**Verilog Hardware Description Language (HDL)**

**Why Use a HDL**

- Easy way to describe complex digital designs.
- Describe digital designs at a very high level of abstraction (behavioral) and a very low level of abstraction (netlist of standard cells).
- Simulate digital designs using Modelsim, Verilog-XL, etc.
- Synthesize digital designs using Synplicity, XST (Xilinx ISE), Ambit, Synopsys, Leonardo, etc.
- Simulate post synthesis netlist to verify timing.

**Verilog vs. VHDL**

- Both designed for simulation not synthesis
- Verilog uses C like syntax – very concise. VHDL is strongly typed.
- Verilog: “Most widely used Hardware Description Language in the U.S.” (someone at Harvard but the web page was moved).
- “In the United States, the commercial industries tend to use more Verilog, while the aerospace and defense industries more heavily favor VHDL”,[http://www.bitweenie.com/listings/verilog-vs-vhdl](http://www.bitweenie.com/listings/verilog-vs-vhdl)
- IMO, Verilog is easier to learn/use than VHDL.
4 Levels of Abstraction

- Behavioral
  1. Describe the algorithm without concern for the actual logic required to implement it.
  2. For example, a Behavioral description for an 8 bit, 2 input multiplexer is shown below in **bold**:

```verilog
// Mux2To1.v
// Behavioral description of 2 input multiplexer with parameter Width
`resetall
`timescale 1ns/10ps
module Mux2To1(A0, A1, Y, Sel);

parameter Width = 8;
input [Width-1:0] A0, A1;
output [Width-1:0] Y;
inout              Sel;

wire [Width-1:0] A0, A1;
reg [Width-1:0] Y;
wire Sel;
always @ (*)
begin
    case (Sel)
        0: Y = A0 ;
        default: Y = A1 ;
    endcase
end
endmodule
```

Parameter Declarations

```verilog
Width 8
```

![Verilog HDL Diagram](image)
3. What’s all this:
   - `//` - comment character
   - `resetall` - resets all compiler directives to default values. Note that this is `~` (underneath the ~) and not `` underneath the “.“.
   - `timescale 1ns / 10ps` - specifies time unit/precision – Important in your Testbenches when you want to wait 20 ns before changing the stimulus.
   - Create Verilog component with a `module` statement.
     1. `parameter` is used to set constants in Verilog just like the `#define` is used in C. However, the parameter can be overridden during instantiation. This way, Mux2To1 can be used for any size vectors.
     2. Bus size is indicated using `[]`.
     3. Specify direction of ports with input, output or inout.
     4. Declare ports and other signals:
        - `wire` or `reg` - Assumes 1 bit wire if you don’t specify.
        - `case` statement used to describe Mux.
        - Blocks of code are grouped together using begin/end like you use `{ }` in C.
        - More about `always`, `wire` and `reg` later.

- **Dataflow**
  1. Describe the algorithm in terms of logical data flow.
  2. For example, the Dataflow description for an 1 bit, 2 input multiplexer is shown below in **bold**:
     ```verilog
     // Mux2To1DFlow.v
     // Dataflow description of 1 bit, 2 input multiplexor
     `resetall
     `timescale 1ns/10ps
     module Mux2To1DFlow(A0,A1,Y,Sel);
     input   A0, A1;
     output  Y;
     input   Sel;
     wire A0, A1, Y;
     wire Sel;

     assign Y = (A1 & Sel) | (A0 & !Sel) ;
     endmodule
     ```
  3. Some Verilog operators:
     - `&` - bitwise And
     - `&&` - logical And
     - `|` - bitwise Or
     - `||` - Logical Or

  4. Verilog code that combines Dataflow and Behavioral coding styles is commonly referred to as RTL (Register Transfer Language).

- **Gate Level**
1. Describe design in a Netlist of the actual logic gates and the interconnection between them. This is usually generated from the RTL by the Synthesis tool.

2. For example:

```plaintext
mx21 I0_I0_U599 (.Q(I0_I0_n602),.I0(I0_I0_I20_I23_QB[0]),
                 .I1(I0_I0_n_296721027),.S(I0_I0_I20_I23_I0_n_20431));
df202 I0_I0_I20_I23_I8_q_reg_1 (.Q(I0_I0_I20_I23_QB[1]),
                                  .C(I0_I0_CLK0),
                                  .D(I0_I0_n603),.SD(I0_I0_I20_I23_QB[0]),.SE(n_624));
mx21 I0_I0_U600 (.Q(I0_I0_n603),.I0(I0_I0_I20_I23_QB[1]),
                 .I1(I0_I0_I20_I23_I8_n_20536),.S(I0_I0_I20_I23_I0_n_20431));
df202 I0_I0_I20_I23_I8_q_reg_2 (.Q(I0_I0_I20_I23_QB[2]),
                                  .C(I0_I0_CLK0),
                                  .D(I0_I0_n604),.SD(I0_I0_I20_I23_QB[1]),.SE(n_624));
```

- From Cadence’s synthesis tool AMBIT targeting AMI 0.5um standard cell library
- Instantiation of 2 - mx21’s and 2 - df202’s
- Ports connected by name here.
  1. mx21 has 4 ports: Q, I0, I1 and S.
  2. I0_I0_U599 is the name of an instance of a mx21.
     - I0_I0_n602 is a wire connected to the Q input.
- You can also connect without the name if you go in order.

- **Switch Level**

  ```plaintext
  pmos (C,Pwr,B) ;
  pmos (Out,C,A) ;
  nmos(Out,Gnd,A) ;
  nmos(Out,Gnd,B) ;
  ```

1. For example, what circuit is described above? Format: (drain,source,gate)
2. Describe design in a Netlist of switches (FETs), and the interconnect between them.

3. Description of a 2 input Nor gate is shown below:

```verilog
// Nor2Switch.v
module Nor2Switch(A, B, Out);
input A, B;
output Out;
wire C;
supply1 Pwr;
supply0 Gnd;

// Instantiate FETs: pmos(source,drain,gate) or nmos(drain,source,gate)

pmos (C,Pwr,B);
pmos (Out,C,A);
nmos(Out,Gnd,A);
nmos(Out,Gnd,B);
endmodule
```

Figure 1. Nor Gate from Palnitkar, p. 221.
Structural Verilog

- Structural Verilog modules are used to instantiate and connect other Verilog modules together.
- Consider the 8 bit, 3 input multiplexer is shown below:

```verilog
// Mux3To1
// Structural HDL implementation of 3 input, 10 bit mux using 2 Mux2To1’s
// parameterized by Width
`resetall
`timescale 1ns/10ps
module Mux3To1( A0, A1, A2, Sel, Y);
parameter Width = 10;
input [Width-1:0] A0, A1, A2;
input [1:0] Sel;
output [Width-1:0] Y;
wire [Width-1:0] A0, A1, A2, Y, YInt;
wire [1:0] Sel;
Mux2To1 #(Width) U_0( .A0 (YInt), .A1 (A2), .Y   (Y), .Sel (Sel[1:1]));
Mux2To1 #(Width) U_1(A0,A1,YInt,Sel[0:0]);
endmodule // Mux3To1
```

- 2 instances of Mux2To1, U_0 and U_1.
- You can either connect ports by name (U_0) or by declaration order (U_1).
- The #(Width) is used to override the default value of 8 in the Mux2To1 module to create a 10 input mux. This could be still be overridden when you instantiate Mux3To1.
- `wire YInt connects the output of instance U_1 to the A0 input of U_0.
- What happens if Sel = 2'b11?
Combination Logic in Verilog

- 2 ways to create Combinational Logic
  - Use `assign` statement shown above in Mux2To1DFlow.v
  - Operators in Verilog (shown below) are similar to C.

### 6.4 Operator Types

Verilog provides many different operator types. Operators can be arithmetic, logical, relational, equality, bitwise, reduction, shift, concatenation, or conditional. Some of these operators are similar to the operators used in the C programming language. Each operator type is denoted by a symbol. Table 6-1 shows the complete listing of operator symbols classified by category.

<table>
<thead>
<tr>
<th>Operator Type</th>
<th>Operator Symbol</th>
<th>Operation Performed</th>
<th>Number of Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>*</td>
<td>multiply</td>
<td>two</td>
</tr>
<tr>
<td></td>
<td>/</td>
<td>divide</td>
<td>two</td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>add</td>
<td>two</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>subtract</td>
<td>two</td>
</tr>
<tr>
<td></td>
<td>%</td>
<td>modulus</td>
<td>two</td>
</tr>
<tr>
<td>Logical</td>
<td>!</td>
<td>logical negation</td>
<td>one</td>
</tr>
<tr>
<td></td>
<td>&amp;&amp;</td>
<td>logical and</td>
<td>two</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Relational</td>
<td>&gt;</td>
<td>greater than</td>
<td>two</td>
</tr>
<tr>
<td></td>
<td>&lt;</td>
<td>less than</td>
<td>two</td>
</tr>
<tr>
<td></td>
<td>&gt;=</td>
<td>greater than or equal</td>
<td>two</td>
</tr>
<tr>
<td></td>
<td>&lt;=</td>
<td>less than or equal</td>
<td>two</td>
</tr>
<tr>
<td>Equality</td>
<td>==</td>
<td>equality</td>
<td>two</td>
</tr>
<tr>
<td></td>
<td>!=</td>
<td>inequality</td>
<td>two</td>
</tr>
<tr>
<td></td>
<td>===</td>
<td>case equality</td>
<td>two</td>
</tr>
<tr>
<td></td>
<td>!==</td>
<td>case inequality</td>
<td>two</td>
</tr>
<tr>
<td>Bitwise</td>
<td>~</td>
<td>bitwise negation</td>
<td>one</td>
</tr>
<tr>
<td></td>
<td>&amp;</td>
<td>bitwise and</td>
<td>two</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>bitwise or</td>
</tr>
<tr>
<td></td>
<td>^</td>
<td>bitwise xor</td>
<td>two</td>
</tr>
<tr>
<td></td>
<td>-- or ^</td>
<td>bitwise xnor</td>
<td>two</td>
</tr>
<tr>
<td>Reduction</td>
<td>%</td>
<td>reduction and</td>
<td>one</td>
</tr>
<tr>
<td></td>
<td>-=</td>
<td>reduction, and</td>
<td>one</td>
</tr>
<tr>
<td></td>
<td>/=</td>
<td>reduction, or</td>
<td>one</td>
</tr>
<tr>
<td></td>
<td>-=</td>
<td>reduction, not</td>
<td>one</td>
</tr>
<tr>
<td></td>
<td>^</td>
<td>reduction, xor</td>
<td>one</td>
</tr>
<tr>
<td></td>
<td>-- or ^</td>
<td>reduction, xnor</td>
<td>one</td>
</tr>
</tbody>
</table>

Figure 2: Verilog Operators from Palnitkar, p. 92.

- For example:
  - Behavioral description of Full Adder
    ```verilog
    assign {COut, Sum} = a + b + CIn;
    ```
Dataflow description of Full Adder

- assign Sum = Cin ^ A ^ B;
- assign COut = (A&B) | (Cin & (A | B));

<table>
<thead>
<tr>
<th>Cin</th>
<th>A</th>
<th>B</th>
<th>COut</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- assign QOut = 8'b1010101;
- assign QOut = 'ha0;

Use an **always** block
- **If then else** and **case** statements only allowed in an **always** block.
- Outputs must be type **reg**.
- Follow these simple rules to avoid inferring a latch (very bad!):
  - The Sensitivity List is @ *.
  - Assign ALL outputs (LHS of =) under ALL conditions
    - case statements cover all possible cases of the condition or have a default case.
    - if/then/else statements cover all possible cases of the condition or have an else condition.

OR

- All outputs are assigned a default value at the top of the always block. Then, override the defaults as needed with **case** or **if/then/else** statements.
- Use = for assignments. (Blocking)
  - “Left Hand Side” of = are the outputs from this always block (end in D).
  - “Right Hand Side” of = are the inputs to this always block which should be flop outputs (or module inputs or RAM outputs).

- See Mux2To1.v example above.
- Another behavioral always block for Mux2To1
  
  ```verilog
  always @ *
  begin
    Y = A0 ;
    if (Sel == 1)
      Y = A1 ;
  end
  ```
Sequential Logic in Verilog

- Always needs an `always` block.
- Use `<=` for all assignments. (Non-Blocking)
- Outputs must be type `reg`, Inputs may be of type `wire` or `reg`.
- Sensitivity list should only have `posedge Clk` or `negedge Clk`.
  - You won’t need asynchronous resets in this class (i.e., `posedge Reset`)
  - Don’t gate clock! Use only Clk to clock your flops.
- For example, the RTL for a D-Flip Flop is shown below:

```verilog
// DReg.v
// Register - Reset to 0, Latches on LE

`resetall
`timescale 1ns/10ps

module DReg(D, LE, Reset, Clk, Q);
parameter Width = 8;

input  [Width-1:0] D;
input  LE, Reset, Clk;
output [Width-1:0] Q;

wire  [Width-1:0] D;
wire LE, Reset, Clk;
reg    [Width-1:0] Q;

always @ (posedge Clk)
begin
  if (Reset)
    Q <= 0; // Reset Q
  else if (LE)
    Q <= D;
end
endmodule
```
RTL for an up/down counter is shown below:

```verilog
module CntrUpDown(D, LE, CE, Up, Reset, Clk, Count);

parameter Width = 8;
input [Width-1:0] D;
input LE, CE, Up, Reset, Clk;
output [Width-1:0] Count;
wire [Width-1:0] D;
wire LE, CE, Up, Reset, Clk;
reg [Width-1:0] Count ;
reg [Width-1:0] CountD ;

// Combinational Block for CountD
always @ *
begin
  CountD = Count ;
  if (LE)
    CountD = D ;
  else if (CE)
    CountD = Up ? Count + 1 : Count - 1 ;
end

// Sequential Block for Count
always @ (posedge Clk)
begin
  if (Reset)
    Count <= 0 ; // Reset Count
  else
    Count <= CountD ;
end
endmodule
```

// CntrUpDown.v  EJR 11/26/05
// Up-Down Counter with latch enable and count enable.
// Loads Counter with D when LE asserted
// Count Up (Up = 1) or Down (Up = 0) when CE asserted
`resetall
`timescale 1ns/10ps
```
• RTL for a shift register:

```verilog
// ShiftReg.v
// Shift Register - shifts ShiftIn into the MSB of Q.
`timescale 1ns/10ps
module ShiftReg(ShiftIn, LE, Reset, Clk, Q);
parameter Width = 8;
input ShiftIn, LE, Reset, Clk;
output [Width-1:0] Q;

reg [Width-1:0] Q;
wire [Width-1:0] D = {ShiftIn, Q[Width-1:1]};

always @(posedge Clk)
begin
  if (Reset)
    Q <= 0; // Reset Q
  else if (LE)
    Q <= D; // Concatenate ShiftIn and Q >> 1
end
endmodule
```
wire vs. reg

- reg is for outputs of always or initial blocks.
- wire is for everything else.
- Use reg signed or wire signed with 2s complement operations like signed comparisons (> , <).

Verilog Naming Rules

- Case Sensitive
- Names must begin with alpha character
- No special characters in a name except _

Testbenches in Verilog

- Generate stimulus to test your design. View the response to the stimulus in the simulator.
- Never gets synthesized – only used for test.
- Instantiate DUT (Device Under Test) and drive it with your testbench.
- Use the following Verilog constructs
  - initial block – execute at the beginning of the simulation
    - Use = for assignment (Blocking).
    - Outputs are type reg.
  - always block
    - Use = for assignment (Blocking).
    - Outputs are type reg.
  - integer (32 bit) and real (64 bit IEEE double precision) data types available.
  - Use for and while loops inside always and initial blocks. For example:

    ```verilog
    integer i ;
    initial
    begin
      for (i=0;i>10;i=i+1)
        begin
          // Verilog code here
        End
      i = 0 ;
      while (i < 10)
        begin
          // Verilog code here
          i ++ ;
        end
    end
    ```

  - The delay operator # - Use in an assignment statement or on a line by itself to pace the stimulus.
  - $display task – Use this to print results to the simulator console. Uses same format as printf.
  - $readmemh task – Reads an ascii file of hexadecimal values into a reg array.
- **$stop** task – Use this to stop the simulation when stimulus is complete.
  - For example, the testbench, CntrUpDown_tb.v for CntrUpDown is shown below:

**Verilog References**
CntrUpDown Example of an Inferred Latch (VerilogExamples)

- Simulate CntrUpDown_tb
- Synthesize CntrUpDown
- Remove default assignment (line 24)
- Simulate
- Synthesize (look at warnings)

```
//CntrUpDown_tb.v
`resetall
`timescale 1ns/10ps
module CntrUpDown_tb();

parameter Width = 16;

integer i;
reg [Width-1:0] D;
reg LE, CE, Up, Reset, Clk;
wire [Width-1:0] Q;
CntrUpDown #(Width) CntrUpDown1(D, LE, CE, Up, Reset, Clk, Q);
initial
begin
  D = 0; LE = 0; CE = 0; Up = 0; Reset = 1; Clk = 0;
  #8;
  Reset = 0; LE = 1; D = 1000;
  #10;
  LE = 0;
  for (i=0;i<16;i=i+1)
    begin
      CE = 1;
      #10;
      CE = 0;
      #10;
    end
  Up = 1;
  for (i=0;i<16;i=i+1)
    begin
      CE = 1;
      #10;
      CE = 0;
      #10;
    end
$stop;
end
always
begin
  #5 Clk = ~Clk;
end
endmodule
```
Modelsim results for CntrUpDown_tb: