Design With Verilog

Dr. Paul D. Franzon

Outline

1. Procedural Examples
2. Continuous Assignment
3. Structural Verilog
5. Common Problems
6. More sophisticated examples

Always Design Before Coding

References

1. Quick Reference Guides
2. Smith & Franzon, Chapter 2-6, 8, Appendix A

Lexical Conventions in Verilog

Logic values:

Logic Value	Description
0	Zero, low, or false
1	One, high or true
Z or z or ?	High impedance, tri-stated or floating
X or x	Unknown, uninitialized, or Don't Care

Integers:

```
1'b1;          4'b0;
```

size 'base value ; *size = # bits,* *HERE: base = binary*
NOTE: zero filled to left

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

Examples:

```
10                    3'b1
8'hF0                 8'hF
5'd11                 2'b10
```

Procedural Blocks

Code of the type

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

is referred to as *Procedural Code* because the statements between the `begin` and the `end` are executed *procedurally,* or in order.

- Note that all variables assigned (i.e. on the left hand side) in procedural code must be of a *register data type*. Here type `reg` is used.
  - Because a 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.
  - Then the statements that follow are executed once
  - The statements in parentheses `(...)` are referred to as the ***sensitivity list.***
  - `always@(posedge clock)` executes whenever a positive edge on the clock occurs
  - `always@(value)` executes whenever any of the statements in parentheses change

... Procedural Code

Blocking vs. Non-Blocking Assignment

What is the difference between the following code segments?

```
initial
  begin
    a = 4'h3; b = 4'h4;
  end

always@(posedge clock)
  begin
    c = a + b;
    d = c + a;
  end;
```

```
initial
  begin
    a = 4'h3; b = 4'h4; c=4'h2;
  end

always@(posedge clock)
  begin
    c <= a + b;
    d <= c + a;
  end;
```

- `=` is referred to as a blocking assignment, because execution of subsequent code is blocked until this statement completes. Essentially, `c = a + b;` and `d = c + a;` are performed in series.
- `<=` is referred to as non-blocking assignment. Essentially, `c <= a + b;` and `d <=c+a;` are performed in parallel

Always Use <= when building registers to prevent possible races.

Registers

Some Flip Flop Types:

```
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
BUT it is a good idea to synchronize reset at the block level to reduce impact of noise.

Behavior ➔ Function

What do the following code fragments synthesize to?

```
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)
     if (clock)
       foo = a;
  end
```

Behavior ➔ Function

Sketch the logic being described:

```
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
```

Behavior ➔ Function

Sketch the truth table, and describe the logic:

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

Behavior ➔ Function

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'b01xx : Y = 1;
    4'b001x : Y = 2;
    4'b0000 : Y = 3;
    4'b0001 : Y = 0;
    default : Y = 2'bx;
  endcase
```

Behavior ➔ Function

Sketch the logic:

```
input [2:0] A;
reg [7:0] Y;
always@(A or B or C)
  begin
    Y = B + C;
    casex (A)
      8'b1xx : Y = B - C;
      8'b000 : Y = B | C;
      8'b001 : Y = B & C;
    endcase
  end
```

Behavior ➔ Function

```
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
```

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?

Exercises

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

Implement hardware that counts the # of 1's in input [7:0] A;

Operators

14 VERILOG HDL 2.0 REFERENCE GUIDE

11.0 Operators

- Operators perform an operation on one or two operands:
  operator operand /unary expression
  operand operator operand /binary expression
- The operands may be either net or register data types.
- The operands may be scalar, vector, or bit selects from a vector.

Usage	Description
Arithmetic Operators	
m + n	Add n to m
m - n	Subtract n from m
- m	Negate m (2's complement)
m * n	Multiply m by n
m / n	Divide m by n
m % n	Modulo (remainder) of m / n
Bitwise Operators	
~m	Invert each bit of m
m & n	AND each bit of m with each bit of n
m | n	OR each bit of m with each bit of n
m ^ n	Exclusive OR each bit of m with n
m ^~ n m ~^ n	Exclusive NOR each bit of m with n
Unary Reduction Operators	
&m	AND all bits in m together (1-bit result)
~&m	NAND all bits in m together (1-bit result)
|m	OR all bits in m together (1-bit result)
~|m	NOR all bits in m together (1-bit result)
^m	Exclusive OR all bits in m (1-bit result)
~^m ^~m	Exclusive NOR all bits in m (1-bit result)
Logical Operators	
!m	Is m not true? (1-bit True/False result)
m && n	Are both m and n true? (1-bit T/F result)
m || n	Are either m or n true? (1-bit T/F result)

Continuous Assignment

Sketch the logic being specified …

```
input [3:0] A, B;
wire [3:0] C, E;
wire D, F, G;
assign C = A ^ B;
assign D = |A;
assign E = {{2{A[3]}}, A[2:1]};
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]];
```

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:

```
wire [31:0] A;
wire        even_parity;
```

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 [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:

Etc.

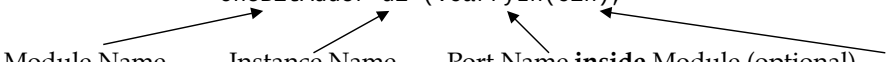
Structural Verilog

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;
```

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

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] ) );
       NOR2 U39 ( .Y(n115), .A0(\value[3] ), .A1(\value[2] ) );
       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(\value[3] ), .QBAR(n104), .CP(clock), .D(
           \value53[3] ), .E(\n54[0] ) );
       EDFF \value_reg[2]  ( .Q(\value[2] ), .CP(clock), .D(\value53[2] ), .E(
           \n54[0] ) );
       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
```

Common Problems and Fixes

Unintenional Latches

- How to detect : Found by Synopsys after "read" command
- How to fix : Make sure every variable is assigned for every way code is executed
- What happens if unfixed : Glitches on "irregular clock" to latch cause set up and hold problems in actual hardware (➔ transient failures)

Problem Code :

```
always@(A or B)
  begin
    if (A) C = ~B;
    else D = |B;
  end
```

Possible Fix :

Common Problems and Fixes

Incomplete Sensitivity List

- How to detect : After "read" command synopsys says "Incomplete timing specification list"
- How to fix : All module inputs have to appear in sensitivity list
- What happens if unfixed : Since simulation results won't match what actual hardware will do, bugs can remain undetected

Problem Code

```
always@(A or B)
  begin
    if (A) C = B ^ A;
    else C = D & E;
    F = C | A;
  end
```

Fix

Common Problems and Fixes

Unintentional Wired-OR logic

- How to detect : After "read" command synopsys says "variable assigned in more than one block"
- How to fix : Redesign hardware so that every signal is driven by only one piece of logic (or redesign as a tri-state bus if that is the intention)
- What happens if unfixed : Unsynthesizable.

This is a symptom of NOT designing before coding

Problem Code

```
always@(A or B)
  if (A) C = |B;

always@(D or E)
  if (D) C = ^E;
```

Possible Fix

Common Problems and Fixes

Improper Startup

- How to detect : Can't
- How to fix : Make sure "don't cares" are propogated
- What happens if unfixed : Possible undetected bug in reset logic

Problem Code

```
always@(posedge clock)
  if (A) Q <= D;

always@(Q or E)
  case (Q)
    0 : F = E;
    default : F = 1;
  endcase
```

```
clock : 00001111
    D : xxxxxxxx
    E : 11110000
    Q : ........
    F :
```

Fix

Debug

- How to prevent a lot of need to debug:
  - Carefully think through design before coding
  - Simulate "in your head"
- How to debug:
  - Track bug point back in design and back in time
    - Check if each "feeding" signal makes sense
  - Compare against a "simulation in your head"
  - If all else fails, recode using a different technique

Larger Examples : Linear Feedback Shift Register (fn)

```
module LFSR_FN (Clock, Reset, Y1, Y2);
input Clock, Reset;
output [7:0] Y1, Y2; reg [7:0] Y1, Y2;
parameter [7:0] seed1 = 8'b01010101;
parameter [7:0] seed2 = 8'b01110111;

function [7:0] LFSR_TAPS8_FN;
input [7:0] A;
integer N;
parameter [7:0] Taps = 8'b10001110;
reg Bits0_6_Zero, Feedback;
begin
  Bits0_6_Zero = ~| A[6:0];
  Feedback = A[7] ^ Bits0_6_Zero;
  for (N=7; N>=1; N=N-1)
    if (Taps[N-1] == 1) LFSR_TAPS8_FN[N] = A[N-1] ^ Feedback;
    else LFSR_TAPS8_FN[N] = A[N-1]; LFSR_TAPS8_FN[0] = Feedback;
end
endfunction /* LFSR_TAP8_FN */

/* Build 2 LFSRs using the LFSR_TAPS8_TASK */
always@(posedge Clock or negedge Reset)
  if (!Reset) Y1 <= seed1; else Y1 <=  LFSR_TAPS8_FN (Y1);
always@(posedge Clock or negedge Reset)
  if (!Reset) Y2 <= seed2; else Y2 <= LFSR_TAPS8_FN (Y2);
endmodule
```

LFSR

- Sketch Design

LFSR (Task)

```
module LFSR_TASK (clock, Reset, Y1, Y2);
input clock, Reset;
output [7:0] Y1;
reg [7:0] Y1;
parameter [7:0] seed1 = 8'b01010101;    parameter [7:0] Taps1 = 8'b10001110;

task LFSR_TAPS8_TASK;
input [7:0] A; input [7:0] Taps;       output [7:0] Next_LFSR_Reg;
integer N;     reg Bits0_6_Zero, Feedback; reg [7:0] Next_LFSR_Reg;
begin
  Bits0_6_Zero = ~| A[6:0];    Feedback = A[7] ^ Bits0_6_Zero;
  for (N=7; N>=1; N=N-1)
    if (Taps[N-1] == 1) Next_LFSR_Reg[N] = A[N-1] ^ Feedback;
    else Next_LFSR_Reg[N] = A[N-1];
  Next_LFSR_Reg[0] = Feedback;
end
endtask /* LFSR_TAP8_TASK */

always@(posedge clock or negedge Reset)
  if (!Reset) Y1 = seed1;
  else LFSR_TAPS8_TASK (Y1, Taps1, Y1);
endmodule
```

Register File

```
module RegFile (clock, WE, WriteAddress, ReadAddress, WriteBus, ReadBus);
input  clock, WE;
input  [4:0] WriteAddress, ReadAddress;
input  [15:0] WriteBus;
output [15:0] ReadBus;

reg [15:0]   Register [0:31];   // thirty-two 16-bit registers

// provide one write enable line per register
wire [31:0] WElines;
integer i;

// Write '1' into write enable line for selected register
assign WElines = (WE << WriteAddress);

always@(posedge clock)
    for (i=0; i<=31; i=i+1)
      if (WElines[i]) Register[i] <= WriteBus;

assign ReadBus  =  Register[ReadAddress];
endmodule
```

Register File

- Sketch Design

Sample Problem

- Accumulator:
  - Design an 8-bit adder accumulator with the following properties:
  - While 'accumulate' is high, adds the input, 'in1' to the current accumulated total and add the result to the contents of register with output 'accum_out'.
  - When 'start' is high ('accumulate' will be low) feed 'in1' into accumulator
  - The 'overflow' flag O/P goes high if the adder overflows (unsigned numbers) (don't register)


```
Module accumulator (accumulate, clear, in1, accum_out,
overflow);

input           accumulate, clear;
input [7:0]   in1;
output [7:0] accum_out;
output          overflow;

reg  [7:0] accum_out;
wire  [7:0] accum_in;
wire carry, overflow, enable;

Fourbitadder u1 (1'b0, in1[3:0], accum_out[3:0],
result[3:0], carry);
Fourbitadder u1 (carry, in1[7:4], accum_out[7:4],
result[3:0], overflow);

assign enable = accumulate || clear;

assign accum_in = clear ? 8'b0 : result;

always@(posedge clock)
  if (enable) accum_out <= accum_in;
endmodule
```

Verilog

- Hint: `{Cout, sum} = A + B;`
  - VL recognizes that adding n-bit numbers gives an n+1 bit result

```
module Accumulator (clock, start, accumulate, in1, accum_out,
   overflow);
input clock, start, accumulate;
input [7:0] In1;
output [7:0] accum_out;
output overflow;
reg [7:0] next_accum_out, accum_out;
reg overflow;
always@(posedge clock)
  accum_out <= next_accum_out;
always@(start or accumulate or in1 or accum_out)
  begin
    overflow = 1'b0;
    if (start) next_accum_out = in1;
    else if (accumulate)
      {overflow, next_accum_out} = accum_out + in1;
    else next_accum_out = accum_out;
  end
endmodule
```

Summary

- What are the two HDL designers "mantra's" so far?
- What are the three basic VL constructs?
- What is structural VL used for?
- What are 3 common problems?