Verilog

- Hardware Description Language
- Design process
  - Think
  - Draw diagram of hardware, figure out where logic and registers go, choose signal names carefully
  - Think
- Then…
  - Write verilog
  - Test it
Verilog

- You’ll write far better verilog if you think of it differently than a standard programming language
  - A way to code an algorithm
  - Often more elegant to use the finer features of the language
- Hardware description language
  - A way to code hardware!
  - It is far better to use only the most basic structures of the language
    - Synthesis tools will have fewer opportunities to interpret (alter) what you really want
    - Less-used CAD tools (such as place & route, design rule checking (DRC), layout versus schematic (LVS), formal verification, automatic test pattern generation (ATPG), etc.) often do not work properly with uncommon language constructs

Writing Efficient Code

- Think about what kind of hardware will result from some particular code even if the code looks simple
  
  if (x<y) begin
  a = 4'b0001;
  end

- An inequality requires a slow carry-propagate subtractor. It is simplified since the sum bits do not need to be computed, but the slow carry-out of the entireadder is needed
Verilog Simulator Tools

- Cadence
  - NC Verilog – what we will use
  - Simvision – waveform viewer
  - Verilog XL – slower, but faster start up time as it doesn’t compile before running
    - Often gives different (helpful!) and slightly more descriptive (helpful!) error messages than NC Verilog

- Synopsys
  - VCS – similar to NC Verilog
  - Virsim – waveform viewer and environment

- Many others…

Verilog vs. VHDL

- Verilog
  - Invented in 1983 at Automated Integrated Design Systems (later Gateway Design Automation) which was purchased by Cadence in 1990. It was transferred into the public domain in 1990 and it became IEEE Std. 1364-1995, or Verilog-95.
  - Strong similarities to C
  - Seems to be more commonly used in high-tech companies

- VHDL (VHSIC Hardware Description Language)
  - Published in 1987 with Dept. of Defense support as IEEE Std. 1076-1987. Updated in 1993 as IEEE Std. 1076-1993, which is still the most widely-used version.
  - Strong similarities to Ada
  - The only (?) HDL language used in government and defense organizations, and seems to be more often used in east-coast companies. Widely taught in universities ↔ used in textbooks—who started it?!
Verilog

• Modules are basic building blocks
  
  module abc (in1, in2, out);
  input in1;
  input in2;
  output out;
  <body of module>
  endmodule

• “Hardware” blocks vs. “Testing” blocks
  – Hardware verilog: only simplest, cleanest code
  – Testing verilog: anything is fine

• Main ways to do logic
  1) wires, assign statements
  2) registers, always blocks

wire, assign

• Picture “always active” hardwired logic
• For now, declare all wires
  wire a, b;
  wire out;

  \[ a \& b \rightarrow out \]
**wire, assign**

- Example:
  ```vhdl
  wire out;
  assign out = a & b;
  ```

  ![Diagram](image1)

  ```vhdl
  wire [3:0] c, d;
  wire [4:0] sum;  // sum one bit wider
  assign sum = {c[3],c} + {d[3],d};
  ```

  ![Diagram](image2)
reg, always

- Picture a much more general way of assigning “wires” or “nodes”
- Use “if/else” and “case” statements
- Can, but don’t use “for” in logic blocks (testing blocks ok)
- Sequential execution – statements execute in order

---

reg, always

- Example:
  ```
  reg out;
  always @(a or b) begin
    out = a & b;
  end
  ```

```
  a
  &
  b
  
  out
```
**reg, always**

- Example, 2-input multiplexer:
  ```vhdl
  reg out;
  always @(a or b or s) begin
    out = s ? b : a;
  end
  ```

**reg, case**

- Example using case:
  ```vhdl
  reg out;
  wire [1:0] in;      // "in" could be wire or reg
  always @(in or a) begin
    case (in)
      2'b00: begin
        out = s;
      end
      2'b11: begin
        out = a;
      end
      default: begin
        out = 1'b0;  // zero
      end
    endcase
  end // end of always block
  ```
Instantiating Flip-Flops/Registers

• How to build a FF/register (not the best one)
  ```verilog
  reg a;
  always @(posedge clk) begin
    a = a_c;
  end
  ```

  • The “=“ is a “blocking assignment” which causes the simulator to “block” on an assignment until the operation is completed

  • It causes a possible race condition
    ```verilog
    reg b, c;
    always @(posedge clk) begin
      b = a;
      c = b;
    end
    ```
    - In this case, a races to c in one cycle!

• The solution is to use a “non-blocking assignment” written with a “<=“ which causes the simulator to schedule all assignments at a particular point in time and perform them all simultaneously

  • In this case, the registers perform as normal FFs behave without a race
    ```verilog
    reg b, c;
    always @(posedge clk) begin
      b <= a;
      c <= b;
    end
    ```
Verilog Register Assignments

• Rule #1 (always follow in 281):
  For combinational logic blocks, use blocking assignments (“=“)

  // OR gate
  always @(a or b) begin
    c = a | b;
  end

Verilog Register Assignments

• Rule #2 (always follow in 281):
  For flip-flops (registers), use non-blocking assignments (“<=“)

  always @(posedge clk) begin
    sum      <= #1 c_sum;
    r_product <= #1 product;
  end
  – Add “#1” to give one unit of clock-to-Q delay to increase waveform readability
    • This does, however, produce a warning during synthesis, but it can be ignored
      (the only warning that can be automatically ignored!)
Register Naming Conventions

- It is helpful to have conventions for signal names
  - Easier for others to understand your code
  - Easier for YOU to understand your code
- \_c – input to a register (e.g., \textit{sum}_c)
- \_r – output of a register (e.g., \textit{sum}_r)
- Make it a suffix of signal names

\[ \text{sum}_c \rightarrow \text{sum} \]
\[ \text{sum} \rightarrow \text{sum}_r \]

Register Naming Conventions

- Another possibility I have seen used in industry is to make it a prefix
- \_c* – input to a register (e.g., \textit{c\_sum})
- \_r* – output of a register (e.g., \textit{r\_sum})

\[ \text{c\_sum} \rightarrow \text{sum} \]
\[ \text{sum} \rightarrow \text{r\_sum} \]
Avoid Inferring State 
For Combinational Circuits

1. Assign (specify) all regs in all paths through every always block
2. Include all input variables in sensitivity list!
3. Use default values immediately after entering always block
   - Greatly reduce chance of bugs if you do this
   - Setting output to x in the default of a case statement can help debugging, but may also cause warnings with some CAD tools

Avoid Inferring State 
For Combinational Circuits

- Common mistake:
  - always @(a) begin
    out = a & b;
  end
  - out updates only when a changes!
  - Synthesis tools and lint checkers will give a warning
Good Style With *regs* and *case* statements

- Example combinational circuit with output *c_freq*
  - Always declare default values at beginning of always block
  - If helpful, declare default values in case statements

```vhdl
always @(freq or xyz or abc) begin
  //defaults
  c_freq = freq;   // hold previous value in this case
  if (xyz==4'b0010) begin
    c_freq = abc;
  end

  case (freq) begin
    3'b000:  c_freq = abc;
    3'b001:  c_freq = abc+3'b001;
    default: c_freq = 3'bxxx;   // error case
  endcase
end
```

B. Baas, © 2011