3. Verilog I

Dr. Paul D. Franzon

Major units within this module

1. Introduction HDL-based Design with Verilog
2. A complete example: count.v
   Design
3. A complete example: count.v
   Verification → Synthesis
4. Further examples

3.2 A complete example: count.v

Dr. Paul D. Franzon

Outline

1. Design before coding
2. count.v example
3. How verilog works internally

References

1. Quick Reference Guides
2. Ciletti, Ch. 4, Appendix I.
3. Smith & Franzon, Chapter 2-6

Attachments Required : Standard Synthesis Script (count.dc)

See Course Outline for further list of references

Objectives and Motivation

Objectives:

- Understand how to execute the mantra "Always Design Before Coding".
- Understand how Verilog models the intrinsic parallelism of hardware.

Motivation:

- "put it all together" in big picture
- Start understanding how HDL models intrinsic parallelism of hardware UNLIKE C
  - Useful to understand basics of "what is under the hood"

References

- Sutherland quick reference guide : Summary of main statement types
  http://www.sutherland-hdl.com/reference-guides.php
- Tutorial 1: Goes through tool usage with this example
- Ciletti:
  - Section 4.1 : Basic Verilog Structures
  - Appendices: A-I : Verilog features
- Smith and Franzon:
  - Section 6.5 : Follows this example

Outline

1. Design before coding
2. Steps in design
3. Counter Example – Verilog 95
4. Counter Example – other coding styles
4. Counter Example – Verilog 2001
6. How Verilog operates internally

Outline

1. Design before coding
2. Steps in design
3. Counter Example – Verilog 95
4. Counter Example – other coding styles
5. Counter Example – Verilog 2001
6. How Verilog operates internally

Mantra #2

- The most important one for this course

ALWAYS DESIGN BEFORE CODING

- Why?
  - Must code at Register Transfer Level
  - ➔ Registers and "transfer" (combinational) logic must be worked out before coding can start

Design Before Coding

- Automatic synthesis does NOT relieve you of logic design
- It does relieve you of:
  - Logic optimization
  - Timing calculations and control
  - In many cases, detailed logic design
- If you don't DESIGN BEFORE CODING, you are likely to end up with the following:
  - A very slow clock (long critical path)
  - Poor performance and large area
  - Non-synthesizable Verilog
  - Many HDL lint errors

Avoid Temptation!

- Temptation #1:
  - "Verilog looks like C, so I'll write the algorithm in C and turn it into Verilog with a few always@ statements"

- Usual results:
  - Synthesis problems, unknown clock level timing, too many registers

Avoid Temptation! (cont'd)

- Temptation #2
  - "I can't work out how to design it, so I'll code up something that looks right and let Synthesis fix it"

- Usual result
  - Synthesis DOES NOT fix it

Avoid Temptation! (cont'd)

- Temptation #3
  - "Look at these neat coding structures available in Verilog, I'll write more elegant code and get better results"
- Usual result of temptation #3
  - Neophytes : Synthesis problems
  - Experts: Works fine but does not usually give a smaller or faster design + makes code harder to read and maintain
- A better logic structure, not better code gives a better design

Outline

1. Design before coding
2. Steps in design
3. Counter Example – Verilog 95
4. Counter Example – other coding styles
5. Counter Example – Verilog 2001
6. How Verilog operates internally

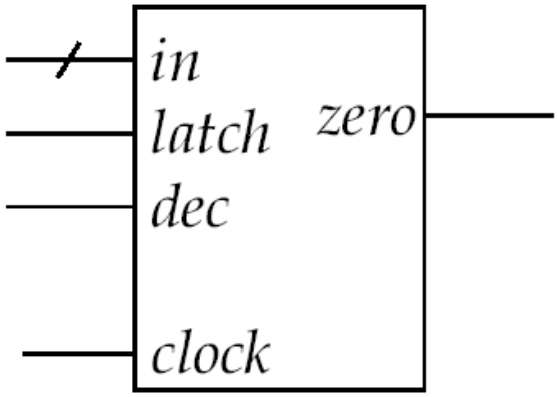
Design Before Coding

Steps in Design

1. Work out the hardware algorithm and overall strategy
2. Identify and name all the registers (flip-flops)
   - Determine system timing while doing this
3. Identify the behavior of each "cloud" of combinational logic
4. TRANSLATE design to RTL
5. Verify Design
6. Synthesize Design

Design Example: Count Down Timer

- Specification:
  - 4-bit counter
  - count value loaded from `in' on a positive clock edge when `latch' is high, latch has priority.
  - count value decremented by 1 on a positive clock edge while `dec' is high.
  - decrement stops at 0
  - `zero' flag active high whenever count value is 0



What NOT To Do

- Coding before design:

```
always@(posedge clock)
for (value=in;value>=0;value--)
  if (value==0) zero = 1
  else zero = 0;
```

- OR:

```
always@(posedge clock)
for (value=in;value>=0;value--)
    @(posedge clock)
  if (value==0) zero = 1
  else zero = 0;
```

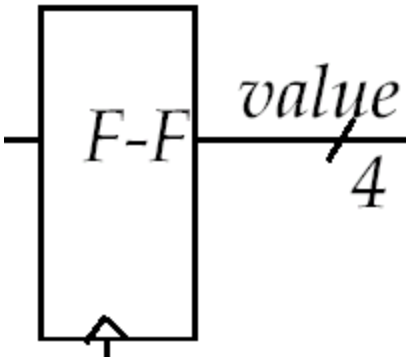
Strategy

1. Work out the hardware algorithm and overall strategy

- Strategy:
  - Load ‘in’ into a register
  - Decrement value of register while ‘dec’ high
  - Monitor register values to determine when zero

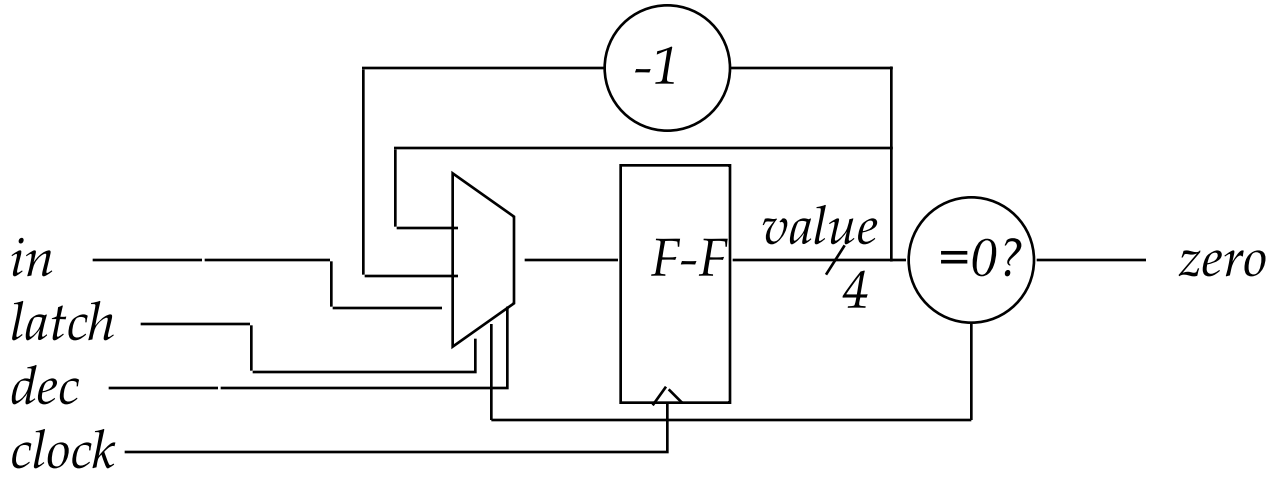
Design

2. Identify and name all the registers (flip-flops)



Design (cont'd)

3. Identify the behavior of each "cloud" of combinational logic



4. TRANSLATE design to RTL

```
module counter (clock, in, latch, dec, zero);

input        clock;  /* clock */
input [3:0] in;      /* starting count */
input        latch;  /* latch `in' when high */
input        dec;    /* decrement count when dec high */
output       zero;   /* high when count down to zero */

reg [3:0] value;   /* current count value */
wire      zero;

always@(posedge clock)
  begin
    if (latch) value <= in;
    else if (dec && !zero) value <= value - 1'b1;
  end

assign zero = (value == 4'b0);
endmodule /* counter */
```

Features in Verilog Code

Note that it follows the hardware design, not the `C' specification

Multibit variables:

```
reg [3:0] value;
```

4-bit `signal' [MSB:LSB] i.e. value[3] value[2] … value[0]

BIG ENDIAN

Specifying constant values:

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

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

Procedural Block:

```
always@(     )
  begin
  |
  |
  v
  end
```

*Executes whenever variables in **sensitivity list ( )** change value change as indicated*

Usually statements execute in sequence, i.e. procedurally
***`begin` … `end`** only needed if more than one statement in block*

Design Example ... Verilog

- Continuous Assignment:

  `assign` is used to implement combinational logic directly

Questions

1. When is the procedural block following the `always@(posedge clock)` executed?

   Whenever clock 0 -> 1

2. When is 'zero' evaluated?

   Whenever any variable on the RHS changes.

3. How is a comment done?

   // or / … */*

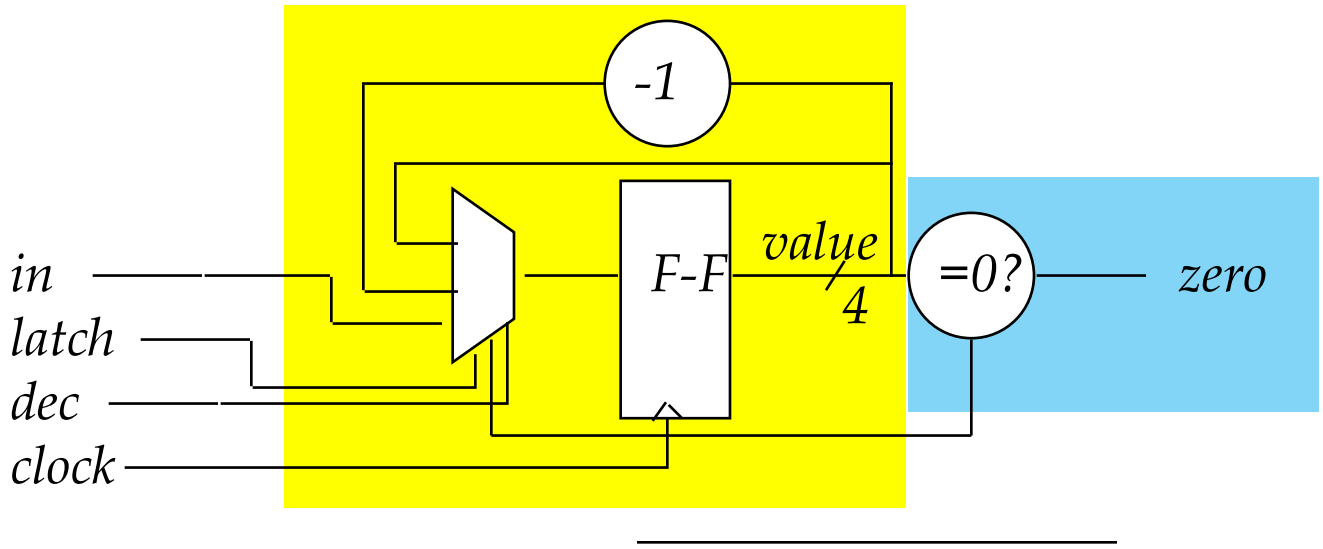
4. What does `1'b1` mean?

   1-bit constant, value 1

5. What does `reg [3:0] value;` declare?

   4-bit variable, procedurally assigned.

Behavior ➔ Function




```
always@(posedge clock)
  begin
    if (latch) value = in;
    else if (dec && !zero) value = value - 1'b1;
 end
```

```
assign zero = (value == 1'b0);
```

Sub-module Summary

- Design BEFORE coding
- After design
  - Registers + input mux ➔ always@(posedge clock) if-else or case
  - Combinational logic ➔ assign or always@(*) procedural block

Proceed to quiz and onto next sub-module

Outline

1. Design before coding
2. Steps in design
3. Counter Example – Verilog 95
4. Counter Example – other coding styles
5. Counter Example – Verilog 2001
6. How Verilog operates internally

Misc. Alternative Coding

```
module counter ( clock, in, latch, dec, zero);
   // Simple down counter with zero flag
   input           clock;  /* clock */
   input [3:0]     in;     /* starting count */
   input           latch;  /* latch `in' when high */
   input           dec;    /* decrement count when dec high */
   output          zero;   /* high when count down to zero */

   reg  [3:0]    value;  /* current count value */
   wire          zero;
   wire [3:0]    value_minus1;
   reg  [3:0]    mux_out;

   // Count Flip-flops with input multiplexor
   always@(posedge clock)begin
        value <= mux_out;
   end

   always @(*) begin
      if(latch) begin
           mux_out <= in;
      end
      else if(dec && !zero) begin
           mux_out <= value_minus1;
      end
      else begin
           mux_out <= value;
      end
   end
   assign value_minus1 = value - 1'b1;
   // combinational logic for zero flag
   assign zero = ~|value;
endmodule /* counter */
```

"Simplified" style

- *Clearly separates comb. Logic from FFs*
- *Use if you get confused*

Alternative Coding

```
module counter ( clock, in, latch, dec, zero);
   // Simple down counter with zero flag
   input           clock;  /* clock */
   input [3:0]     in;     /* starting count */
   input           latch;  /* latch `in' when high */
   input           dec;    /* decrement count when dec high */
   output          zero;   /* high when count down to zero */

   reg [3:0]  value;  /* current count value */
   reg        zero;
   // register 'value' and associated input logic
   always@(posedge clock) begin
       if (latch) value <= in;
       else if (dec && !zero) value <= value - 1'b1;
   end
   // combinational logic to produce 'zero' flag
   always@(value) begin
       if(value == 4'b0)
          zero = 1'b1;
       else
          zero = 1'b0;
   end
endmodule /* counter */
```

Outline

1. Design before coding
2. Steps in design
3. Counter Example – Verilog 95
4. Counter Example – other coding styles
4. Counter Example – Verilog 2001
6. How Verilog operates internally

Verilog 2001 Version

```
module counter (     input          clock,
                     input   [3:0]  in,
                     input          latch,
                     input          dec,
                     output  reg    zero);
   /* current count value */
   reg [3:0] value;

   always@(posedge clock) begin
       if (latch)
           value <= in;
       else if (dec && !zero)
           value <= value - 1'b1;
   end

   always@(*) begin
      if(value == 4'b0)
         zero = 1;
      else
         zero = 0;
   end
endmodule /* counter */
```

Combine Port List; Port Declarations; and Variable Declarations

Automatic sensitivity list (ONLY for combinational logic)

Sub-module Summary

- Different coding styles:
  - Combining mux input with register in one block
  - Clearly separating registers and all combinational logic
  - Simple logic can be expressed either in a procedural block or in an assign statement
    - Assign limited to ONE line
  - Synthesis results do not depend on coding style
- Verilog 2001
  - Improves coding efficiency
    - Combining ports and declarations
    - always@(*)
  - I recommend just using this style

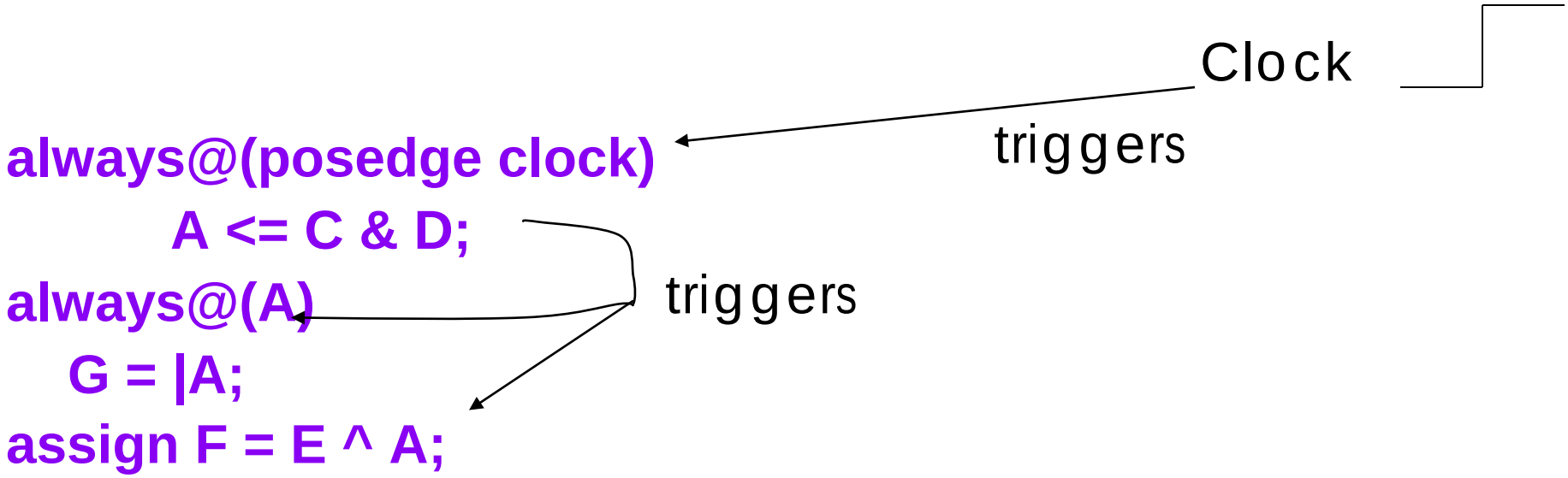
Please proceed to quiz and next sub-module

Outline

1. Design before coding
2. Steps in design
3. Counter Example – Verilog 95
4. Counter Example – other coding styles
4. Counter Example – Verilog 2001
6. How Verilog operates internally

Intrinsic Parallelism

- How Verilog models the intrinsic parallelism of hardware



Want the appearance that F and G change at the same time – How?

Intrinsic Paralellism

- Algorithm for

  always@(A) G = |A;

  assign F = E ^ A;

when A changes:

In same time "step" the following steps are performed internally and sequentially:

nextG = |A;

nextF = E ^ A;

At end of time step:

G = nextG; F=nextF;

Simulation time is "frozen during evaluation

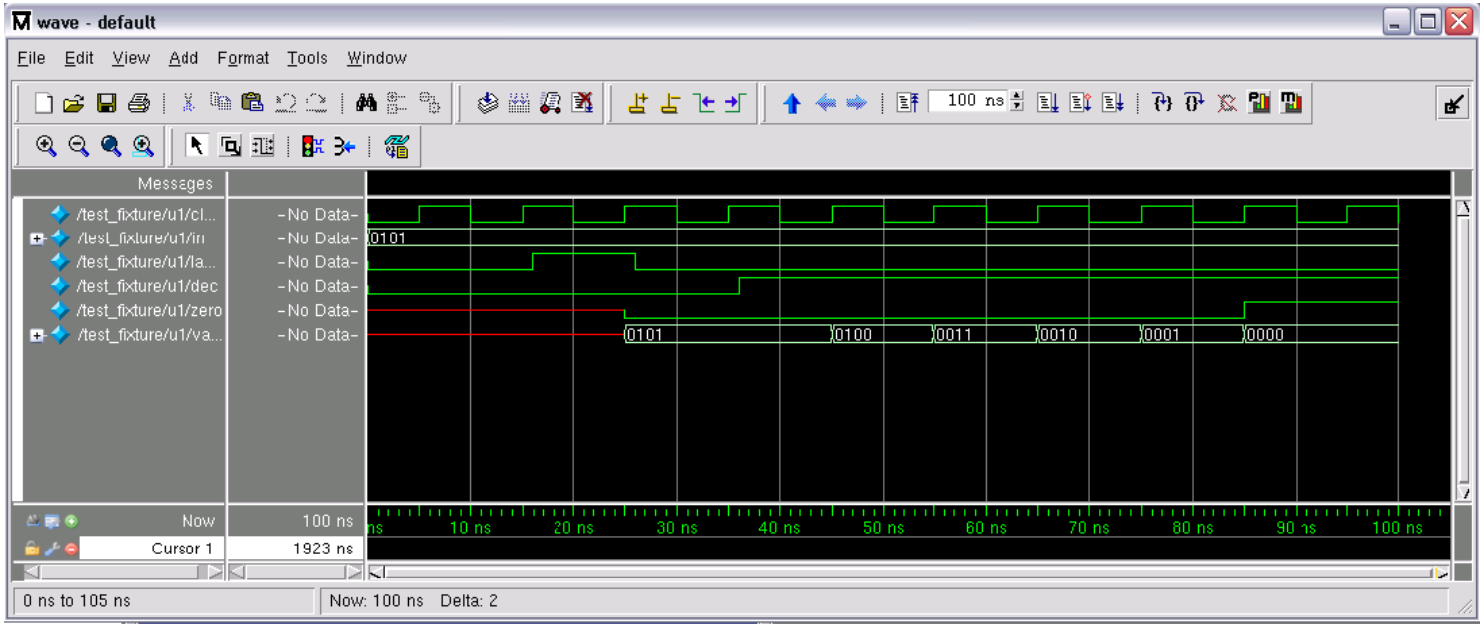
In waveform viewer see G and F changing simultaneously

"Hidden" internal variables used for sequenctial evalution

Actual variables are assigned once all known

Timing

- The RTL only specifies the clock level timing, logic has zero delay
- The internal logic delays are worked out (and optimized) during synthesis
- Pre-synthesis, many signals will look like they are changing right on each clock edge
  - Think of them changing just after the clock edge



Sub-module Summary

- Verilog operates as follows:
  - Statement and block triggers are stored
    - E.g. always@ contents and RHS of assign's
  - When a set of statements and blocks are triggered time is frozen while they are all evaluated
    - Internal variables are used during this evaluation
    - Visible variables are updated at the end of the evaluation
    - This happens in one instant of simulation time, as it appears on the screen

Proceed to the quiz and next unit