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><strong>Arithmetic Operators</strong></td>
</tr>
<tr>
<td>+</td>
<td>m + n</td>
</tr>
<tr>
<td>-</td>
<td>m - n</td>
</tr>
<tr>
<td>-</td>
<td>-m</td>
</tr>
<tr>
<td>*</td>
<td>m * n</td>
</tr>
<tr>
<td>/</td>
<td>m / n</td>
</tr>
<tr>
<td>%</td>
<td>m % n</td>
</tr>
<tr>
<td></td>
<td><strong>Bitwise Operators</strong></td>
</tr>
<tr>
<td>~</td>
<td>~m</td>
</tr>
<tr>
<td>&amp;</td>
<td>m &amp; n</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>m ^ n</td>
</tr>
<tr>
<td>~^</td>
<td>m ~^ n</td>
</tr>
</tbody>
</table>

Use with great care (big!)

}\ Not synthesizable

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

\- Sutherland Reference Guide
reg [1:0] B;
reg C;
assign C = & B;

\[
\begin{array}{|c|c|}
\hline
\text{Unary Reduction Operators} & \text{Description} \\
\hline
\& & \&m \text{ AND all bits in } m \text{ together (1-bit result)} \\
\sim\& & \sim\&m \text{ NAND all bits in } m \text{ together (1-bit result)} \\
| & |m \text{ OR all bits in } m \text{ together (1-bit result)} \\
\sim| & \sim|m \text{ NOR all bits in } m \text{ together (1-bit result)} \\
^\wedge & ^\wedge m \text{ Exclusive OR all bits in } m \text{ (1-bit result)} \\
\sim^\wedge & \sim^\wedge m \text{ Exclusive NOR all bits in } m \text{ (1-bit result)} \\
\hline
\end{array}
\]
### Logical Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>Indicates if m is not true? (1-bit True/False result)</td>
<td>True/False</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Checks if both m and n are true? (1-bit True/False result)</td>
<td>True/False</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>Checks if m is equal to n? (1-bit True/False result)</td>
<td>True/False</td>
</tr>
<tr>
<td>!=</td>
<td>Checks if m is not equal to n? (1-bit True/False result)</td>
<td>True/False</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>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>===</td>
<td>Checks if m is identical to n? (1-bit True/False result)</td>
<td>True/False</td>
</tr>
<tr>
<td>!==</td>
<td>Checks if m is not identical to n? (1-bit True/False result)</td>
<td>True/False</td>
</tr>
</tbody>
</table>

### Relational Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>Checks if m is less than n? (1-bit True/False result)</td>
<td>True/False</td>
</tr>
<tr>
<td>&gt;</td>
<td>Checks if m is greater than n? (1-bit True/False result)</td>
<td>True/False</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Checks if m is less than or equal to n? (True/False result)</td>
<td>True/False</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Checks if m is greater than or equal to n? (True/False result)</td>
<td>True/False</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 m left n-times</td>
</tr>
<tr>
<td><code>&gt;&gt;</code></td>
<td><code>m &gt;&gt; n</code> Shift m right n-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 sel is true, select m: else select n</td>
</tr>
<tr>
<td><code>{}</code></td>
<td><code>{m,n}</code> Concatenate m to n, creating larger vector</td>
</tr>
<tr>
<td><code>{ { } }</code></td>
<td><code>{n[m]}</code> Replicate m n-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

- Directly specifies logical structure
  - Procedural code often specifies *behavior* of logic

```vhdl
wire [1:0] B, C, D;
tri [1:0] E;

assign D = B & C;
assign E = B[0] ? C : D;
```
Examples

wire [3:0] sum;
wire carry;

// A, B, C 4 bits wide
// D 1 bit wide

assign {carry, sum} = A + (D ? B : C);
Examples

```verilog
wire [2:0] E, F;
wire [1:0] G, H;
// A, B 3-bits wide

assign E = A << 2;
assign F = B >>> 1;
assign G = {2{A[0]}};
assign H = {A[1],B[1]};
```
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]];
Summary – Continuous Assignment

- What is the difference between &B and A&B?

- Are the following statements “legal”
  ```c
  // A and B are 4-bits wide
  if (A == B) ...
  if (A && B) ...
  if (&B) ...
  ```
Summary – Continuous Assignment

- What operators ONLY have the function to rearrange, replicate, etc. bits?
  - i.e. Don’t produce logic

- When are continuous assignment statements evaluated?

- All variables assigned in continuous assignment must be of what type(s)?
Summary – Continuous Assignment

- In contrast, variables assigned in procedural blocks must be of what type?

- How do you specify a synthesizable tri-state buffer?
**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

```
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        Cout;
wider [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:

![Diagram of a structural example showing a FourBitAdder with OneBitAdder components connected in a chain.](image)
Instance Formats

- Module instance formats

```
// Simple format in which variable order matters
OneBitAdder u1 (Cin, A[0], B[0], Result[0], chain[1]);

// Advanced format in which variable order does not matter
OneBitAdder u2 (.CarryIn(chain[1]), .In1(A[1]), .In2(B[1]), .Sum(Result[1]), .CarryOut(chain[2]));
```
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)
No glue logic!

- 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.

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

  ```
  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 Hieararchy 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] , 
n106, n107, n109, n110, n111, 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(\value3 ), .A1(\value2 ) );
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(\value3 ), .QBAR(n104), .CP(clock), .D( 
\value53[3] ), .E(\n54[0] ) );
\n54[0] ) );
EDFF \value_reg[1] ( .Q(\value1 ), .QBAR(n103), .CP(clock), .D( 
\value53[1] ), .E(\n54[0] ) );
EDFF \value_reg[0] ( .Q(\value0 ), .QBAR(n105), .CP(clock), .D( 
\value53[0] ), .E(\n54[0] ) );
endmodule
Variable type rules

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'b0 : 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
Parameter

- Parameter is like “constant” in C

```vhdl
parameter bit_width = 8;

reg [bit_width-1 : 0] Register;
```
In Verilog 2001…

- **Parameter Passing**
  - Can pass parameters to modules, overwriting their local values

```verilog
module top;
Parameter RFsize1 = 64;
Parameter AddressSize1 = 6;

RegFile #( .size(RFsize1), .Asize(AddressSize1) ) U1 ( ... );
endmodule

module RegFile( ... 
    input [Asize-1:0] WriteAddress, ReadAddress, 
);  
parameter Asize=5;
parameter size=32;
reg [15:0] Register [0:size-1];
```
**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

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

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:

```verilog
wire [31:0] A;
e.g. A=32’b1010  ➔ 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?