

Hardware Description Languages

- Alternatives to specifying circuits:
 - Structural description (schematics)
 - connect primitive components
 - build complex systems using hierarchy
 - complex components comprising simpler components
 - abstraction/modularization
 - behavior determined by components and composition
 - Behavioral description (program)
 - describe circuit computation as algorithms/programs
 - programming language tailored to hardware
 - parallel processes for components
 - requires synthesis to produce a circuit
 - enabled by advances in logic/sequential synthesis (1980s)

Verilog - 1

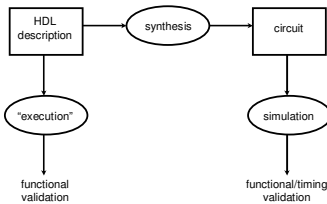
HDLs

- ISP (circa 1977) - research project at CMU
 - simulation, but no synthesis
- Abel (circa 1983) - developed by Data-I/O
 - targeted to programmable logic devices
 - not good for much more than state machines
- Verilog (circa 1985) - developed by Gateway (now part of Cadence)
 - similar to Pascal and C
 - fairly efficient and easy to write
 - IEEE standard
- VHDL (circa 1987) - DoD sponsored standard
 - similar to Ada (emphasis on re-use and maintainability)
 - simulation semantics visible
 - very general but verbose
 - IEEE standard

Verilog - 2

Simulation vs. Synthesis

- Early HDLs supported execution/simulation, not synthesis
 - C/algol, with parallel processes
- Synthesis constrains the language
 - current HDLs have a "synthesizable subset"



Verilog - 3

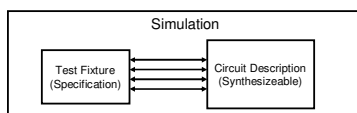
Verilog vs. VHDL

- The "standard" languages
- Very similar
 - Many tools provide front-ends to both
 - Verilog is "simpler"
 - less syntax, fewer constructs
 - VHDL supports large, complex systems
 - better support for modularization
 - more grungy details
 - "hello world" is much bigger in VHDL

Verilog - 4

Synthesis and Simulation

- Verilog/VHDL support both
- We will design using a synthesizable subset
 - e.g. no "initial" blocks
- Provide the circuit environment using Verilog simulation code
 - "Test fixture"
 - Arbitrary Verilog code
 - read files, print values, control simulation
 - Test fixture is the specification
 - used to decide if external circuit behavior is correct



Verilog - 5

Verilog

- Supports structural and behavioral descriptions
- Structural
 - explicit structure of the circuit
 - e.g., each logic gate instantiated and connected to others
 - we will use schematics to describe structure
 - much easier to understand
- Behavioral
 - program describes input/output behavior of circuit
 - many structural implementations could have same behavior
 - e.g., different implementation of one Boolean function
 - we will use Verilog for the "primitive" components
 - what is "primitive" is a matter of style

Verilog - 6

Verilog Introduction

- the **module** describes a component in the circuit
- Two ways to describe:
 - Structural Verilog
 - list of components and how they are connected
 - just like schematics, but using text
 - hard to write, hard to decode
 - useful if you don't have integrated design tools
 - we will ignore this, but you can look at netlists produced by Sue
 - Behavioral Verilog
 - describe *what* a component does, not *how* it does it
 - synthesized into a circuit that has this behavior
- Combination of both (example later)

Verilog - 7

Simple Behavioral Model

- Combinational logic
 - describe output as a function of inputs

```

module and_gate (out, in1, in2);
  input    in1, in2;
  output   out;

  assign out = in1 & in2;

endmodule
    
```

Verilog - 8

Verilog Module

- Corresponds to a circuit component
 - "parameter list" is the list of external connections, aka "ports"
 - ports are declared "input", "output" or "inout"
 - inout ports used on tri-state buses
 - port declarations declare the variables as wires

```

module name
├── ports
│   ├── input
│   ├── output
│   └── inout
└── inputs/outputs

module full_addr (A, B, Cin, S, Cout);
  input  A, B, Cin;
  output S, Cout;
  assign {Cout, S} = A + B + Cin;
endmodule
    
```

Verilog - 9

Verilog Data Types and Values

- Bits - value on a wire
 - 0, 1
 - X - don't care
 - Z - undriven, tri-state
- Vectors of bits
 - A[3:0] - vector of 4 bits: A[3], A[2], A[1], A[0]
 - Treated as an *unsigned integer* value
 - e.g. A < 0 ??
 - Concatenating bits/vectors into a vector
 - e.g. sign extend
 - B[7:0] = {A[3], A[3], A[3], A[3], A[3:0]};
 - B[7:0] = {4{A[3]}, A[3:0]};
 - Style: Use a[7:0] = b[7:0] + c;
 - Not: a = b + c; // need to look at declaration

Verilog - 10

Verilog Numbers

- 14 - ordinary decimal number
- 14 - 2's complement representation
- 12'b0000_0100_0110 - binary number with 12 bits (_ is ignored)
- 3'h046 - hexadecimal number with 12 bits
- Verilog values are *unsigned*
 - e.g. C[4:0] = A[3:0] + B[3:0];
 - if A = 0110 (6) and B = 1010(-6)
C = 10000 not 00000
i.e. B is zero-padded, not sign-extended

Verilog - 11

Verilog Operators

Verilog Operator	Name	Functional Group
{}	bit-select or part-select	
()	parenthesis	
!	logical negation	Logical
~	negation	Bit-wise
&	reduction AND	Reduction
	reduction OR	Reduction
~&	reduction NAND	Reduction
~	reduction NOR	Reduction
^	reduction XOR	Reduction
~^ or ^~	reduction XNOR	Reduction
+	unary (sign) plus	Arithmetic
-	unary (sign) minus	Arithmetic
{}	concatenation	Concatenation
{}	replication	Replication
*	multiply	Arithmetic
/	divide	Arithmetic
%	modulus	Arithmetic
+	binary plus	Arithmetic
-	binary minus	Arithmetic
<<	shift left	Shift
>>	shift right	Shift

>	greater than	Relational
>=	greater than or equal to	Relational
<	less than	Relational
<=	less than or equal to	Relational
==	logical equality	Equality
!=	logical inequality	Equality
===	case equality	Equality
!==	case inequality	Equality
&	bit-wise AND	Bit-wise
^	bit-wise XOR	Bit-wise
^~ or ^~	bit-wise XNOR	Bit-wise
	bit-wise OR	Bit-wise
&&	logical AND	Logical
	logical OR	Logical
?:	conditional	Conditional

Verilog - 12

Verilog Variables

- wire
 - variable used to connect components together
 - inputs and outputs are wires
 - outputs can be declared as regs
- reg
 - variable that saves a value as part of a behavioral description
 - usually corresponds to a wire in the circuit
 - is *NOT* a register in the circuit
- The rule:
 - Declare a variable as a reg if it is assigned (=) in an **always** block
 - assign doesn't count (confusing isn't it?)

Verilog - 13

Verilog Continuous Assignment

- Assignment is continuously evaluated
 - **assign** corresponds to a connection or a simple component with the described function
 - target is *not* a reg variable
 - use of Boolean operators (~ for bit-wise, ! for logical negation)
- ```

assign A = X | (Y & ~Z);
assign B[3:0] = 4'b01XX;
assign C[15:0] = 4'h00ff;
assign #3 {Cout, S[3:0]} = A[3:0] + B[3:0] + Cin;

```
- bits can take on four values (0, 1, X, Z)
  - variables can be n-bits wide (MSB:LSB)
  - use of arithmetic operator
  - multiple assignment (concatenation)
  - delay of performing computation, only used by simulator, not synthesis

Verilog - 14

## Comparator Example

```

module Compare1 (A, B, Equal, Alarger, Blarger);
 input A, B;
 output Equal, Alarger, Blarger;

 assign Equal = (A & B) | (~A & ~B);
 assign Alarger = (A & ~B);
 assign Blarger = (~A & B);
endmodule

```

Verilog - 15

## Comparator Example

```

// Make a 4-bit comparator from 4 1-bit comparators
module Compare4(A4, B4, Equal, Alarger, Blarger);
 input [3:0] A4, B4;
 output Equal, Alarger, Blarger;
 wire e0, e1, e2, e3, A10, A11, A12, A13, B10, B11, B12, B13;

 Compare1 cp0(A4[0], B4[0], e0, A10, B10);
 Compare1 cp1(A4[1], B4[1], e1, A11, B11);
 Compare1 cp2(A4[2], B4[2], e2, A12, B12);
 Compare1 cp3(A4[3], B4[3], e3, A13, B13);

 assign Equal = (e0 & e1 & e2 & e3);
 assign Alarger = (A13 | (A12 & e3) |
 (A11 & e3 & e2) |
 (A10 & e3 & e2 & e1));
 assign Blarger = (~Alarger & ~Equal);
endmodule

```

Verilog - 16

## Simple Behavioral Model - the always block

- always block
  - always waiting for a change to a trigger signal
  - then executes the body

```

module and_gate (out, in1, in2);
 input in1, in2;
 output out;
 reg out;

 always @(in1 or in2) begin
 out = in1 & in2;
 end
endmodule

```

Not a real register!!  
A Verilog register  
Needed because of  
assignment in  
always block

specifies when block is executed  
ie. triggered by which signals

Verilog - 17

## always Block

- A procedure that describes the function of a circuit
  - Can contain many statements including if, for, while, case
  - Statements in the always block are executed *sequentially*
    - (Continuous assignments are executed in parallel)
  - The entire block is executed at once
  - The *final* result describes the function of the circuit for current set of inputs
    - intermediate assignments don't matter, only the final result
- begin/end used to group statements
- This is *not* C programming!!!!!!
  - It is a *description* of a combinational function
  - Describes what the function computes, not how it computes it
    - Synthesis tools determine the how for you
      - for better or worse

Verilog - 18

## "Complete" Assignments

- If an always block executes, and a variable is *not* assigned
  - variable keeps its old value
  - *NOT* combinational logic  $\Rightarrow$  latch is inserted
  - This is *not* what you want
- Any variable assigned in an always block should be assigned for any execution of the block
- Defaults are usually a good idea

Verilog - 19

## Triggers

- Rule: always block input signals must be in trigger list
  - Emacs verilog mode can do this automatically for you
- Leaving out an input trigger usually results in a sequential circuit
- Example: The output of this "and" gate depends on the input history

```
module and_gate (out, in1, in2);
 input in1, in2;
 output out;
 reg out;

 always @(in1) begin
 out = in1 & in2;
 end
endmodule
```

Verilog - 20

## Verilog if

- Same as C if statement

```
// Simple 4-1 mux
module mux4 (sel, A, B, C, D, Y);
 input [1:0] sel; // 2-bit control signal
 input A, B, C, D;
 output Y;
 reg Y; // target of assignment

 always @(sel or A or B or C or D)
 if (sel == 2'b00) Y = A;
 else if (sel == 2'b01) Y = B;
 else if (sel == 2'b10) Y = C;
 else if (sel == 2'b11) Y = D;
endmodule
```

Verilog - 21

## Verilog if

- Another way

```
// Simple 4-1 mux
module mux4 (sel, A, B, C, D, Y);
 input [1:0] sel; // 2-bit control signal
 input A, B, C, D;
 output Y;
 reg Y; // target of assignment

 always @(sel or A or B or C or D)
 if (sel[0] == 0)
 if (sel[1] == 0) Y = A;
 else Y = B;
 else
 if (sel[1] == 0) Y = C;
 else Y = D;
endmodule
```

Verilog - 22

## Verilog case

- Sequential execution of cases
  - only first case that matches is executed (no break)
  - default case can be used

```
// Simple 4-1 mux
module mux4 (sel, A, B, C, D, Y);
 input [1:0] sel; // 2-bit control signal
 input A, B, C, D;
 output Y;
 reg Y; // target of assignment

 always @(sel or A or B or C or D)
 case (sel)
 2'b00: Y = A;
 2'b01: Y = B;
 2'b10: Y = C;
 2'b11: Y = D;
 endcase
endmodule
```

Verilog - 23

## Verilog case

- Without the default case, this would create a latch for Y
- Assigning X to a variable means synthesis is free to assign any value

```
// Simple binary encoder (input is 1-hot)
module encode (A, Y);
 input [7:0] A; // 8-bit input vector
 output [2:0] Y; // 3-bit encoded output
 reg [2:0] Y; // target of assignment

 always @(A)
 case (A)
 8'b00000001: Y = 0;
 8'b00000010: Y = 1;
 8'b00000100: Y = 2;
 8'b00001000: Y = 3;
 8'b00010000: Y = 4;
 8'b00100000: Y = 5;
 8'b01000000: Y = 6;
 8'b10000000: Y = 7;
 default: Y = 3'bx; // Don't care when input is not 1-hot
 endcase
endmodule
```

Verilog - 24

## Verilog case (cont)

- Cases are executed sequentially
  - The following implements a *priority* encoder

```
// Priority encoder
module encode (A, Y);
input [7:0] A; // 8-bit input vector
output [2:0] Y; // 3-bit encoded output
reg [2:0] Y; // target of assignment

always @(A)
case (1'b1)
A[0]: Y = 0;
A[1]: Y = 1;
A[2]: Y = 2;
A[3]: Y = 3;
A[4]: Y = 4;
A[5]: Y = 5;
A[6]: Y = 6;
A[7]: Y = 7;
default: Y = 3'bX; // Don't care when input is all 0's
endcase
endmodule
```

Verilog - 25

## Parallel Case

- A priority encoder is more expensive than a simple encoder
  - If we know the input is 1-hot, we can tell the synthesis tools
  - "parallel-case" pragma says the order of cases does not matter

```
// simple encoder
module encode (A, Y);
input [7:0] A; // 8-bit input vector
output [2:0] Y; // 3-bit encoded output
reg [2:0] Y; // target of assignment

always @(A)
case (1'b1) // synopsys parallel-case
A[0]: Y = 0;
A[1]: Y = 1;
A[2]: Y = 2;
A[3]: Y = 3;
A[4]: Y = 4;
A[5]: Y = 5;
A[6]: Y = 6;
A[7]: Y = 7;
default: Y = 3'bX; // Don't care when input is all 0's
endcase
endmodule
```

Verilog - 26

## Verilog casex

- Like case, but cases can include 'X'
  - X bits not used when evaluating the cases

Verilog - 27

## casex Example

```
// Priority encoder
module encode (A, valid, Y);
input [7:0] A; // 8-bit input vector
output [2:0] Y; // 3-bit encoded output
output valid; // Asserted when an input is not all 0's
reg [2:0] Y; // target of assignment
reg valid;

always @(A) begin
valid = 1;
casex (A)
8'bXXXXXXX1: Y = 0;
8'bXXXXXXX10: Y = 1;
8'bXXXXXXX100: Y = 2;
8'bXXXX1000: Y = 3;
8'bXXXX10000: Y = 4;
8'bXXX100000: Y = 5;
8'bX1000000: Y = 6;
8'b10000000: Y = 7;
default: begin
valid = 0;
Y = 3'bX; // Don't care when input is all 0's
end
endcase
end
endmodule
```

Verilog - 28

## Verilog for

- for is similar to C
- for statement is executed at compile time
  - result is all that matters, not how result is calculated

```
// simple encoder
module encode (A, Y);
input [7:0] A; // 8-bit input vector
output [2:0] Y; // 3-bit encoded output
reg [2:0] Y; // target of assignment

integer i; // Temporary variables for program only
reg [7:0] test;

always @(A) begin
test = 8b'00000001;
Y = 3'bX;
for (i = 0; i < 8; i = i + 1) begin
if (A == test) Y = i;
test = test << 1;
end
end
endmodule
```

Verilog - 29

## Another Behavioral Example

- Combinational block that computes Conway's Game of Life rule

```
module life (neighbors, self, out);
input self;
input [7:0] neighbors;
output out;
reg count;
integer i;

always @(neighbors or self) begin
count = 0;
for (i = 0; i < 8; i = i+1) count = count + neighbors[i];
out = 0;
out = out | (count == 3);
out = out | ((self == 1) & (count == 2));
end
endmodule
```

Integers are temporary compiler variables

always block is executed instantaneously, if there are no delays only the final result is used

Verilog - 30

## Verilog while/repeat/forever

- while (expression) statement
  - execute statement while expression is true
- repeat (expression) statement
  - execute statement a fixed number of times
- forever statement
  - execute statement forever