4. Design With Verilog

Dr. Paul D. Franzon

Outline
1. Procedural Examples
2. Continuous Assignment
3. Structural Verilog
4. Common Problems
5. More sophisticated examples

Always Design Before Coding
4.2 Operators, Continuous Assignment, and Structural Verilog

Dr. Paul D. Franzon

Outline
1. Operators
2. Continuous Assignment
3. Structural Verilog
Objectives and Motivation

Objectives:
- Identify the functions captured by the different operators available in Verilog.
- Understand how continuous assignment can be used to specify logic and wire arrangements.
- Understand how module instancing is used to specify a netlist connecting modules together.

Motivation:
- Enrichen your knowledge of synthesizable Verilog that can be described using continuous assignment and structure
- Understand the operators that can be used in procedural code as well
References

- **Ciletti:**
  - Inside front cover: Summary
  - Sections 6.4: Synthesis of tri-states
  - Appendix C: Verilog Data Types
  - Appendix D: Operators
- **Smith and Franzon**
  - Sections 2.1, 2.2: Datatypes
  - Chapter 3: Structural code and continuous assignment
- **Sutherland Reference Guide**
11.0 Operators

<table>
<thead>
<tr>
<th>Usage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>m + n Add n to m</td>
</tr>
<tr>
<td>-</td>
<td>m - n Subtract n from m</td>
</tr>
<tr>
<td>-</td>
<td>-m Negate m (2's complement)</td>
</tr>
<tr>
<td>*</td>
<td>m * n Multiply m by n</td>
</tr>
<tr>
<td>/</td>
<td>m / n Divide m by n</td>
</tr>
<tr>
<td>%</td>
<td>m % n Modulus of m / n</td>
</tr>
</tbody>
</table>

**Bitwise Operators**

| ~     | -m Invert each bit of m |
| &     | m & n AND each bit of m with each bit of n |
| |  | OR each bit of m with each bit of n |
| ^     | m ^ n Exclusive OR each bit of m with n |
| ^=    | m ^= n Exclusive NOR each bit of m with n |

Use with great care (big!)

Not synthesizable

reg [1:0] A, B, C, D;
assign C = A & B;

c/- Sutherland Reference Guide
### Unary Reduction Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>&amp;m</td>
<td>AND all bits in m together (1-bit result)</td>
</tr>
<tr>
<td>NAND</td>
<td>~&amp;m</td>
<td>NAND all bits in m together (1-bit result)</td>
</tr>
<tr>
<td>OR</td>
<td></td>
<td>m</td>
</tr>
<tr>
<td>NOR</td>
<td>~</td>
<td>m</td>
</tr>
<tr>
<td>XOR</td>
<td>^</td>
<td>m</td>
</tr>
<tr>
<td>Exclusive NOR</td>
<td>^~m</td>
<td>Exclusive NOR all bits in m (1-bit result)</td>
</tr>
</tbody>
</table>

```verilog
reg [1:0] B;
reg C;
assign C = & B;
```

![Diagram](image)
### Logical Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>True/False Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>!m</td>
<td>m not true (1-bit)</td>
</tr>
<tr>
<td>&amp; &amp;</td>
<td>m &amp; n</td>
<td>m and n true</td>
</tr>
<tr>
<td></td>
<td></td>
<td>n</td>
</tr>
</tbody>
</table>

### Equality Operators (compares logic values of 0 and 1)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>True/False Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>m == n</td>
<td>m equals n</td>
</tr>
<tr>
<td>!=</td>
<td>m != n</td>
<td>m not equals n</td>
</tr>
</tbody>
</table>

### Identity Operators (compares logic values of 0, 1, X and Z)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>True/False Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>m == n</td>
<td>m identical to n</td>
</tr>
<tr>
<td>!=</td>
<td>m != n</td>
<td>m not identical to n</td>
</tr>
</tbody>
</table>

### Relational Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>True/False Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>m &lt; n</td>
<td>m less than n</td>
</tr>
<tr>
<td>&gt;</td>
<td>m &gt; n</td>
<td>m greater than n</td>
</tr>
<tr>
<td>&lt;=</td>
<td>m &lt;= n</td>
<td>m less than or equal to n</td>
</tr>
<tr>
<td>&gt;=</td>
<td>m &gt;= n</td>
<td>m greater than or equal to n</td>
</tr>
</tbody>
</table>

```plaintext
reg A, B, C, D, E, F;
if (A<B) then if (C==D) then if (E&&F)
```
### Logical Shift Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;&lt;</code></td>
<td><code>m &lt;&lt; n</code> Shift <code>m</code> left <code>n</code>-times</td>
</tr>
<tr>
<td><code>&gt;&gt;</code></td>
<td><code>m &gt;&gt; n</code> Shift <code>m</code> right <code>n</code>-times</td>
</tr>
</tbody>
</table>

### Miscellaneous Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>? :</code></td>
<td><code>sel?m:n</code> If <code>sel</code> is true, select <code>m</code>; else select <code>n</code></td>
</tr>
<tr>
<td><code>{}</code></td>
<td><code>{m,n}</code> Concatenate <code>m</code> to <code>n</code>, creating larger vector</td>
</tr>
<tr>
<td><code>{{}}</code></td>
<td><code>{n[m]}</code> Replicate <code>m</code> <code>n</code>-times</td>
</tr>
<tr>
<td><code>-&gt;</code></td>
<td><code>-&gt; m</code> Trigger an event on an event data type</td>
</tr>
</tbody>
</table>

- **Mux**
- **Rearranging bits**
- **Not synthesizable**

Verilog 2001 adds `<<<` and `>>>` (signed shift) and `**` (exponentiation)
Continuous Assignment

Sketch the logic being specified ...

```verilog
input [3:0] A, B;
wire [3:0] C, E;
wire D, F, G;
assign C = A ^ B;
assign D = |A;
assign F = A[0] ? B[0] : B[1];
assign G = (A == B);
```
Continuous Assignment

Sketch the logic being specified ...

```
input A, B, C;

tri F;

assign F = A ? B : 1'bz;
assign F = ~A ? C : 1'bz;
```
Continuous Assignment

Sketch the logic being specified ...

input [3:0] A, B, C;
wire [3:0] F, G;
wire H;
assign F = A + B + C + D;
assign G = (A+B) + (C+D);
assign H = C[A[1:0]];

+  
A
B
C
D  F

+  
A
B
C
D  G

+  
0
C[3]
C[2]
C[1]
C[0]

A[1:0]
H

Faster!
Structural Verilog

Complex modules can be put together by ‘building’ (instancing) a number of smaller modules.

e.g. Given the 1-bit adder module with module definition as follows, build a 4-bit adder with carry_in and carry_out

```verilog
module OneBitAdder (CarryIn, In1, In2, Sum, CarryOut);

4-bit adder:
module FourBitAdder (Cin, A, B, Result, Cout);
input    Cin;
input [3:0] A, B;
output [3:0] Result;
output    Cout;
wire [3:1] chain;

OneBitAdder u1 (.CarryIn(Cin), .In1(A[0]), .In2(B[0]),
                   .Sum(Result[0]), .CarryOut(chain[1]));
OneBitAdder u2 (.CarryIn(chain[1]), .In1(A[1]), .In2(B[1]),
                   .Sum(Result[1]), .CarryOut(chain[2]));
OneBitAdder u3 (.CarryIn(chain[2]), .In1(A[2]), .In2(B[2]),
                   .Sum(Result[2]), .CarryOut(chain[3]));
OneBitAdder u4 (Chain[3], A[3], B[3], Result[3], Cout); // in correct order
endmodule
```
Structural Example

- Sketch:
Structural Verilog

Features:
Four copies of the same module (OneBitAdder) are built (‘instanced’) each
with a unique name (u1, u2, u3, u4).

Module instance syntax:
OneBitAdder u1 (.CarryIn(Cin),
Module Name    Instance Name   Port Name inside Module (optional)
Net name

All nets connecting to outputs of modules must be of wire type (wire or tri):
wire [3:1] chain;

(Note: Illustrative only, NOT a good way to build an adder)
Applications of Structural Verilog

- To assemble modules together in a hierarchical design.
- Final gate set written out in this format (“netlist”).
- Design has to be implemented as a module in order to integrate with the test fixture.

Hierarchy and Scope:
- Implements hierarchy
  - Copies of OneBitAdder are instanced inside the module FourBitAdder
- Variable scope
  - CarryIn: Scope is inside Module
  - chain[1]: Scope is inside Module
  - u2.CarryIn: Allows CarryIn in module u2 to be referenced from FourBitAdder (useful in traversing hierarchy in simulator)
Notes on Hierarchy

- Generally it is a good idea to only implement logic in the leaf cells of a hierarchical design, and not at a higher level

  i.e.

  ```verilog
  module good(A,B,C);
  good_leaf u1(A,B);
  good_leaf u2(A,C);
  endmodule
  ```

  ```verilog
  module bad(A,B,C);
  assign D=C&D;
  good_leaf u1(A,B);
  good_leaf u2(A,C);
  endmodule
  ```

- Why?
  - Hint: Consider what module must be synthesized in a single run

- Note: See Hierarchy notes for more on partitioning
module counter ( clock, in, latch, dec, zero );

input [3:0] in;
input clock, latch, dec;
output zero;
wire value[3], value[1], value53[2], value53[0], n54[0],
value[2], value[0], value53[1], value53[3], n103, n104, n105,
n106, n107, n108, n110, n111, n112, n113, n114, n115;
NOR2 U36 ( .Y(n107), .A0(n109), .A1(value[2]) );
NAND2 U37 ( .Y(n109), .A0(n105), .A1(n103) );
NAND2 U38 ( .Y(n114), .A0(value[1]), .A1(value[0]) );
NOR2 U39 ( .Y(n115), .A0(value[3]), .A1(value[2]) );
XOR2 U40 ( .Y(n110), .A0(value[2]), .A1(n108) );
NAND2 U41 ( .Y(n113), .A0(n109), .A1(n114) );
INV U42 ( .Y(n54[0]), .A(n106) );
INV U43 ( .Y(n108), .A(n109) );
AOI21 U44 ( .Y(n106), .A0(n112), .A1(dec), .B0(latch) );
INV U45 ( .Y(zero), .A(n112) );
NAND2 U46 ( .Y(n112), .A0(n115), .A1(n108) );
OAI21 U47 ( .Y(n111), .A0(n107), .A1(n104), .B0(n112) );
DSEL2 U48 ( .Y(value53[3]), .D0(n111), .D1(in[3]), .S0(latch) );
DSEL2 U49 ( .Y(value53[2]), .D0(n110), .D1(in[2]), .S0(latch) );
DSEL2 U50 ( .Y(value53[1]), .D0(n113), .D1(in[1]), .S0(latch) );
DSEL2 U51 ( .Y(value53[0]), .D0(n105), .D1(in[0]), .S0(latch) );
EDFF value_reg[3] ( .Q(value[3]), .QBAR(n104), .CP(clock), .D(value53[3]), .E(n54[0]) );
EDFF value_reg[2] ( .Q(value[2]), .CP(clock), .D(value53[2]), .E(n54[0]) );
EDFF value_reg[1] ( .Q(value[1]), .QBAR(n103), .CP(clock), .D(value53[1]), .E(n54[0]) );
EDFF value_reg[0] ( .Q(value[0]), .QBAR(n105), .CP(clock), .D(value53[0]), .E(n54[0]) );
endmodule
**Variable type rules**

```verilog
definition
module Test (A,B,C,D,E) 
  in A;
  out B, C, D;
  inout E;

  wire B, D;
  reg C;
  tri E;

  always@(A) C=A;
  assign B=A
  assign E = A ? 1'bz : D;

  Nested ul (D);
endmodule
```

*Inside module:*

- Inputs implicitly a net data type (e.g. wire)
  - (Not usually an issue in synthesizable design as should not be modified in module)
- Outputs neither net or register type
  - (Must be declared based on how assigned)
- Inout can NOT be a register type
  - (a tri declaration makes most sense in a synthesizable design)
- Outputs of instanced modules are of type wire
Exercises

Which alternative best describes the behavior of the logic in the following verilog fragment.

```verilog
wire [4:0] A;
wire [2:0] B;
```

If A = 5’b10101, then

A. B=3’b000;
B. B=3’b011;
C. B=3’b100;
D. B=3’b111;
### Exercises

**Sketch Design, including hierarchy;**

Declare needed variables and ports

```verilog
module Ex1 (A, B, C);
  input [1:0] A;
  mod1 u1 (A, B);
  mod2 u2 (A, B, C);
endmodule

module mod1 (E, F)
  input [1:0] E;
  assign F = &E;
endmodule

module mod2 (A, B, C)
  input [1:0] A;
  mod3 u3 (A, B, C);
endmodule

module mod3 (G, H, I)
  input [1:0] G;
  assign I = G & {2{H}};
endmodule
```

```verilog
output B;
output [1:0] C;
wire B;
wire [1:0] C;
output F;
wire F;
input H;
output [1:0] I;
wire [1:0] I;
```
Questions: Continuous Assignment

- When are expressions evaluated?

- What types of variables can be assigned?

- Is this the only way to build synthesizable tri-state buffers?

Exercise -- Use Continuous Assignment to Make an even Parity Generator:

```vhdl
wire [31:0] A;
e.g. A=32'b1010  \(\rightarrow\) even_parity = 1;
wire even_parity;
```
Questions: Structural Verilog

- Outputs of instanced modules should be of what type?

- Should you have logic above the leaf cell level in a hierarchical design?