VERILOG I

Verilog

• Verilog is a Hardware Description Language (HDL)
• Design process
  – Think
  – Draw diagram of hardware, figure out where logic and registers go, choose signal names carefully
  – Think
  – Then...
    • Write verilog
    • Test it

“A man who carries a cat by the tail learns something he can learn in no other way” - Samuel Clemens
Verilog

- You’ll write far better verilog if you think of it differently than a standard programming language which:
  - Is a way to code an algorithm
  - Often results in a more elegant solution when the programmer uses the finer features of the language
- On the other hand, a hardware description language:
  - A way to code hardware
  - Results in a far better solution when the designer uses only the most basic structures of the language
  - Synthesis tools will have fewer opportunities to interpret (destroy) the circuit 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
  
  ```verilog
  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 entire adder is needed
- Think about the hardware you want and what you’ll get with the verilog you write
  - Example: A 4x upsampler could be built with a 4-input mux; why not use a much-simpler AND?
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
  
  ```verilog
  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

1) wire, assign

- Picture “always active” hardwired logic
- For now, declare all wires
  ```verilog
  wire a, b;
  wire out;
  ```
**wire, assign**

- Example:
  ```verilog
c
d
wire out;
assign out = a & b;
```

- Example, multibit operands:
  ```verilog
c, d
wire [3:0] c, d;
wire [4:0] sum; // sum one bit wider
assign sum = {c[3],c} + {d[3],d};
```

```verilog
+ 
```

sum
2) \textit{reg, always}

- Picture a much more general way of assigning "wires" or "nodes"
- Use "if/else" and "case" statements are permitted
- You could, but don’t use "for loops" in logic blocks (use in testing blocks is ok)
- Sequential execution – statements execute in order

\textbf{Example:}

```verilog}
reg out;
always @(a or b) begin
    out = a & b;
end
```

\begin{center}
\begin{tikzpicture}
\node (a) at (0,0) {$a$};
\node (b) at (0,-1) {$b$};
\node (out) at (1,0) {$out$};
\draw (a) -- (out);
\draw (b) -- (out);
\node at (1,-0.5) {$\&$};
\end{tikzpicture}
\end{center}
**reg, always**

- Example: 2-input multiplexer:
  ```vhdl
  reg out;
  always @(a or b or s) begin
    if (s == 1'b0) begin
      out = a;
    end
    else begin
      out = b;
    end
  end
  ```

**Example:**

- Figure: A 2-input multiplexer schematic diagram.

**reg, always**

- Example: 2-input multiplexer (simpler but less clear way of writing if/then/else called "inline if" or "conditional operator" which is also found in many computer languages):
  ```vhdl
  reg out;
  always @(a or b or s) begin
    out = s ? b : a;
  end
  ```

**Example:**

- Figure: A 2-input multiplexer schematic diagram.
**reg, case**

- Example using case:
  ```verilog
  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
  ```

**Concurrency**

- All circuits operate independently and concurrently
  - Different from most programming paradigms
- This is natural if we remember the best Verilog describes hardware

![Circuits Diagram](image)
Concurrent Operation

- You should think of Verilog modules as operating on independent circuits (remember hardware orientation).

```verilog
always begin
    a = (b&c) | d; // 5-unit delay
    a = ~a;
    #5;
end

always begin
    f = ~(g ^ h); // 7-unit delay
    f = ~f;
    #7;
end
```

---

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 makes a race condition possible
  ```verilog
  reg b, c;
  always @(posedge clk) begin
    b = a;
    c = b;
  end
  ```
  - In this case, a races to c in one cycle!
Instantiating Flip-Flops/Registers

- 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 EEC 281): For combinational logic blocks, use blocking assignments (“=”)

```verilog
// OR gate
always @(a or b) begin
  c = a | b;
end
```
Verilog Register Assignments

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

```verilog
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!)

Signal 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
- Add a suffix to signal names to indicate they are from an earlier pipeline stage or a later pipeline stage
  - *_c – input to a register (e.g., sum_c)
  - *_r – output of a register (e.g., sum_r)

- (_c) combinational logic version of sum
- (_r) registered version of sum

B. Baas, EEC 281
Signal Naming Conventions

- Another possibility I have seen used in industry is to make it a prefix but this has the possibly-negative feature that associated signals are not adjacent when sorted alphabetically
  - $c_\ast$ – input to a register (e.g., $c_{\text{sum}}$)
  - $r_\ast$ – output of a register (e.g., $r_{\text{sum}}$)

\[ \begin{align*}
  c_{\text{sum}} & \quad \text{sum} \\
  \text{sum} & \quad r_{\text{sum}}
\end{align*} \]

Signal Naming Conventions

- If a signal is pipelined across multiple pipe stages, it is probably a good idea to indicate that—with a suffix such as \_pipeX for example
  - $inA$ – signal in pipeline stage 1

\[ \begin{align*}
  inA & \quad inA_{\text{pipe}2} \quad inA_{\text{pipe}4} \\
  \quad & \quad inA_{\text{pipe}3} \quad inA_{\text{pipe}5}
\end{align*} \]
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. Set 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

• Common mistake:
  - always @(a) begin
    
    out = a & b;
    
  end
  - out updates only when a changes!
  - Synthesis tools and lint checkers will give a warning
  - Verilog 1364-2001 allows the use of the always @(*)
    
    or
  
    always @*
  
  construct which eliminates this type of bug, but it is not supported by all modern CAD tools
Good Style With *regs* and *case* statements

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

```vhdl
// this "always" block instantiates the combinational logic
always @(freq or xyz or abc) begin
  // defaults
  c_freq = freq;   // hold previous value in this case
  // main logic block
  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
// this "always" block instantiates the register
always @(posedge clock1) begin
  freq  <= #1   c_freq;
end
```

B. Baas, ECE 281