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
4.1 Procedural Code

Dr. Paul D. Franzon

Outline
1. Literals
2. Flip-flops
3. Behavioral logic descriptions

References
1. Quick Reference Guide
2. Ciletti, Ch. 4-6
3. Smith & Franzon, Chapter 2-6, 8, Appendix A
Objectives

Objectives:

- Describe how to represent numbers in Verilog.
- Understand why non-blocking assignment should be used when flip-flops are being inferred.
- Describe how some of the different types of flip-flops can be represented in Verilog.
- Describe how to capture complex combinational logic in procedural blocks, including encoders and decoders.
- Identify how to represent latches in procedural blocks, both intentionally and unintentionally.
- Understand how don’t cares are used in procedural blocks.
- Identify when a for loop is appropriate to use when describing combinational logic.
Motivation

Motivation:

- Enrichen your knowledge of synthesizable Verilog that can be described procedurally
- Justify use of “<=” when assigning to flip flops
- Capturing the behavior of a complex logic block in a procedural assignment can be very useful
References

- Ciletti:
  - Inside front cover: Summary
  - Sections 5.6, 5.7: Flip-flops
  - Section 5.8: Procedural code examples
  - Sections 6.2: Synthesis of various blocks (priority, don’t cares)
  - Appendix C: Verilog Data Types
  - Appendix G.6 G.7: Explains assignment and simulator operation
  - Appendix H: Flip-flops
  - Appendix I: Verilog 2001

- Smith and Franzon
  - Sections 2.1, 2.2: Datatypes
  - Chapter 5: Procedural code

- Sutherland Reference guide: Good crisp refresher
**Lexical Conventions in Verilog**

Logic values:

<table>
<thead>
<tr>
<th>Logic Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Zero, low, or false</td>
</tr>
<tr>
<td>1</td>
<td>One, high or true</td>
</tr>
<tr>
<td>Z or z or ?</td>
<td>High impedance, tri-stated or floating</td>
</tr>
<tr>
<td>X or x</td>
<td>Unknown, uninitialized, or Don’t Care</td>
</tr>
</tbody>
</table>

Integers:

\[
1\text{’b1;} \quad 4\text{’b0;} \\
\text{size ‘base value ;} \quad \text{size = # bits, \ HERE: base = binary} \\
\text{NOTE: zero filled to left}
\]

Other bases: \( h = \text{hexadecimal, d = decimal (which is the default)} \)

Examples:

<table>
<thead>
<tr>
<th>Integer</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3’b1</td>
</tr>
<tr>
<td>8’hF0</td>
<td>8’hF</td>
</tr>
<tr>
<td>5’d11</td>
<td>2’b10</td>
</tr>
</tbody>
</table>
**Procedural Blocks**

Code of the type

```vhdl
always@(input1 or input2 or ...)
begin
    if-then-else or case statement, etc.
end
```

is referred to as *Procedural Code*

- Statements between `begin` and `end` are executed *procedurally*, or in order.
- Variables assigned (i.e. on the left hand side) in procedural code must be of a *register data type*. Here type `reg` is used.
  - Variable is of type `reg` does NOT mean it is a register or flip-flop.
- The procedural block is executed when triggered by the `always@` statement.
  - The statements in parentheses ( . . . ) are referred to as the *sensitivity list*. 
Negative Numbers

- Unless explicitly managed by the designer, Verilog uses 2’s complement arithmetic
- How do you form the two’s complement of a number, e.g. A=3’d1?

Example:

```verilog
reg [2:0] A, B, neg;
reg [3:0] C, D, sum;
always@(*) begin
    A = 3’d-1; // A = 111
    C = 4’d-2; // C = 1110
    D = 3’d-1;
    A = 3’d2; B = 3’d4;
    sum = A - B;
end
```

Adding two B-bit numbers produces an N+1 bit result, due to carry out.
Blocking vs. Non-Blocking

- Or Why I use “<=” to specify flip-flops
- Blocking:
  
  ```
  Begin
  A = B;
  C = D;
  End
  ```
  - Assignment of C **blocked** until A=B completed; i.e. They execute in sequence

- Non-Blocking
  
  ```
  Begin
  A <= B;
  C <= D;
  End
  ```
  - Assignment of C **NOT blocked** until A=B completed
  - i.e. They execute in parallel
**Blocking Statements**

- Hand execute the following:
  ```vhdl
  // test fixture
  initial
  begin
    a = 4'h3; b = 4'h4;
  end
  // code
  always@(posedge clock)
  begin
    c = a + b;
    d = c + a;
  end;
  ```

- Results:
  
  c = 7, d = 10

In "d = c + a", c is the value it will have after the block completes execution. i.e. The value at the input of the flip-flop when d is being evaluated.
**Non-Blocking**

- Contrast it with this code:

```vhdl
// test fixture
initial
begin
    a = 4'h3; b = 4'h4; c=4'h2;
end
// code
always@(posedge clock)
begin
    c <= a + b;
    d <= c + a;
end;
```

After execution:

```
c=7; d=5;
```
Why is this?

- Because how Verilog captures the intrinsic parallelism of hardware.

Inside the Sim:

- `always@(posedge clock) begin` is used to trigger the evaluation of expressions.
- `c<=a+b; d<=c+a;` are expressions that are evaluated during the clock transition.
- The values assigned internally to temporary variables `c` and `d` appear on flip-flops output signal names.
- Time "freezes" at the point where the expressions are evaluated.
- Values appear on flip-flops output signal names after the evaluation.
Blocking vs. Non-Blocking

- Which describes better what you expect to see?
  - Non-blocking assignment

- Note:
  - Use non-blocking for flip-flops
  - Use blocking for combinational logic
    - Logic can be evaluated in sequence – not synchronized to clock
  - Don’t mix them in the same procedural block
Some Flip Flop Types:

```verilog
reg Q0, Q1, Q2, Q3, Q4;

// D Flip Flop
always@(posedge clock)
    Q0 <= D;

// D Flip Flop with asynchronous reset
always@(posedge clock or negedge reset)
    if (!reset) Q1 <= 0;
    else Q1 <= D;

// D Flip Flop with synchronous reset
always@(posedge clock)
    if (!reset) Q2 <= 0;
    else Q2 <= D;

// D Flip Flop with enable
always@(posedge clock)
    if (enable) Q3 <= D;

// D Flip Flop with synchronous clear and preset
always@(posedge clock)
    if (!clear) Q4 <= 0;
    else if (!preset) Q4 <= 1;
    else Q4 <= D;
```

**Note:**
Registers with asynchronous reset are smaller than those with synchronous reset
+ don’t need clock to reset
+ easier to integrate with test features.
Reset

- Reset is an important part of the control strategy
  - Used to initialize the chip to a known state
  - Distributed to registers that determine state
  - E.g. FSM state vector
  - Usually asserted on startup and reset
  - Globally distributed
  - Not a designer-controlled signal
Behavior ➔ Function

What do the following code fragments synthesize to?

```verilog
reg foo;
always @(a or b or c)
begin
  if (a)
    foo = b | c;
  else foo = b ^ c;
end

reg foo;
always @(clock or a)
begin
  if (clock)
    foo = a;
end
```
Sketch the logic being described:

```verbatim
input [1:0] sel;
input [3:0] A;
reg Y;
always@(sel or A)
    casex (sel)
        0 : Y = A[0];
        1 : Y = A[1];
        2 : Y = A[2];
        3 : Y = A[3];
        default : Y = 1'bx;
    endcase
endcase
```
Behavior ➔ Function

Sketch the truth table, and describe the logic:

```vhdl
input [3:0] A;
reg [1:0] Y;
always@(A)
  casex (A)
    4'b0001 : Y = 0;
    4'b0010 : Y = 1;
    4'b0100 : Y = 2;
    4'b1000 : Y = 3;
    default : Y = 2'bx;
  endcase
```

Note: All the statements here are mutually exclusive – truly parallel statements – no priority implied.
Don’t Cares in Synthesis

- Real logic can only take on values of 0 or 1
- Don’t cares used by Karnaugh map optimizer during synthesis to minimize the logic

```verilog
input [3:0] A;
reg [1:0] Y;
always @(A)
case x (A)
    4'b0001 : Y = 0;
    4'b0010 : Y = 1;
    4'b0100 : Y = 2;
    4'b1000 : Y = 3;
    default : Y = 2'bx;
endcase
```

```
<table>
<thead>
<tr>
<th>A[3:2]</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y[0]</td>
<td>x</td>
<td>0</td>
<td>x</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>
```

&& and ! negate | | or
Behavior ⇒ Function

Sketch the truth table, and describe the logic:

\[
\text{input } [3:0] \ A; \\
\text{reg } [1:0] \ Y; \\
\text{always@}(A) \\
\quad \text{case}x \ (A) \\
\quad \quad 4'b1xxx : Y = 0; \\
\quad \quad 4'bx1xx : Y = 1; \\
\quad \quad 4'b001x : Y = 2; \\
\quad \quad 4'b0000 : Y = 3; \\
\quad \quad 4'b0001 : Y = 0; \\
\quad \quad \text{default} : Y = 2'bx; \\
\text{endcase}
\]
Behavior ➔ Function

Sketch the logic:

```vhdl
input [2:0] A;
reg [7:0] Y;
always@(A or B or C)
  begin
    Y = B + C;
    casex (A)
      3'b1xx : Y = B - C;
      3'bx00 : Y = B | C;
      3'b000 : Y = B & C;
    endcase
  end
```

Arithmetic Unit with Priority Decoder

```
B
C
Y
A[2]
B
C
+
-
B
C
A[1].A[0]
```
Priority Logic

- If the alternatives in the case or if-then-else statement are mutually exclusive, non-priority logic is implied
  - What is synthesized?

- Which is faster, priority logic or non-priority logic?
Behavior ➔ Function

```verilog
integer    i, N;
parameter  N=7;
reg        [N:0] A;
always@(A)
    begin
        OddParity = 1'b0;
        for (i=0; i<=N; i=i+1)
            if (A[i]) OddParity = ~OddParity;
    end
```
Behavior $\rightarrow$ Function

Sketch the logic:

```vhdl
reg A, B, C, D, E, F;
always@(A or B or C or D)
begin
  E = A | B;
  if (C) then F = E; else F = A ^ B;
  E = E ^ C;
end
```
Exercises

Which alternative best describes the behavior of the logic in the following Verilog fragment. Notice the use of blocking assignment.

```verilog
reg [3:0] A, B, C;

always@(posedge clock)
begin
    B = {A[1:0], A[3:2]};
    C = A + B;
end
```

If A = 4'b1101, and B=4'b0001 before the positive edge of the clock, then after the positive edge.

A. B=4'b0011; C=4'b0001;
B. B=4'b0111; C=4'b1011;
C. B=4'b0111; C=4'b1110;
D. B=4'b0111; C=4'b0100;

Answer: D
Exercise

Which code fragment correctly captures the following logic. Notice the use of blocking assignment.

A. always@(posedge clock)
   begin
     A = B;
     B = A;
   end

B. always@(posedge clock)
   begin
     B = A;
     A = B;
   end

C. always@(posedge clock)
   begin
     C = B;
     D = A;
     B = D;
     A = C;
   end

D. always@(posedge clock)
   begin
     A = B = A;
   end

Answer: None of these!
Exercises

Implement a 2-bit Grey scale encoder: (I.e. Binary encoding of 1..4 differ by only 1 bit)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
</tr>
</tbody>
</table>

Implement hardware that counts the # of 1’s in input [7:0] A. Use a for loop
**Procedural Code**

- `always@(posedge clock)` results in what?

- Variables assigned procedurally are declared as what type?

- What type of assignment should be used when specifying flip-flops?

- When is the block evaluated?