3. Verilog I
Dr. Paul D. Franzon

Sub-Modules
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
3.1 Introduction to Design With Verilog

Dr. Paul D. Franzon

Outline
1. HDL-based Design Flow
2. Introduction to the Verilog Hardware Description Language
3. Main features used in synthesizable design

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
Course “Mantras”

- One clock, one edge, Flip-flops only
- Design BEFORE coding
- Behavior implies function
- Clearly separate control and datapath
Objectives

- Describe how the behavior of a flip-flop can be captured in both Verilog and VHDL.
- Describe the structure of a Verilog module.
- Understand how the behavior of a combinational logic block can be captured with an “always@(A or B)” style of procedural block.
- Understand how combinational logic can be captured in a continuous assignment statement.
- Understand that the purpose of my mantra “Always Design Before Coding”.
Motivation

- Starting to get into core of this course
  - How to design using HDLs
- The complexity is NOT in the language but how to use it
  - Design before coding
    - Each design portion becomes a portion of code
  - Behavior implies function
    - Expected behavior of a piece of HDL must match behavior of corresponding logic
References

- Sutherland Quick Reference Guide
- Ciletti:
  - Sections 5.1 – 5.9: Go into more detail than I do (for now)
- Smith & Franzon:
  - Section 6.4: teaches these basic constructs
Purpose of HDLs

- **Purpose of Hardware Description Languages:**
  - Capture design in Register Transfer Language form
    - i.e. All registers specified
  - Use to simulate design so as to verify correctness
  - Pass through Synthesis tool to obtain reasonably optimal gate-level design that meets timing
  - Design productivity
    - Automatic synthesis
    - Capture design as RTL instead of schematic
    - Reduces time to create gate level design by an order of magnitude

- **Synthesis**
  - Basically, a Boolean Combinational Logic optimizer that is timing aware
Basic Verilog Constructs

- **Flip-Flop**
  - Behavior:
    - For every positive edge of the clock $Q$ changes to become equal to $D$
  - Write behavior as “code”

- **always@()**
  - Triggers execution of following code block
  - () called “sensitivity list”
    - Describes when execution triggered
**Mantra #3**

- **Behavior implies function**
  - Determine the behavior described by the Verilog code
  - Choose the hardware with the matching behavior

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

- **Code behavior:**
  - Q re-evaluated every time there is a rising edge of the clock
  - Q remains unchanged between rising edges
- **This behavior describes the behavior of an edge-triggered Flip-flop**
Verilog example

- What is the behavior and matching logic for this code fragment?

  ```verilog
  always@(clock or D)
    if (clock)  Q <= D;
  ```

- Hint: `always@(foo or bar)` triggers execution whenever foo or bar changes
**Flip-Flops**

- Every variable assigned in a block starting with

  ```vlsilog
  always@(posedge clock) or always@(negedge clock)
  ```

  becomes the output of an edge-triggered flip-flop

- This is the only way to build flip-flops
module flipflop (D, clock, Q);
input D, clock;
output Q;
reg Q;
always@(posedge clock)
begin
    Q <= D;
end
endmodule
VHDL model for Flip-flop

entity flipflop is
  port (clock, D:in bit; Q: out bit);
end flipflop;
architecture test of flipflop is
begin
  process
  begin
    wait until clock'event and clock = '1';
    Q <= D;
  end process;
end test;
Verilog vs. VHDL

- **Verilog**
  - Based on C, originally Cadence proprietary, now an IEEE Standard
  - Quicker to learn, read and design in than VHDL
  - Has more tools supporting its use than VHDL
Verilog vs. VHDL

- VHDL
  - VHSIC (Very High Speed Integrated Circuit) Hardware Description Language
  - Developed by the Department of Defense, based on ADA
  - An IEEE Standard
  - More formal than Verilog, e.g. Strong typing
  - Has more features than Verilog
Verilog vs. VHDL (cont’d)

- In practice, there is little difference
  - How you design in an HDL is more important than how you code
  - Can shift from one to another in a few days
Verilog Combinational Logic

- Combinational Logic Example
- How would you describe the behavior of this function in words?
- And in Code?
Behavior ➔ Function

always@(a or b or c)
if (a) foo = b^c;
else foo = b | c;

- All logical inputs in sensitivity list
- If; else ➔ Multiplexor
- Behavior = whenever input changes, foo = mux of XOR or OR
- Same behavior as combinational logic
**Procedural Blocks**

- Statement block starting with an “always@” statement is called a procedural block

- Why?
  - Statements in block are generally executed in sequence (i.e. procedurally)
**Alternative Coding Style for CL**

- Verilog has a short hand way to capture combinational logic
- Called “continuous assignment”

```verilog
assign foo = a ? b^c : b | c;
```

LHS re-evaluated whenever anything in RHS changes

```verilog
f = a ? d : e;  same as “if (a) f=d else f=e;
```
Input Logic to Flip-Flops

- Can include some combinational logic in FF procedural block

```plaintext
always@ (posedge clock)
  if (a) foo <= c;
  else foo <= b;
```

- Behavior ➔ function
  - foo is re-evaluated on every clock edge ➔ output of FF
  - If;else ➔ MUX
RTL Coding Styles

● That’s it!

● Three coding styles
  ● `always@(??edge clock)` ➔ FFs and input logic
  ● `always@(*)` ➔ Combinational logic (CL) specified by its behavior
  ● `assign a = ....` ➔ Continuous Assignment CL specified as structure

● The hard part is NOT coding but DESIGN
Exercise

What does each piece of logic specify?

Flip Flops and Associated Input Logic

always@(posedge clock)
case (sel)
  0 : E <= D + B;
  1 : E <= B;
endcase

Simple Combinational Logic

assign H = C | F;

Complex Combinational Logic

always@(A or B or C)
case (A)
  2'b00 : D = B;
  2'b01 : D = C;
  2'b10 : D = 1'b0;
  2'b11 : D = 1'b1;
endcase