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
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
# Arithmetic Operators

<table>
<thead>
<tr>
<th>Usage</th>
<th>Meaning</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>A + B</td>
<td>add</td>
</tr>
<tr>
<td>-</td>
<td>A - B</td>
<td>subtract</td>
</tr>
<tr>
<td>-</td>
<td>- A</td>
<td>negate</td>
</tr>
<tr>
<td>*</td>
<td>A * B</td>
<td>multiply</td>
</tr>
<tr>
<td>/</td>
<td>A / B</td>
<td>divide</td>
</tr>
<tr>
<td>%</td>
<td>A % B</td>
<td>modulus</td>
</tr>
<tr>
<td>**</td>
<td>A ** B</td>
<td>A to the power B</td>
</tr>
</tbody>
</table>

e.g. B = -A; Forms the two’s complement of A and stores it in B
Bitwise operators

Operate vectors bit-by-bit

<table>
<thead>
<tr>
<th>Usage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td>Invert each bit of A</td>
</tr>
<tr>
<td>&amp;</td>
<td>AND each bit of A with B</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>XOR each bit of A with B</td>
</tr>
<tr>
<td>~^</td>
<td>XNOR each bit of A with B</td>
</tr>
</tbody>
</table>

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

![Diagram of AND gate]
Unary Reduction Operators

- Operate on all the bits of the vector to produce a unary (one bit) result

<table>
<thead>
<tr>
<th></th>
<th>Usage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>&amp;A</td>
<td>AND all bits of A together</td>
</tr>
<tr>
<td>~&amp;</td>
<td>~&amp;A</td>
<td>NAND all bits of A together</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>OR all bits of A together</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NOR all bits of A together</td>
</tr>
<tr>
<td>^</td>
<td>^A</td>
<td>XOR all bits of A together</td>
</tr>
<tr>
<td><del>^ or ^</del></td>
<td>~^A or ^~A</td>
<td>XNOR all bits of A together</td>
</tr>
</tbody>
</table>

```plaintext
reg [1:0] B;
wire C;
assign C = & B;
```
**Logical Operators**

- Take one-bit inputs and produce a 1-bit result
- Generally used in if-else and case statements but can be used for single bit logic
- Conventional operators can also be used in if-else and case statements IF they produce a 1-bit result

<table>
<thead>
<tr>
<th>Usage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>!A</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>A&amp;&amp;B</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A is not true
A and B are both true
A or B are true

Examples:
```
reg A, B;
if (A&&B)
if (A&B) // also OK

reg [1:0] A, B;
if (A&&B) // NOT OK
```
## Equality and Identity Operators

- Also produce a 1-bit result
- Can be used in if-else, case statements or to create logic

**Equality operators only compare logic 0 and 1**

<table>
<thead>
<tr>
<th>Usage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>==</code></td>
<td>A==B</td>
</tr>
<tr>
<td><code>!=</code></td>
<td>A!=B</td>
</tr>
</tbody>
</table>

**Identity operators compare logic 0 and 1, x and z**

<table>
<thead>
<tr>
<th>Usage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>===</code></td>
<td>A===B</td>
</tr>
<tr>
<td><code>!==</code></td>
<td>A!==B</td>
</tr>
</tbody>
</table>

Example: \(A=4'b00xx; \quad B=5'b100xx\)

\[
C=(A===B[4:0]); \quad // \quad C=1
\]

\[
D=(A==B[4:0]); \quad // \quad D=x : \quad x \text{ returned if } z \text{ or } x \text{ in operators}
\]
Relational Operators

- Also produce a 1-bit result:

<table>
<thead>
<tr>
<th>Usage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; A&lt;B</td>
<td>True if A&lt;B</td>
</tr>
<tr>
<td>&gt; A&gt;B</td>
<td>True if A&gt;B</td>
</tr>
<tr>
<td>&lt;= A&lt;=B</td>
<td>True if A&lt;=B</td>
</tr>
<tr>
<td>&gt;= A&gt;=B</td>
<td>True if A&gt;=B</td>
</tr>
</tbody>
</table>
**Shift**

**Logical Shift**

<table>
<thead>
<tr>
<th>Usage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;&lt;</td>
<td>A &lt;&lt; B  Shift A left by B bits, zero fill on right</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>A &gt;&gt; B  Shift A right by B bits, zero fill on left</td>
</tr>
</tbody>
</table>

**Arithmetic Shift (Verilog 2001)**

<table>
<thead>
<tr>
<th>Usage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;&lt;&lt;</td>
<td>A &lt;&lt;&lt; B  Shift A left by B bits, zero fill on right</td>
</tr>
<tr>
<td>&gt;&gt;&gt;&gt;</td>
<td>A &gt;&gt;&gt;&gt; B  Shift A right by B bits, sign fill on left if A is signed, zero filled if not</td>
</tr>
</tbody>
</table>

Note: Shifting by a fixed amount just rearranges wires
Shift by a variable amount creates a shifter in combinational logic.
## Miscellaneous Operators

<table>
<thead>
<tr>
<th>Usage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>? :</td>
<td>sel ? A : B If sel, choose A else choose B (similar to if else)</td>
</tr>
<tr>
<td>{}</td>
<td>{A,B} Concatenate A and B, creating a longer vector</td>
</tr>
<tr>
<td>{}{}</td>
<td>{A{B}} Replicate B A times, creating a longer vector</td>
</tr>
<tr>
<td>-&gt;</td>
<td>-&gt;A Trigger Not synthesizable</td>
</tr>
</tbody>
</table>
Comments on Operations

Operator Expansion

- Operators expanded to the largest element in the expression, before evaluation
  - Unsigned vectors are zero filled
  - Signed vectors are sign bit filled
- {} and {{}} are done BEFORE expansion, and then treated as an operator at the new width
- If assigned to a shorter variable, high order bits are lost
  
  Ex: 
  ```
  wire [3:0] A, B, C;
  assign C =     ({A,A} >> 2) | B[1:0]
  //C = lower 4 bits of (8 bits | 0000 00B[1]B[0])
  ```

Signed Arithmetic (Verilog 2001)

- If any operand is unsigned, arithmetic is unsigned
- If ALL operators are signed, signed arithmetic is conducted
- Operator can be cast as $signed or $unsigned
**Operator Precedence**

Order in which operators are evaluated

For example, in

\[ A = A == B \ ? \ {C,D} \ : \ {2\{E\}} \ | \ F^G; \]

Order evaluation:

1. \{2\{E\}\} \ {C,D}\n2. \(F^G)\n3. A == B\n4. |\n5. ?:\

No () needed in this example
Examples

reg [1:0] A, B, C;
reg [2:0] D, E, F;
reg [3:0] H;
reg G;

always@(*) begin
    C = A | E; // C=11
    D = E << 2; // D=100
    G = |A; // G=1
    F = {A, B[0]} | E; // F=101
    H = {2{B}}; // H=0101
end

// C=11
// D=100
// G=1
// F=101
// H=0101

Q1: If A = 2'b10; B=2'b01; E = 3'b101; what is C, D, G, F, H after execution?
reg [1:0] A, B, C, D;
reg [2:0] E, F;
reg [3:0] H;
reg G;

always@(*) begin
    C = A | E;
    D = E << 2;
    G = |A;
    F = {A, B[0]} | E;
    H = {2{B}};
end
Examples

```verilog
wire [2:0] E;
wire [2:0] signed 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]};
```

![Diagram of signal assignments](diagram.png)
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);
```

![Diagram of logic circuits]
Examples

- Shifts, arithmetic, signed, unsigned

Example:

```verilog
reg signed [3:0] A, B, C;
reg [3:0] D, E, F;
always@(*)
begin
  \ assume for calculation that A=D=1001
  B = A <<< 1;    // B=0010
  C = A >>> 1;    // C=1100
  E = D <<< 1;    // E=0010
  F = D >> 1;     // F=0100
end
```

A[1]--- B[1]
A[0]--- B[0]

0

A[3]--- C[3]
A[1]--- C[1]
A[0]--- C[0]
Summary

- Verilog contains many useful operators
  - Arithmetic +-*
  - Logic &A ^A |A A&B A^B A|C etc.
  - Relational === == > etc.
  - Shift – logical and arithmetic
  - Concatanation and replication – extending vectors
  - ?: = if else

- Go to sub-module quiz and then to next sub-module
Continuous Assignment

- Directly specifies logical structure in one line of code
  - Procedural code often specifies behavior of logic

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

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

Statements executed whenever anything on RHS changes ("continuously assigned")

All variables assigned in this way must be type wire or tri
Examples

wire [3:0] sum;
wire carry;
// A, B, C 4 bits wide
// D 1 bit wide

assign {carry, sum} = A + (D ? B : C);
    // Note A+D ? B : C would be very different!
Continuous Assignment

Sketch the logic being specified ...

```vhdl
input A, B, C;
tri F;
assign F = A ? B : 1'bz;
assign F = ~A ? C : 1'bz;
```

Tri-State Buffer
• Connect to shared bus

In general only use a shared bus as a last resort (can be hard to debug).

(Exception: Some FGPAs have lots of tri-state buffers - use as a mux.)
Further Examples

wire [3:0] A, B, D;
wire [1:0] C, E;
assign B = {A,A} >> C;
assign D = {4{A[3]},A} >> E;

D: Signed Right Shift
wire [3:0] A;
wire [1:0] B;
wire C;
assign C = A[B];

B: Rotator
C: Mux
Summary – Continuous Assignment

- Any one line expression is efficiently captured using continuous assignment
- Tri-state muxes MUST be specified using continuous assignment
- All variables assigned continuously must be of type wire or tri

Go to sub-module quiz before proceeding to next module
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);

module FourBitAdder (Cin, A, B, Result, Cout);

input Cin;
input [3:0] A, B;
output [3:0] Result;
output Cout;
wire 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 u4 (Chain[3], A[3], B[3], Result[3], Cout); // in correct order
endmodule
```
**Structural Example**

- **Sketch:**

```
+------------------------+    +------------------------+
| FourBitAdder           |    | FourBitAdder           |
| u1                     |    | u2                     |
| OneBitAdder            |    | OneBitAdder            |
| CarryIn                |  u1 | CarryIn                |
| In1                    |    | In1                    |
| In2                    |    | In2                    |
| Sum                    |    | Sum                    |
| CarryOut               |    | CarryOut               |
| Result[0]              |    | Result[0]              |
| Etc.                   |    | Etc.                   |
```

©2013, Dr. Paul D. Franzon, www.ece.ncsu.edu/erl/faculty/paulf.html
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]));
```

Same order as module port order

Name of port in module
OneBitAdder

Name of variable in module
FourBitAdder
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 OneBitAdder Module
  - chain[1]: Scope is inside FourBitAdder 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.

```verilog
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.

  *good_leaf* can be synthesized once and replicated in *good*.

  *In bad, bad must be synthesized including all logic (longer synthesis run)*

- Note: See Hieararchy notes for more on partitioning.
Sample Netlist

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, n109, 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] ) );
  XOR2 U40 ( .Y(n110), .A0(\value[2] ), .A1(n108) );
  NAND2 U41 ( .Y(n113), .A0(n109), .A1(n114) );
  INV U42 ( .Y(\value54[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[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

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

Note: Variables inside module Nested are DIFFERENT variables, even if I give them the same name
Parameter

- Parameter is like “constant” in C

```verilog
class parameter bit_width = 8;

reg [bit_width-1 : 0] Register;
```
**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];
```
Exercise

Sketch Design, including hierarchy;
Declare needed variables and ports

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

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

module mod2 (G,H,I)
input [1:0] G;
input H; output [1:0] I;
wire [1:0] I;
assign I = G & {2{H}};
endmodule
Summary – Structural Verilog

- Structural Verilog permits instanced modules to be wired together in a netlist.
- The outputs of all modules must be declared as type wire within the module they are instanced.
- Parameter passing is useful to make resizable designs whose size can be defined from the top level of the hierarchy.

- Goto sub-module quiz and next sub-module in next unit.