Big picture: Two main Hardware Description Languages (HDL) out there

- VHDL
  - Designed by committee on request of the DoD
  - Based on Ada

- Verilog
  - Designed by a company for their own use
  - Based on C

- Both now have IEEE standards
- Both are in wide use
Overall Module Structure

module name (args...);
    begin
        parameter ...; // define parameters
        input ...;     // define input ports
        output ...;   // define output ports
        wire ...;     // internal wires
        reg ...;      // internal or output regs
        // the parts of the module body are
        // executed concurrently
        <module/primitive instantiations>
        <continuous assignments>
        <procedural blocks (always/initial)>
    endmodule

Overall Module Structure

module NAND2 (Y, A, B);
    begin
        parameter ...; // define parameters
        input A, B;   // define input ports
        output Y;    // define output ports
        wire ...;    // internal wires
        reg ...;     // internal or output regs
        // the parts of the module body are
        // executed concurrently
        assign Y = ~(A & B);
        <procedural blocks (always/initial)>
    endmodule
Overall Module Structure

module NAND2 (Y, A, B);
begin
    parameter ...; // define parameters
    input A, B;    // define input ports
    output Y;     // define output ports
    wire ...;     // internal wires
    reg ...;      // int. or output regs
    // the parts of the module body are
    // executed concurrently
    nand _i1 (Y, A, B);
endforeachmodule

endmodule
module NAND2 (Y, A, B);
begin
    parameter delay = 10; // define parameters
    input A, B;           // define input ports
    output Y;            // define output ports
    wire ...;            // internal wires
    reg ...;             // int. or output regs
    // the parts of the module body are
    // executed concurrently
    <module/primitive instantiations>
    <continuous assignments>
    always @(A or B)
        begin
            #delay Y = ~(A & B);
        end
endmodule
Assignments

- Continuous assignments to wire vars
  - assign variable = exp;
  - Always at the “top level” of the module
    - In the concurrent execution section
    - Results in combinational logic

Assignments

- Procedural assignment to reg vars
  - Always inside procedural blocks
    - Meaning “always” or “initial” blocks
  - blocking variable = exp;
  - non-blocking variable <= exp;
  - Can result in combinational or sequential logic
Block Structures

- Two types:
  - `always // repeats until simulation is done`
    ```
    begin
    ...
    end
    ```
  - `initial // executed once at beginning of simulation`
    ```
    begin
    ...
    end
    ```

Data Types

- `reg` and `wire` are the main variable types
- Possible values for `wire` and `reg` variables:
  - 0: logic 0, false
  - 1: logic 1, true
  - X: unknown logic value
  - Z: High impedance state
- `integer, time, and real` are used in behavioral modeling, and in simulation *(not synthesis)*
Registers

- Abstract model of a data storage element
- A `reg` holds its value from one assignment to the next
  - The value “sticks”
- Register type declarations
  - `reg a; // a scalar register`
  - `reg [3:0] b; // a 4-bit vector register`

Wires (nets)

- `wire` variables model physical connections
- They don’t hold their value
  - They must be driven by a “driver” (a gate output or a continuous assignment)
  - Their value is Z if not driven
- Wire declarations
  - `wire d; // a scalar wire`
  - `wire [3:0] e; // a 4-bit vector wire`
Memories

- Verilog models memory as an array of reg
- Each element in the memory is addressed by a single array index
- Memory declarations:
  - \( \text{a 256 word 8-bit memory (256 8-bit vectors)} \)
  \[ \text{reg [7:0] imem[0:255];} \]
  - \( \text{a 1k word memory with 32-bit words} \)
  \[ \text{reg [31:0] dmem[0:1023];} \]

Accessing Memories

\[ \text{reg [7:0] imem[0:255]; // 256 x 8 memory} \]
\[ \text{reg [7:0] foo; // 8-bit reg} \]
\[ \text{reg [2:0] bar, baz; // 3-bit regs} \]

\[ \text{foo = imem[15]; // get word 15 from imem} \]
\[ \text{bar = foo[6:4]; // extract 3 bits from foo} \]
\[ \text{baz = bar; // assign all 3 bits of baz} \]
Other types

- **Integers:**
  - `integer i, j; \ \ declare two scalar ints`
  - `integer k[0:7]; \ \ an array of 8 ints`
- **$time** - returns simulation time
  - Useful inside `$display` and `$monitor` commands…

Number Representations

- **Two forms are available:**
  - Simple decimal numbers: 45, 123, 49039…
  - `<size>`’<base>`<number>`
  - `base` is d, h, o, or b
  - `size` is number of bits (not digits!)
- `4'b1001` // a 4-bit binary number
- `8'h2f` // an 8-bit hex number
Relational Operators

  - The result is 0 if the relation is false, 1 if the relation is true, X if either of the operands has any X’s in the number
- A===B, A!==B
  - These require an exact match of numbers, X’s and Z’s included

Relational Operators

- !, &&, ||
  - Logical NOT, AND, OR of expressions
  - e.g. if (!((A == 5) && (B != 2)))
- ~, &, |, ^
  - bitwise NOT, AND, OR, and XOR
  - e.g. Y = ~(A & B);
- {a, b[3:0]} // example of concatenation
Testbench Template

- Testbench template generated by Cadence

Module “test”

Instance “top”

Verilog netlist module (DUT)

initial block with testbench code

DUT inputs

DUT outputs

DUT schematic (twoBitAdd)
Edit Test Fixture

Testbench Template

\`
\texttt{'timescale 1ns / 100ps}
\texttt{module test;}

\texttt{wire Cout;}
\texttt{reg Cin;}
\texttt{wire [1:0] Sum;}
\texttt{reg [1:0] A;}
\texttt{reg [1:0] B;}

\texttt{twoBitAdd top(Cout, Sum, A, B, Cin);}

\texttt{'include "testfixture.verilog"}
\texttt{endmodule}
\`
Edit Test Fixture

Module test:

```
module test;

    wire Cout;
    reg Cin;
    wire [1:0] Sum;
    reg [1:0] A;
    reg [1:0] B;

    twoBitAdd top(Cout, Sum, A, B, Cin);

    `ifdef verilog

    // Please enter any additional verilog stimulus in the
    // `/home//test fixture template` or `test fixture` file
    // include `/home//test fixture template` or `test fixture`
    // endif

    endmodule
```

DUT schematic

```
reg type for DUT inputs

twoBitAdd

wire type for DUT outputs
```
testfixture.verilog

- Again, template generated by Cadence

```verilog
// Verilog stimulus file.
// Please do not create a module in this file.

// Default verilog stimulus.
initial
begin
    A[1:0] = 2'b00;
    B[1:0] = 2'b00;
    Cin = 1'b0;
end
```

Edit Test Fixture
Testbench code

- All your test code will be inside an initial block!
  - Or, you can create new procedural blocks that will be executed concurrently
  - Remember the structure of the module
    - If you want new temp variables you need to define those outside the procedural blocks
  - DUT inputs and outputs have been defined in the template
    - DUT inputs are reg type
    - DUT outputs are wire type

```
initial
begin
  a[1:0] = 2'b00;
  b[1:0] = 2'b00;
  cin = 1'b0;
  $display("Starting...");
  #20
  $display("A = %b, B = %b, c = %b, Sum = %b, Cout = %b", a, b, cin, sum, cout);
  if (sum != 00) $display("ERROR: Sum should be 00, is %b", sum);
  if (cout != 0) $display("ERROR: cout should be 0, is %b", cout);
  a = 2'b01;
  $display("A = %b, B = %b, c = %b, Sum = %b, Cout = %b", a, b, cin, sum, cout);
  if (sum != 00) $display("ERROR: Sum should be 01, is %b", sum);
  if (cout != 0) $display("ERROR: cout should be 0, is %b", cout);
  b = 2'b01;
  $display("A = %b, B = %b, c = %b, Sum = %b, Cout = %b", a, b, cin, sum, cout);
  if (sum != 00) $display("ERROR: Sum should be 10, is %b", sum);
  if (cout != 0) $display("ERROR: cout should be 0, is %b", cout);
  $display("...Done");
$finish;
end
```
$\text{display, }$monitor

$\text{display}(\text{format-string, args});$
- like a printf
- $\text{fdisplay}$ goes to a file...
- $\text{fopen}$ and $\text{fclose}$ deal with files

$\text{monitor}(\text{format-string, args});$
- Wakes up and prints whenever args change
- Might want to include $\text{time}$ so you know when it happened...
- $\text{fmonitor}$ is also available...

---

Conditional, For

- If (\text{<expr>}) \text{<statement>} else \text{<statement>}
  - else is optional and binds with closest previous if that lacks an else
  - if (index > 0)
    - if (rega > regb)
      - result = rega;
    - else
      - result = regb;
- For is like C
  - for (initial; condition; step)
  - for (k=0; k<10; k=k+1)
    - \text{<statement>};

\text{No } k++\text{ syntax in Verilog...}
for

parameter MAX_STATES 32;
integer state[0:MAX_STATES-1];
integer i;
initial
begin
begin
for(i=0; i<32; i=i+2)
state[i] = 0;
for(i=1; i<32; i=i+2)
state[i] = 1;
end

while

- A while loop executes until its condition is false

count = 0;
while (count < 128)
begin
$display("count = %d", count);
count = count + 1;
end
repeat

* repeat for a fixed number of iterations

```plaintext
cycles = 128;
integer count;
initial
begin
  count = 0;
  repeat(cycles)
  begin
    $display("count = \%", count);
    count = count+1;
  end
end
```

Nifty Testbench

```plaintext
reg [1:0] aina[0:4]; // define memory arrays to hold input and result
reg [1:0] bina[0:4];
reg [2:0] results[0:4];
integer i;
initial begin
readmemb("ain.txt", aina);
readmemb("bin.txt", bina);
readmemb("results.txt", results);
a[1:0] = 2'b00; // initialize inputs
b[1:0] = 2'b00;
cin = 1'b0;
end
$display("Starting...");
#10 $display("A = %b, B = %b, Sum = %b, Cout = %b", a, b, cin, sum, cout);
for (i=0; i<4; i=i+1) // loop through all values in the memories
begin
  a = aina[i]; // set the inputs from the memory arrays
  b = bina[i];
  #10 $display("A = %b, B = %b, Sum = %b, Cout = %b", a, b, cin, sum, cout);
  if ({cout, sum} != results[i])
    $display("Error: Sum should be %b, is %b instead", results[i], sum); // check results
end
$display("...Done");
$finish;
end
```
Another Nifty Testbench

```verilog
integer i,j,k;
initial
begin
A[1:0] = 2'b00;
B[1:0] = 2'b00;
Cin = 1'b0;
$display("Starting simulation...");
for(i=0;i<3;i=i+1)
begin
  for(j=0;j<3;j=j+1)
  begin
    for(k=0;k<1;k=k+1)
    begin
      #20 $display("A =%b B=%b Cin=%b, Cout-Sum=%b", A, B, Cin, Cout, S);
if ({Cout,S} != A + B + Cin)
  $display("ERROR: CoutSum should equal %b, is %b,(A + B + Cin), {Cin,S});
  Cin=˜Cin; // invert Cin
end
B[1:0] = B[1:0] + 2'b01; // add 1 to the B input
A = A+1; // shorthand notation for adding
end
$display("Simulation finished...");
end
```

Another Example

```verilog
initial begin // executed only once
  a = 2'b01; // initialize a and b
  b = 2'b00;
end
always begin // execute repeatedly until simulation completes
  #50 a = ~a; // reg a inverts every 50 units
end
always begin // execute repeatedly
  #100 b = ~b // reg b inverts every 100 units
end
```

What’s wrong with this code?
Another Example

initial begin  // executed only once
  a = 2’b01;  // initialize a and b
  b = 2’b00;
  #200 $finish; // make sure the simulation finishes!
end

always begin  // execute repeatedly until simulation completes
  #50 a = ~a; // reg a inverts every 50 units
end

always begin  // execute repeatedly
  #100 b = ~b // reg b inverts every 100 units
end