VERILOG: INTRODUCTION AND COMBINATIONAL LOGIC
Verilog

• Verilog is a *Hardware* Description Language (HDL)
• You’ll design far better hardware if you think of it differently than a standard programming language
• A “standard programming language” such as C, C++, python, java, etc.:
  – Is a way to code an *algorithm* or *is a way to calculate a result*
  – Is written by a *software writer*
  – Often results in a more elegant solution when the programmer uses finer features of the language
• On the other hand, a hardware description language:
  – Is a way to describe *hardware*
  – Is written by a *hardware designer*
  – Results in a far better solution when the designer uses only the most basic features of the language
Verilog

• Why are hardware designs made with simpler language features better?
  – Synthesis tools will have fewer opportunities to interpret (destroy) the circuit you really want
  – Less-commonly-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

• In this class:
  – To engrain good practices, use only the simplest constructs shown in handouts, lectures, and labs to receive full credit
  – Corollary: avoid copying code off the web or other outside sources!
Verilog

• Design process
  – Think
  – Draw diagrams of hardware, figure out where logic and registers go, choose signal names carefully
  – Think

  – Then…
    • Write verilog
    • Test it
    • Optimize it

"A man who carries a cat by the tail learns something he can learn in no other way" -Samuel Clemens
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 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 – verilog simulator
  – Simvision – waveform viewer
  – Verilog XL – slower run time, but faster start up time as it spends less time compiling 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

• Modelsim

• 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*.
  - Later versions include *Verilog-2001* and *Verilog-2005*.
  - 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?!
• Modules are basic building blocks. These are two example module definitions which you should use:

```verilog
module abc (in1, in2, out);
    input in1;
    input in2;
    output out;
<body of module>
endmodule
```
or

```verilog
module abc (in1, in2, out);
    input in1,
    input in2,
    output out // notice no comma on last one;
<body of module>
endmodule
```

Others exist, but use only one of these two forms in this class

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

• Comments
  – Single line
    assign b = c;     // A comment
  – Multiple lines
    /* This module filters a series of images at a
       rate of “f” frames per second      */

• Values of a single wire or register (not buses)
  – 0 and 1       // binary zero and one
  – x or X        // unknown value due to things such as uninitialized or
                  // two drivers driving the same net
  – z or Z        // high impedance, e.g., a node not driven by any circuit
  – others        // don’t worry about others
Verilog

• Constants
  – Can be specified in a number of formats; use just these four in this class:
    • binary
    • hexadecimal
    • octal
    • decimal
  – Syntax: `[size.in.bits][first.letter.of.base.of.representation][value]`
  – Underscore characters (“_”) are ignored and can greatly help readability
  – Make sure to specify enough digits to cover the full range of the constant. Although Quartus will probably not complain, other CAD tools may do something you are not expecting especially with more complex number formats.
  – Examples:

  | Constant value in binary | 1'b0 | 0 |
  | 1'b1 | 1 |
  | 4'b0101 | 0101 |
  | 5'h0B | 01011 // specify two hex digits for 5 bits |
  | 16'h3F09 | 0011111100001001 // four hex digits for 16 bits |
  | 12'b0000_1010_0101 | 000010100101 // underscores are ignored |
  | 8'd003 | 00000011 // specify three base 10 digits for 8 bits |
Constants With \texttt{parameter} and `\texttt{define}`

- There are two main methods to simplify constants by using readable text to represent a number
  - \texttt{parameter}
    - Local to a module
    - Usage:
      
      \begin{verbatim}
      parameter HALT = 4'b0101;
      ...
      if (inst == HALT) begin
      \end{verbatim}
    - Definitely use this for state names in state machines in this class
  - `\texttt{define}` macro
    - Global text macro substitution using a compiler directive
    - Usage:
      
      \begin{verbatim}
      `define HALT 4'b0101
      ...
      if (inst == `HALT) begin \hspace{1em} // requires “back tick” “grave accent”
      \end{verbatim}
    - Best when helpful to put all definitions in a global file; probably do not use in this class
Verilog

- Operators: bit-wise
  - negation \( \sim \)
  - AND \( \& \)
  - OR \( | \)
  - XOR ^
  - Shift left by \( b \) bits \( a << b \)
  - Shift right by \( b \) bits \( a >> b \)

- Operators: logical (e.g., test for if-then-else)
  - negation \( ! \)
  - AND \( && \)
  - OR \( || \)

- Basic arithmetic
  - addition \( + \)
  - subtraction \( - \)
  - multiplication \( * \)
  - division \( / \)
  - modulus \( \% \)
• Equality, inequalities, and relational operators—all return a 1-bit true or false result
  – equal
  – not equal
  – less than
  – greater than
  – less than or equal
  – greater than or equal

• Miscellaneous
  – concatenation
    
    For example, to replicate the sign bit of a 4-bit value $a$ two times and assign it to $b$:

    ```verilog
    reg [5:0] b;  // b is a 6-bit reg
    b = {a[3], a[3], a};
    
    If $a$ were 1010, then $b$ would be 111010
    ```

  – delay of $n$ time units
    
    For this class, delays should appear in only two locations
    1) In testbench code to time test signals
    2) As a clock-to-Q delay in flip-flop declarations
    3) (They are also used to add crude delay approximations, but not in this class)
There are three main ways to specify hardware circuits:

1) wires, assign statements
2) registers, always blocks
3) Instantiate another module

Modules are the basic building blocks of HW

```verilog
module abc (in1, in2, out);
    input in1;
    input in2;
    output out;
    assign...
    always...
    always...
    square_root sqr1 (clk, reset, in1, out1);
    square_root sqr2 (clk, reset, in2, out2);
endmodule
```
Verilog

- Ports of an instantiated module can be connected to signals referenced in the declaration assuming they are in the same order but this is dangerous so don’t do it. Instead write out both the port name and connected wire as shown below.

```verilog
// example module “abc” to be instantiated below
module abc (in1, in2, out);
    input in1;
    input in2;
    output out;
    ...
endmodule

// Don’t use this method! It works but typos can be difficult to catch
abc instance1A (phase3, angle, magnitude3);

// This is ok, ports are in order
abc instance2A (
    .in1 (phase1),
    .in2 (angle),
    .out (magnitude1) // notice there is no comma on last port
);

This is ok, ports are not in the same order as in the module declaration
abc instance3A (
    .in2 (angle), // in2 comes before in1 here but everything
    .in1 (phase2), // still works ok
    .out (magnitude2)
);
```
1) **wire, assign**

- Picture “always active” hardwired logic
- For now, declare all wires

```vbnet
wire a, b;
wire out;
```

![Diagram of circuit with inputs a and b, and output out]
1) wire, assign

- Example:

```vhdl
wire out;
assign out = a & b;
```
1) *wire, assign*

- **Example: multibit operands**

```verilog
wire [3:0] c, d;  // c and d are both 4 bits
wire [4:0] sum;   // sum is 5 bits so no overflow
assign sum = {c[3],c} + {d[3],d};
```

![Diagram of an adder circuit](image)
2) reg, always

- Picture a much more general way of assigning “wires” or “signals” or “buses”
- “if/then/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 to specify a circuit
- Syntax:
  
  ```
  always @(sensitivity list) begin
    statements
  end
  ```

- statements are executed when any signal in sensitivity list changes
2) reg, always

- Example: there is no difference whatsoever in this AND gate from the AND gate built using assign

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

![Diagram of an AND gate]

\[ a \quad \& \quad b \quad \Rightarrow \quad \text{out} \]
Delays

- Delays may be inserted into always blocks to cause the simulator to pause

```verilog
always @( ...) begin
    a = b;
    #5;       // 5-unit delay
    a = ~c;
    #5;
    a = (c|d)^(e|f);
end
```
Delays

- As previously mentioned, this is normally done in only testbench Verilog where it is essential either in 1) the testbench itself, or 2) the clock generator (see Verilog Testing notes)

```verilog
always @( ...) begin
  reset = 1'b1;
  in    = 16'h0000;
  #10;   // 10-unit delay
  reset = 1'b0;
  in    = 16'h0001;
  #10;
  in    = 16'h0002;
  #10;
  in    = 16'h0003;
  #10;
  ...
end
```
2) *reg, always*

- Example #1: 2:1 multiplexer

```verilog
reg out;
always @(a or b or s) begin
  if (s == 1'b0) begin
    out = a;
  end
  else begin
    out = b;
  end
end
```
2) reg, always

• Example #2: 2:1 multiplexer.
• Normally always include begin and end statements even though they are not needed when there is only one statement in the particular block. Text struck out below could be taken out but always add it anyway.

```verilog
reg out;
always @(a or b or s) begin
  if (s == 1'b0) begin
    out = a;
  end
  else begin
    out = b;
  end
end
```

2) *reg, always*

- Example #3: 2:1 multiplexer
- Unclear and highly discouraged, but it works

```verilog
reg out;
always @(a or b or s) begin
    out = b;
    if (s == 1'b0) begin
        out = a;
    end
end
```
2) \textit{reg, always}

- Example #4: 2:1 multiplexer
- Unclear and highly discouraged, but it works

```verilog
reg out;
always @(a or b or s) begin
  out = a;
  if (s == 1'b1) begin
    out = b;
  end
end
```
2) reg, always

- Example #5: 2:1 multiplexer
- Simpler but less clear way of writing if/then/else called "inline if" or "conditional operator" which is also found in some computer languages

```verilog
reg out;
always @(a or b or s) begin
  out = s ? b : a;
end
```

![Multiplexer Diagram]
Example #1: 4:1 multiplexer with zero on two inputs

```vhdl
reg out; // must be a reg to be set in an always block!
wire s1, s0; // could be wire or reg

always @(c or d or s1 or s0) begin
  case ({s1, s0})
    2'b00: begin
      out = c;
    end
    2'b01: begin
      out = 1'b0;
    end
    2'b10: begin
      out = 1'b0;
    end
    2'b11: begin
      out = d;
    end
    default: begin
      out = 1'b0; // zero
    end
  endcase
end // end of always block
```

```
c 00
0 01
0 10
d 11
s1, s0
```
2) *reg, always*

- Example #2: 4:1 multiplexer with zero on two inputs
- Here the case’s default section is used

```vhdl
reg out;       // must be a reg to be set in an always block!
wire s1, s0;  // could be wire or reg

always @(c or d or s1 or s0) begin
  case ({s1, s0})
    2'b00: begin
      out = c;
    end
    2'b11: begin
      out = d;
    end
    default: begin
      out = 1'b0;
    end
  endcase
end   // end of always block
```
2) \textit{reg, always}

- Example \#3: 4:1 multiplexer with zero on two inputs
- Here \textit{out} is set to a default value first

```vhdl
reg out;       // must be a reg to be set in an always block!
wire s1, s0;  // could be wire or reg

always @(c or d or s1 or s0) begin
  out = 1'b0;

  case ({s1, s0})
    2'b00: begin
      out = c;
    end
    2'b11: begin
      out = d;
    end
  endcase
end       // end of always block
```

![Diagram of 4:1 multiplexer with input combinations and output]

\[ c \rightarrow 00 \quad 0 \rightarrow 01 \quad d \rightarrow 11 \quad s1, s0 \quad out \],
Example #4: 4:1 multiplexer with zero on two inputs

Here *if* statements are used. Clearly there are many solutions.

```verilog
reg out;             // must be a reg to be set in an always block!
wire s1, s0;        // could be wire or reg

always @(c or d or s1 or s0) begin
  out = 1'b0;
  if ({s1,s0} == 2'b00) begin
    out = c;
  end
  if (s1==1'b1 && s0==1'b1) begin
    out = d;
  end
end  // end of always block
```

Avoid Inferring State For Combinational Circuits

1. Include \textbf{all input} variables in the sensitivity list!
2. Set the value of all regs in \textbf{all} paths through every \textbf{always} block
   
   - A nice solution is to set default values immediately after entering the \textbf{always} block
     
     - You will eliminate the chance of this bug if you do this
     - Setting output to \textbf{x} in the default of a case statement can help debugging, but may also cause warnings with some CAD tools
Mistake #1: Inferring State In Combinational Circuits By: An Incomplete Sensitivity List

- `always @(a) begin  // missing b
  out = a & b;
end`

- *out* updates when *a* changes as expected
- *out* does not update when *b* changes!
  - Put another way, *b* can change all it wants but *out* will not update—this requires a memory element to remember the last value of *out*

- 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
Mistake #2: Inferring State In Combinational Circuits By: Not Setting a *reg* In All Paths

- Example combinational circuit with output *freq_c*
  - If *xyz==4'b0010*, *freq_c* is a function of the inputs
  - If *freq==3'b000* or *3'b001*, *freq_c* is a function of the inputs
  - Otherwise, *freq_c* keeps its old value—which implies memory is needed! Which means our combinational circuit is broken.

```vhdl
// this "always" block does not instantiate combinational logic
always @(freq or xyz or abc) begin
  if (xyz==4'b0010) begin
    freq_c = abc;
  end

  case (freq) begin
    3'b000: freq_c = abc;
    3'b001: freq_c = abc + 3'b001;
  endcase
end
```

---

B. Baas
Mistake #2: Inferring State In Combinational Circuits By: Not Setting a *reg* In All Paths

- Example combinational circuit with output *c_freq*
  - Solution: *freq_c* is set regardless of the values of all inputs
  - One solution: always declare default values at beginning of *always* blocks
  - If helpful, declare default case in case statements

```vhdl
// this "always" block instantiates combinational logic
always @(freq or xyz or abc) begin
    // "default" section of always block
    freq_c = freq; // some default value

    // main logic block
    if (xyz==4'b0010) begin
        freq_c = abc;
    end
    else begin // perhaps not always applicable
        freq_c = ...;
    end

    case (freq) begin
        3'b000: freq_c = abc;
        3'b001: freq_c = abc+3'b001;
        default: freq_c = ...; // perhaps you know freq should never be anything other than 000 or 001, or perhaps you do want
    endcase
end
```

B. Baas
Concurrency

• All circuits operate independently and concurrently
  – Different from most programming paradigms

• This is natural if we remember “hardware verilog” describes real circuit hardware
You should think of Verilog modules as operating on independent circuits (remember *hardware* orientation).

```verilog
always begin  // this block executes repeatedly without pausing
  a = (b & c) | d;
  #5;        // 5-unit delay
  a = ~a;
  #5;
end

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

```
\begin{tikzpicture}
  \draw[->] (0,0) -- (7,0) node[right] {$t$};
  \draw (0,0) -- (0,0.5);
  \draw (5,0) -- (5,0.5);
  \draw (7,0) -- (7,0.5);
  \draw (0,0) -- (1,0.5);
  \draw (1,0.5) -- (4,0.5);
  \draw (4,0.5) -- (5,0.5);
  \draw (5,0.5) -- (7,0.5);
  \node at (1,0.5) {$a$};
  \node at (5,0.5) {$f$};
\end{tikzpicture}
```
Other Special Block Styles

- This block executes only once at the beginning of the simulation. It is the normal way to write testbench code.
  
  ```
  initial begin
  ... 
  end
  ```

- This block executes repeatedly; it begins another execution cycle as soon as it finishes. Therefore it must contain some delays. This is a good construct for a clock oscillator.
  
  ```
  always begin
  ... 
  end
  ```