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
Revision

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

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

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

always@(A)
  E = ~A;

assign D = C ^ E;
```
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;       4'b0;
size 'base value ;   size = # bits,   HERE: base = binary
```

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

Examples:

- 10
- 00 ..01010
- 3’d5
- 101
- 8’hF0
- 11110000
- 8’hF
- 00001111
- 10
- 5’d11
- 01011
- 2’b10

Default is 32-bit decimal

Unspecified high order bits are zero-filled if left most bit is 0 or 1

High order bits are truncated if the size is smaller
Special Values

- \( x \) = don’t care or unknown
- \( z \) = high impedance (not driven to a specific value)

\[
\begin{array}{ccc}
S & D & \text{out} \\
0 & 0 & \text{floating (z)} \\
0 & 1 & \text{floating (z)} \\
1 & 0 & 1 \\
1 & 1 & 0 \\
\end{array}
\]

- \( x \) and \( z \) are self-extending if they are the left most bit

- \( 8'bx0: \) xxx_xxxx0
- \( 3'bz: \) zzz

Underscore is ignored
Signed Values:
<\textit{size}>'s<\textit{base format}><\textit{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 operators (+, <<<, etc.) deal with the value (namely, whether or not it will be sign-extended after the operation)

Examples:
- $4\text{'}hF = 1111$ in binary
- $4\text{'}shF = 1111$ in binary (but will sometimes be interpreted as -1)
- $6\text{'}shF = 001111$ (does not change extension rules)
Other Lexical Conventions

- Spaces, underscores 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 $

- Verilog reserved words can not be used for variables,
  - E.g. always, module, etc.
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:

\[
[\text{<high>: <low}>] \text{<variable>} ; \\
\text{or} \quad [\text{<low>: <high}>] \text{<variable>} ;
\]

- Left bit is considered the most significant
- By convention, use \[\text{<high>:<low>}\] (Big Endian)
  - DATA[0] is the left most, most significant bit
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>]`.

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

- Arrays (1-D and 2-D arrays of vectors)
  - Example: `reg [15:0] RegisterFile [0:31]; // 32 x 16-bit vectors`
  - Note arrays can NOT be passed through input/output ports
  - In synthesizable design:
    - 1-D arrays used to model register files and un-synthesized memories
    - 2-D arrays have little relevance

- Integer // used as a loop counter
Not used in synthesis but useful in test fixture:
  - Real
  - Time (accessed w/ system function $time)

- See Sutherland, section 7.0 for details
Summary

- Lexical rules are specified so that numbers can be precisely enumerate size/base value

- All signals in synthesizable Verilog are bits or vectors of bits

- Do exercise before proceeding to next sub-module
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
  ```

  Execution of next statement "blocked" until this statement is completed. i.e Statements execute sequentially.

- **Non-Blocking (“<=“)**
  
  ```
  always@(  )
  begin
    A <= B;
    B <= A;
  end
  ```

  Execution of next statement NOT "blocked". i.e Statements execute as if they were being run in parallel.
Assignment

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

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

- Non-Blocking ("<=")
  
  ```
  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@(posedge clock)
  begin
    A <= B;
    B <= A;
  end
  ```

B has same value as A after clock event is over \(\Rightarrow\) B connected to input of A

Always use "<=" with flip-flops
Examples

- Describe logic and draw timing for the following:

```verilog
// 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;
```

```
\begin{tabular}{c c c c c}
  A & B & C & D & E \\
  0 & 0 & 0 & 0 & 0 \\
  0 & 1 & 0 & 1 & 1 \\
  1 & 0 & 1 & 0 & 1 \\
  1 & 1 & 1 & 0 & 1 \\
\end{tabular}
```

- Draw timing for the following:

```
\begin{tabular}{c c c c c}
  clock & A & B & C & D & E \\
  1 & 1 & 1 & 1 & 1 & 1 \\
\end{tabular}
```
... *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
  ```

  Yes
… 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
  ```

  ![Logic Diagram](image)

  No
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
**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@(*) always@(*)
      begin begin
        A <= B & C; A = B & C;
        D <= A & E; D = A & E;
      end end
      ```

D needs the value of A that existed BEFORE this block was triggered. i.e.
This might be the value of A from the previous clock cycle. This is not simple to work out how to obtain.

A changes then D changes. I.e. A is an input of the second AND
Registers

Some Flip Flop Types:

```verbatim
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 is a global signal intended to get the chip into a known state. It is not to be locally generated or modified.
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
Summary

- Always use non-blocking assignment ("\(<=\)"") when assigning to flip-flops
  - Even if you don’t think it is needed – Makes code easier to maintain
- Always use blocking assignment ("\(=\)"") when describing combinational logic
- Apply a global reset signal to all registers that determine machine “state”
  - FSMs, some counters, some status registers, etc.
- Reset is a global signal coming from outside the digital design
  - Not to be modified or locally generated

- Proceed to sub-module quiz and next sub-module
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
```

Diagram of Latch:
Avoid unintentional latches

Example:

```verbatim
reg foo;
always @(a or b or c)
  begin
    if (a)
      foo = b | c;
``` 

In a non-clocked procedural block, each variable has to be assigned every time the code is executed, so as not to specify memory, or an unintended latch.

a could be the output of glitchy logic \(\rightarrow\) a glitchy clock, creating unpredictable errors in the actual logic, even if the RTL worked fine pre-synthesis.

```verbatim
b
\rightarrow
\square
\rightarrow
\boxed{\text{foo}}
```

\(c\)

\(a\)
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.

Here default is needed to prevent latches.
Case statement

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

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

Diagram of a case statement with inputs A, C, D, E, F and output B.
**Casex statement**

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

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

Note: No “unknown” (x) in actual design
**Priority Encoder**

- What does the following specify?

```verilog
always@(A or B or C)
    casex(A)
        3'b1xx : out = B;
        3'bx1x : out = C;
        default : out = 2'b0;
    endcase
```

- Rewrite as if-else

```verilog
always@(A or B or C)
    if (A[2]) out = B;
    else if (A == 3'b01x) out = C;
    else out = 2'b0;
```

**Demultiplexer**

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

```
always@(*)
case (s)
  1'b0: begin
    z0 = in; z1 = 1'b0;
  end
  1'b1: begin
    z0 = 1'b0; z1 = in;
  end
endcase
```

<table>
<thead>
<tr>
<th>s</th>
<th>z0</th>
<th>z1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>in</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>in</td>
</tr>
</tbody>
</table>
**Decoder**

- What does the following specify?

```verbatim
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
```

"one-hot" decoder - only one output line is active at a time
**Truth Tables**

- If you get stuck how to specify a piece of logic, then simply write its truth table
- THEN capture truth table as a case statement
- Example: **Grey code counter**
- Truth Table: 

```
<table>
<thead>
<tr>
<th>Code:</th>
<th>NextCount</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>01</td>
</tr>
<tr>
<td>01</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>00</td>
</tr>
</tbody>
</table>
```
- Code:

```
always@(*)
    
11     : 10
10     : 00
```

```
case x (Count)
    2'b00 : NextCount = 2'b01;
    2'b01 : NextCount = 2'b11;
    2'b11 : NextCount = 2'b10;
    2'b10 : NextCount = 2'b00;
    2'bxx : NextCount = 2'bxx;
endcase
```

always@(posedge clock)
Count <= NextCount;

```
0 : 00
1 : 01
2 : 11
3 : 10
```
Exercise

- Write Verilog to capture the following logic:

```verilog
always@(*)
  case (A[1:0])
    2'b00: F = B | C;
    2'b01: F = D;
    2'b10: F = ~E[1:0];
    2'b11: F = E[3:2];
  endcase

reg [1:0] B, C, D, F;
reg [3:0] A, E;
```
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

```
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

```

\[ 00 \quad 01 \quad 11 \quad 10 \]
\[ \begin{array}{cccc}
  x & 0 & x & 1 \\
  0 & x & x & x \\
  x & x & x & x \\
  1 & x & x & x \\
\end{array} \]

\[ & & \text{&& and} & ! \text{negate} \]

\[ || \quad \text{or} \]
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

The only valid use of a for loop is to iterate through a fixed array. If in doubt, don’t use.

Unroll the for loop to interpret

Note: Parameter to permit bit-width specification.
Notes

- Only if-then-else, case, casex, and for loops are useful when specifying the behavior of combinational logic
  - Generally I used casex all the time, rather than having to decide between case and casex – casex is more general and quite useful

- Other procedural constructs – while, repeat, casez, disable – are not used in synthesizable logic

- for has very limited utility
  - If you are using to describe a sequence of steps or an action loop, you are probably using it incorrectly
Exercises

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
```

<table>
<thead>
<tr>
<th>A</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>00</td>
</tr>
<tr>
<td>0010</td>
<td>01</td>
</tr>
<tr>
<td>0100</td>
<td>10</td>
</tr>
<tr>
<td>1000</td>
<td>11</td>
</tr>
<tr>
<td>All others</td>
<td>xx</td>
</tr>
</tbody>
</table>

4-2 encoder

Note: All the statements here are mutually exclusive – truly parallel statements – no priority implied
Sketch the logic being described:

```verilog
input [1:0] sel;
input [3:0] A;
reg Y;
always@(sel or A)
  case #x (sel)
    0 : Y = A[0];
    1 : Y = A[1];
    2 : Y = A[2];
    3 : Y = A[3];
    default : Y = 1’bx;
  endcase
wire Y;
assign Y = A[sel];
// builds same logic
```

The ‘default’ here is not essential but useful to
(1) Enforce discipline preventing latches
(2) Ensures proper startup behavior (see later)

Synopsys optimizes it out in this case.
Sketch the truth table, and describe the logic:

```
input [3:0] A;
reg [1:0] Y;
always@(A)
  casex (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
```

<table>
<thead>
<tr>
<th>A</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1xxx</td>
<td>00</td>
</tr>
<tr>
<td>01xx</td>
<td>01</td>
</tr>
<tr>
<td>001x</td>
<td>10</td>
</tr>
<tr>
<td>0000</td>
<td>11</td>
</tr>
<tr>
<td>0001</td>
<td>00</td>
</tr>
</tbody>
</table>

Priority Encoder
(A[3] has priority over other bits, etc.)
Behavior → Function

Sketch the logic:

```verilog
input [2:0] A;
reg [7:0] Y;
always@(A or B or C)
    begin
        Y = B + C;
        case x (A)
            3'b1xx : Y = B - C;
            3'bx01 : Y = B | C;
            3'b000 : Y = B & C;
        endcase
    end
```

Arithmetic Unit with Priority Decoder

Starting with Y=B+C; prevents unintentional latches much like default does.
Priority Logic

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

  *A single mux*

- Which is faster, priority logic or non-priority logic?

  *Non-priority (usually)*
**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>

Could also build truth table and use a case statement.

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

```vhdl
reg [3:0] count;
integer i;
always@(A)
begin
  count = 0;
  for (i=0; i<=7; i=i+1)
    count = count + A[i];
end
```

Could also build truth table and use a case statement.
Summary - Behavioral Combinational Logic

- Common constructs
  
  always@(*) if else // simple muxes  
  always@(*)
  
  casex ( )
    
    : ;
  
  default : ;
  
  endcase
  
  // logic that considers multiple alternatives

- Less common construct

  always@(*) for (i..) // iterating through an array

- Important to prevent unintentional latches by making sure every variable is assigned no matter how the code is executed

- Goto sub-module quiz and then proceed to first submodule of 4.2