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:

- 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
Revision

- Draw logic and a timing diagram corresponding the following code extracts:

```plaintext
// all variables are 1-bit wide

always@(posedge clock)
    A <= B | C;

always@(A)
    E = ~A;

assign D = C ^ E;
```

```
clock
  ┌───┐
  │   │
  B 0
  │   │
  C 1
  │   │
  A
  │   │
  E
  │   │
  D
```

©2012, Dr. Paul D. Franzon, www.ece.ncsu.edu/erl/faculty/paulf.html
Specifying Numbers 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:

\[ 1^{'} b1; \quad \quad 4^{'} b0; \]

\[ \text{size} \ ' \text{base value} ; \quad \text{size} = \# \text{bits}, \quad \text{HERE: base = binary} \]

\[ \text{DEFAULT is 32-bit decimal} \]

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

Examples:

- \( 10 \rightarrow 3^{'} b1 \)
- \( 8^{'} hF0 \rightarrow 8^{'} hF \)
- \( 5^{'} d11 \rightarrow 2^{'} b10 \)
Specifying Numbers

- Special Values
  - $x = \text{don't care or unknown}$
  - $z = \text{high impedance (not driven to a specific value)}$

- $x$ and $z$ are self-extending

<table>
<thead>
<tr>
<th>S</th>
<th>D</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>floating (z)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>floating (z)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- $8'hx$: xxxx xxxx
- $3'bz$: zzz
Verilog 2001 Syntax

- **Signed Values:**
  \(<size>'s<base\ format><value>\)
- The “s” denotes that a value is signed (all other values are assumed unsigned)
- This does not change the value, but it does change how certain operators deal with the value (namely, whether or not it will be sign-extended)

- **Examples:**
  - 4’hF   = 1111 in binary
  - 4’shF  = 1111 in binary (but will sometimes be interpreted as -1)
  - 6’shF  = 001111 (does not change extension rules)
Verilog 2001 Syntax

- **Signed Values:**
  \[ <\text{size}>'s<\text{base format}><\text{value}> \]

- The “s” denotes that a value is signed (all other values are assumed unsigned)

- This does not change the value, but it does change how certain operators deal with the value (namely, whether or not it will be sign-extended)

- **Examples:**
  - 4’hF = 1111 in binary
  - 4’shF = 1111 in binary (but will sometimes be interpreted as -1)
  - 6’shF = 001111 (does not change extension rules)
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’d-1?

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
```
Other Lexical Conventions

- Spaces and new lines are ignored
  - Eg. Write 8 bits as 0101 1111 to make more readable

- Verilog is Case Sensitive

- Identifiers have to start with a character (A-Z, a-z) and may only contain characters, numbers, _ and $
**Vectors**

In RTL most variables are vectors of bits

Vectors can be specified by declaring the range of bit numbers with the variable name. The form of the declaration is: 

\[ \langle \text{high} \rangle : \langle \text{low} \rangle \] <variable> ;

or 

\[ \langle \text{low} \rangle : \langle \text{high} \rangle \] <variable> ;

```verilog
wire [7:0] BYTE; // declare 8-bit data named BYTE
reg [15:0] INFO;  // declare 16-bit register named INFO
reg [0:11] DATA;  // declare 12-bit register named DATA
```

- Left bit is considered the most significant
- By convention, use \[ \langle \text{high} \rangle : \langle \text{low} \rangle \]
Specifying Parts of Vectors

Given vector declarations, it is possible to reference parts of a register (down to a single bit). The format of the reference follows the pattern `<vector>[<bit range>]`.

<table>
<thead>
<tr>
<th>Vector</th>
<th>Reference</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INFO[5]</td>
<td>// bit 5 of INFO</td>
<td></td>
</tr>
<tr>
<td>DATA[7:0]</td>
<td>// least significant byte of DATA</td>
<td></td>
</tr>
</tbody>
</table>
**Other Types**

- Arrays (1-D array of vectors) – see later section
- Integer
- Real
- Time (accessed w/ system function $\$time$)
  - Rarely used in the description of a hardware module but are frequently useful in the Test Bench.
- See Sutherland, section 7.0 for details
Procedural Blocks

Code of the type

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*. 
Blocking and Non-blocking Assignment

- Or WHY I always use "<=" for flip-flops
- Blocking ("=")
  ```
  always@(  )
  begin
    A = B;
    B = A;
  end
  ```
- Non-Blocking ("<=")
  ```
  always@(  )
  begin
    A <= B;
    B <= A;
  end
  ```
Assignment

- Assuming \( A=3; B=4; \) before execution, determine their values after execution

- **Blocking ("=")**
  ```verilog
  always@(posedge clock)
  begin
    A = B;
    B = A;
  end
  ```

- **Non-Blocking ("\<=")**
  ```verilog
  always@(posedge clock)
  begin
    A <= B;
    B <= A;
  end
  ```
Assignment

- Now draw the logic each describes
- Blocking ("=")
  ```
  always@(posedge clock)
  begin
    A = B;
    B = A;
  end
  ```
- Non-Blocking ("<=")
  ```
  always@(  )
  begin
    A <= B;
    B <= A;
  end
  ```
Examples

- Describe logic and draw timing for the following:

```vhdl
// all variables are 1-bit wide
reg B, D, E;
always@(posedge clock)
    B <= A ^ C;
always@(posedge clock)
    D <= A | C;
always@(posedge clock)
    E <= B & D;
```

<table>
<thead>
<tr>
<th>Clock</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
... Examples

- Will the following produce the same logic?

```verilog
// all variables are 1-bit wide
reg B, D, E;
always@(posedge clock)
  begin
    B <= A ^ C;
    D <= A | C;
    E <= B & D;
  end
```
... Examples

- Will the following produce the same logic?
  
  ```verilog
  // all variables are 1-bit wide
  reg B, D, E;
  always@(posedge clock)
    begin
      B = A ^ C;
      D = A | C;
      E = B & D;
    end
  ```
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
Combinational Logic

• Procedural description of combinational logic can be very useful
  • Can describe **structure** or **behavior** of logic block
    ◆ If you are just describing structure, consider continuous assignment instead
  • Always uses blocking assignment
    ◆ Example. Consider (B, C, E are outputs of flip-flops)
      
      ```
      always@(*) begin
      A <= B & C;
      D <= A & E;
      end
      
      always@(*) begin
      A = B & C;
      D = A & E;
      end
      ```
Behavior ➔ Function

What do the following code fragments synthesize to?

```vhdl
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
```
Case statement

- What about this statement?

```verilog
reg [1:0] B;
always@(*)
  case (A)
    2'b00 : B = C;
    2'b01 : B = D;
    2'b10 : B = E;
    default : B = F;
  endcase
```

Evaluate () against statements on LHS of:

- Potential matches evaluated in SEQUENCE.
- Statement on RHS executed upon match.
- Default statement executed if no other match occurs (default optional)

After statement on RHS executed, jump to end.
Case statement

- What logic behavior does it simulate? I.e. What would it build?

```verilog
reg [1:0] B;
always@(*)
case (A)
  2'b00 : B = C;
  2'b01 : B = D;
  2'b10 : B = E;
  default : B = F;
endcase
```
**Casex statement**

- Casex permits matches against z (high impedance) and x (don’t care)
- E.g.

```vhdl
reg [1:0] B;
always@(*)
  casex (A)
    2'b00 : B = C;
    2'b01 : B = D;
    2'b1x : B = E;
  endcase
```
Priority Encoder

- What does the following specify?

```verilog
always@(A or B or C)
  case(A)
    3'blxx : out = B;
    3'bx1x : out = C;
    default : out = 2'b0;
  endcase
```

- Rewrite as if-else
Demultiplexer

- Demultiplexer (demux) activates one or more outputs depending on an input and control signal.
  - Often used for Decoders
- Write Verilog for:
Decoder

- What does the following specify?

```vhdl
always@(address)
    case (address)
        2'b00 : line = 4'b0001;
        2'b01 : line = 4'b0010;
        2'b10 : line = 4'b0100;
        2'b11 : line = 4'b1000;
    endcase
```
Exercise

- Write Verilog to capture the following logic:
Exercise

- Determine the minimal 2-stage logic implementation

```verilog
reg B;
always@(*)
  casex (A)
    4'bxx01 : B = 1'b1;
    4'b0011 : B = 1'b1;
    4'b11xx : B = 1'b0;
    default : B = 1'bx;
  endcase
```
For loops

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
Sketch the logic being described:

```verilog
input [1:0] sel;
input [3:0] A;
reg Y;
always@(sel or A)
case (sel)
  0 : Y = A[0];
  1 : Y = A[1];
  2 : Y = A[2];
  3 : Y = A[3];
  default : Y = 1'bx;
endcase
```
Exercises

Sketch the truth table, and describe the logic:

```vhdl
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
```

©2012, Dr. Paul D. Franzon, www.ece.ncsu.edu/erl/faculty/paulf.html
Exercises: 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)
casex (A)
  4'b0001 : Y = 0;
  4'b0010 : Y = 1;
  4'b0100 : Y = 2;
  4'b1000 : Y = 3;
default : Y = 2'bx;
endcase
```
Behavior ➔ Function

Sketch the truth table, and describe the logic:

```verbatim
input [3:0] A;
reg [1:0] Y;
always @(A)
    case x (A)
        4'b1xxx : Y = 0;
        4'bx1xx : Y = 1;
        4'b001x : Y = 2;
        4'b0000 : Y = 3;
        4'b0001 : Y = 0;
        default : Y = 2'bx;
    endcase
```
Sketch the logic:

```verbatim
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'bx01 : Y = B | C;
        3'b000 : Y = B & C;
    endcase
end
```
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?
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
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?