison includes comparison of **x** and **z** values. All bits must match for equality. The result is either TRUE or FALSE. |

**4 ‘b 110z == 4 ‘b110z => FALSE**

**4 ‘b 110z === 4 ‘b 110z => TRUE**

# Conditional Operator

- The conditional operator (?:) can be used in place of the if statement when one of two values is to be selected for assignment.
- The general form of the conditional operator is:

```
:: = <expression> ? <expression> : <expression>
```

- If the first expression is TRUE, then the value of the operator is the second expression. Otherwise the value is the third expression.
- May be either a part of a procedural or continuous statement. => Conditional operator can be used both in behavioural and gate level structural modelling.
- Example:

```
Quot = (NegDivisor != NegDivident) ? -Quot : Quot;
```

# Repeat Loop

- **Four different loop statements in Verilog: *repeat*, *for*, *while*, and *forever*.**
- **In a repeat loop a loop count is given in parenthesis**
- **The value of the loop count is determined once at the beginning of the execution of the loop**
- **The loop is executed the given number of times**
- **The general form of the repeat is:**

```
repeat (<expression>) <statement>
```

- **• Example:**

```
repeat (`DvLen+1)
  begin
  ...
  end
```

# For Loop

- **Has an initialized loop counter**

```
::= for ( <assignment>;<expression>;<assignment>)
        <statement>
```

- **The first assignment is executed once at the beginning of the loop (before the body of the loop)**
- **The expression is executed before the body of the loop**
- **Execution stays in the loop while the expression is TRUE**
- **The second assignment is executed after the body of the loop**
- **Example:**

```
for (i=16; i; i=i-1)
 begin
   /* statement(s) */
 end
```

# While Loop

- **Also a constant number of iterations**

  ::= while (<expression>) <statement>

- **If the expression is TRUE, the statement is executed**
- **Example:**

```
i = 16;
while(i)
 begin
  // statement
  i = i-1;
end
```

- **The while statement should not be used to wait for a change in a value generated external to its always statement**

# Forever Loop

- **The forever loop loops forever**
- **The general form:**

```
::= forever <statement>
```

- **Example:**
- **// an abstract microprocessor model**

```
always
begin
  PowerOninitializations;
  forever
    begin
      FetchAndExecute
    end
end
```

# Exiting Loops on Exceptional Conditions

- **Any of the loop statements may be exited through use of the *disable* statement**
- **A disable statement terminates any named begin-end block and execution then begins with the statement following the block**
- **Begin-end blocks may be named by placing the name of the block after a colon following the begin keyword:**

```
begin : break
  for (i=0; i < n; i=i+1)
  begin : continue
    if (a==0)
     disable continue //proceed with i=i+1
     ...
    if (a == b)
     disable break; //exit for loop
     ...
    end
end
```

# Verilog simulation

- In theory...
- Each always and initial statement is simulated as a separate process, one at time.
- Once started, the simulator continues executing a process until either a delay control (#), a wait with a FALSE condition, or an event ( @ ) statement is encountered.
- In those cases, the simulator stops executing the process and finds the next item in the time queue.
- A while statement will never stop the simulator from executing the process.

  => the while statement that waits for an external variable to change will cause the simulator to go into a endless loop
  - the process that controls external variable will never get a chance to change it
- Also wait statement can go into a endless loop