VERILOG:
INTRODUCTION AND
COMBINATIONAL LOGIC
Verilog

• Verilog is a *Hardware* Description Language (HDL)

• 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
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*
- Done by a *software writer*
- 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*
- Done by a *hardware designer*
- 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
  ```
  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?!
Verilog

- **Modules are basic building blocks**

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

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

- **Comments**
  - Single line
    ```verilog
call  b = c;  // A comment
```
  - Multiple lines
    ```verilog
    /* 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
Constants

- Can be specified in a number of formats; use just these:
  - binary
  - hexadecimal
  - octal
  - decimal

- Syntax: `[size.in.bits]'[first.letter.of.base.of.representation][value]

- 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:

<table>
<thead>
<tr>
<th>Constant value in binary</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>1'b0</code></td>
</tr>
<tr>
<td><code>1'b1</code></td>
</tr>
<tr>
<td><code>4'b0101</code></td>
</tr>
<tr>
<td><code>5'h0B</code></td>
</tr>
<tr>
<td><code>16'h3F09</code></td>
</tr>
</tbody>
</table>
  | `8'd003`      | `00000011`    | // specify three base 10 digits for 8 bits
Verilog

• Operators: bit-wise
  – negation ~
  – 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 order but this is dangerous so don’t do it. Instead write out both the port name and connected value as shown below.

```verilog
module abc (in1, in2, out);
    input in1;
    input in2;
    output out;
    ...
endmodule

abc instance001A (  .in1    (phase1),
                    .in2    (angle),
                    .out    (magnitude1)   // notice there is no comma on last port );

abc instance002A (  .in2    (angle),   // in2 comes before in1 here but
                    .in1    (phase2),   // everything still works ok
                    .out    (magnitude2) );

// Don’t use this method! It works but typos can be difficult to catch
abc instance003A (phase3, angle, magnitude3);
1) *wire, assign*

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

- Example:

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

- Example: multibit operands

```vhdl
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]
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 in this AND gate from the AND gate built using $assign$

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

![Waveform diagram](image)
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) \textit{reg, always}

- Example #1: 2:1 multiplexer

\begin{verbatim}
reg out;
always @(a or b or s) begin
  if (s == 1'b0) begin
    out = a;
  end
  else begin
    out = b;
  end
end
\end{verbatim}
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) **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
```

![2:1 multiplexer diagram]
2) **reg, always**

- **Example #1: 4:1 multiplexer with zero on two inputs**

  ```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
    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
  ```
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
```

```plaintext
\[
\begin{array}{c|c}
  c & 00 \\
  0 & 01 \\
  0 & 10 \\
  d & 11 \\
\end{array}
\]
```

- `s1, s0`
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
```

```plaintext
\begin{array}{c|c|c|c}
\text{s1, s0} & \text{00} & \text{01} & \text{10} & \text{11} \\
\hline
\text{c} & 0 & 1 & 0 & 1 \\
\text{d} & 1 & 0 & 1 & 1 \\
\end{array}
```

B. Baas
2) *reg*, *always*

- **Example #4: 4:1 multiplexer with zero on two inputs**
- **Here *if* statements are used. Clearly there are many solutions.**

```verbatim
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
```

```
<table>
<thead>
<tr>
<th>c</th>
<th>00</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>01</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>d</td>
<td>11</td>
</tr>
</tbody>
</table>
```
Avoid Inferring State For Combinational Circuits

1. Include **all input** variables in the sensitivity list!
2. Set the value of all regs in **all** paths through every **always** block
   - A nice solution is to set default values immediately after entering the **always** block
     - You will eliminate the chance of this bug 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
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
```
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

```verilog
// 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
```

freq

abc

xyz
Concurrency

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

• This is natural if we remember “hardware verilog” describes real circuit hardware
Concurrent Operation

- 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
```
Other Special Block Syles

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

• This block executes repeatedly; it begins another execution cycle as soon as it finishes. Therefore it must contain some delays.
  
  ```plaintext
  always begin
  ... 
  end
  ```