EECS150 - Digital Design

Lecture 5 - Verilog Introduction

Feb 1, 2011

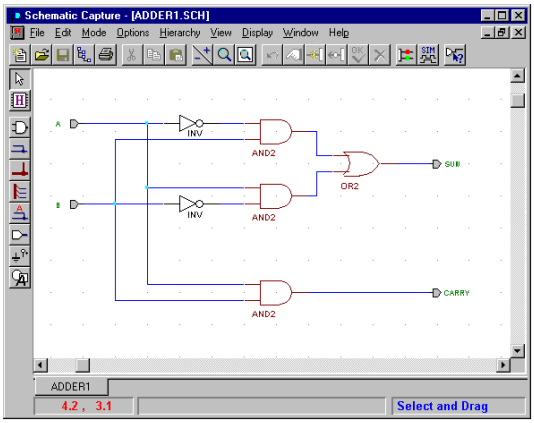
John Wawrzynek

Outline

- Background and History of Hardware Description
- Brief Introduction to Verilog Basics
- Lots of examples
  - structural, data-flow, behavioral
- Verilog in EECS150

Design Entry

- Schematic entry/editing used to be the standard method in industry and universities.
- Used in EECS150 until 2002
- ☺ Schematics are intuitive. They match our use of gate-level or block diagrams.
- ☺ Somewhat physical. They imply a physical implementation.
- ☹ Require a special tool (editor).
- ☹ Unless hierarchy is carefully designed, schematics can be confusing and difficult to follow on large designs.



- Hardware Description Languages (HDLs) are the new standard
  - except for PC board design, where schematics are still used.

Hardware Description Languages

- Basic Idea:
  - Language constructs describe circuits with two basic forms:
  - **Structural descriptions:** connections of components. Nearly one-to-one correspondence to with schematic diagram.
  - **Behavioral descriptions:** use high-level constructs (similar to conventional programming) to describe the circuit function.
- Originally invented for simulation.
  - Now "logic synthesis" tools exist to automatically convert from HDL source to circuits.
  - High-level constructs greatly improves designer productivity.
  - However, this may lead you to falsely believe that hardware design can be reduced to writing programs!*

"Structural" example:

```
Decoder(output x0,x1,x2,x3;
                    inputs a,b)
{
   wire abar, bbar;
   inv(bbar, b);
   inv(abar, a);
   and(x0, abar, bbar);
   and(x1, abar, b   );
   and(x2, a,    bbar);
   and(x3, a,    b   );
}
```

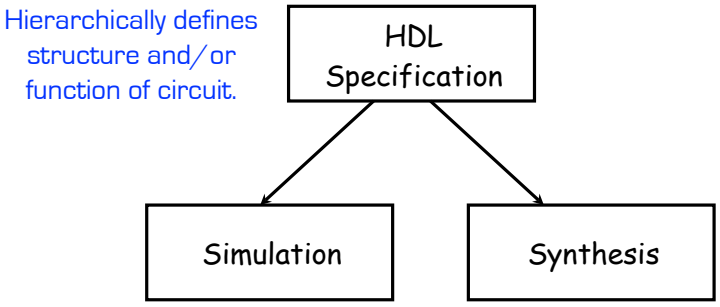
"Behavioral" example:

```
Decoder(output x0,x1,x2,x3;
                    inputs a,b)
{
   case [a b]
            00: [x0 x1 x2 x3] = 0x1;
            01: [x0 x1 x2 x3] = 0x2;
            10: [x0 x1 x2 x3] = 0x4;
            11: [x0 x1 x2 x3] = 0x8;
    endcase;
}
```

Warning: this is a fake HDL!

*Describing hardware with a language is similar, however, to writing a parallel program.

Sample Design Methodology



Note: This in not the entire story. Other tools are useful for analyzing HDL specifications. More on this later.

Verilog

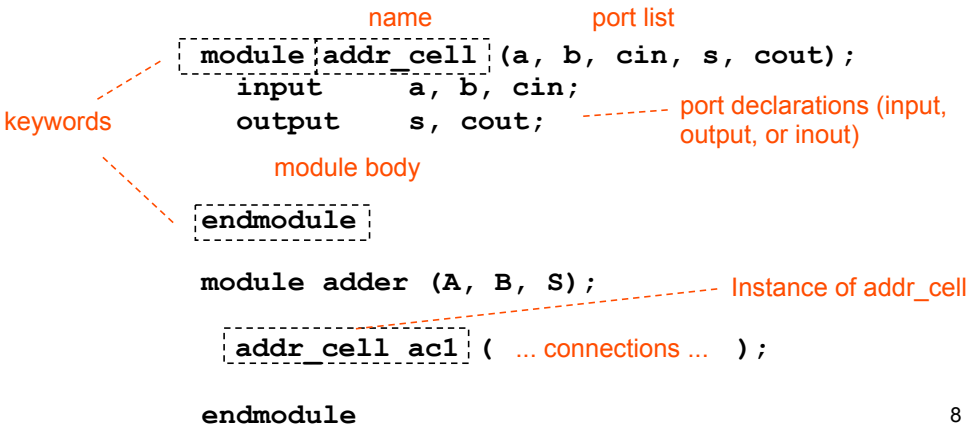
- A brief history:
  - Originated at Automated Integrated Design Systems (renamed Gateway) in 1985. Acquired by Cadence in 1989.
  - Invented as simulation language. Synthesis was an afterthought. Many of the basic techniques for synthesis were developed at Berkeley in the 80's and applied commercially in the 90's.
  - Around the same time as the origin of Verilog, the US Department of Defense developed VHDL (A double acronym! VSIC (Very High-Speed Integrated Circuit) HDL). Because it was in the public domain it began to grow in popularity.
  - Afraid of losing market share, Cadence opened Verilog to the public in 1990.
  - An IEEE working group was established in 1993, and ratified IEEE Standard 1394 (Verilog) in 1995. We use IEEE Std 1364-2001.
  - Verilog is the language of choice of Silicon Valley companies, initially because of high-quality tool support and its similarity to C-language syntax.
  - VHDL is still popular within the government, in Europe and Japan, and some Universities.
  - Most major CAD frameworks now support both.
  - Latest Verilog version is "system Verilog" .
  - Latest HDL: C++ based. OSCI (Open System C Initiative).

Verilog Introduction

- A **`module`** definition describes a component in a circuit
- Two ways to describe module contents:
  - Structural Verilog
    - List of sub-components and how they are connected
    - Just like schematics, but using text
    - tedious to write, hard to decode
    - You get precise control over circuit details
    - May be necessary to map to special resources of the FPGA
  - Behavioral Verilog
    - Describe what a component does, not how it does it
    - Synthesized into a circuit that has this behavior
    - Result is only as good as the tools
- Build up a hierarchy of modules. Top-level module is your entire design (or the environment to test your design).

Verilog Modules and Instantiation

- Modules define circuit components.
- Instantiation defines hierarchy of the design.



Note: A module is not a function in the C sense. There is no call and return mechanism. Think of it more like a hierarchical data structure.

Structural Model - XOR example

```
module xor_gate ( out, a, b );
  input     a, b;
  output    out;
  wire      aBar, bBar, t1, t2;

  not invA (aBar, a);
  not invB (bBar, b);
  and and1 (t1, a, bBar);
  and and2 (t2, b, aBar);
  or  or1 (out, t1, t2);

endmodule
```

module name

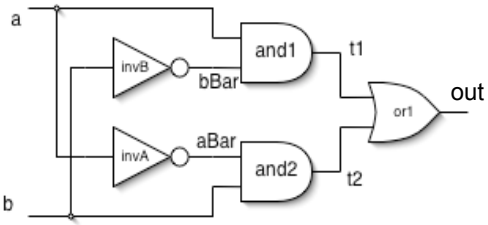
port list

port declarations

internal signal declarations

Built-in gates

instances

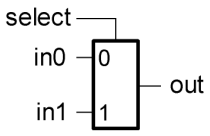


Instance name

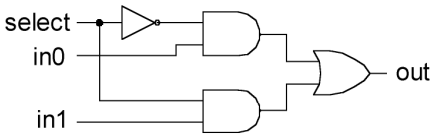
Interconnections (note output is first)

- Notes:
  - The instantiated gates are not "executed". They are active always.
  - xor gate already exists as a built-in (so really no need to define it).
  - Undeclared variables assumed to be wires. Don't let this happen to you!

Structural Example: 2-to1 mux



a) 2-input mux symbol



b) 2-input mux gate-level circuit diagram

```
/* 2-input multiplexor in gates */
module mux2 (in0, in1, select, out);
   input in0,in1,select;
   output out;
   wire s0,w0,w1;

   not (s0, select);
   and (w0, s0, in0),
       (w1, select, in1);
   or  (out, w0, w1);

endmodule // mux2
```

C++ style comments

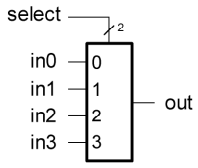
Built-ins don't need Instance names

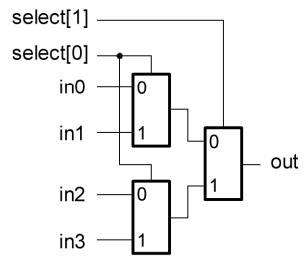
Multiple instances can share the same "master" name.

Built-ins gates can have > 2 inputs. Ex:

```
and (w0, a, b, c, d);
```

Instantiation, Signal Array, Named ports






```
/* 2-input multiplexor in gates */
module mux2 (in0, in1, select, out);
   input in0,in1,select;
   output out;
   wire s0,w0,w1;
   not (s0, select);
   and (w0, s0, in0),
       (w1, select, in1);
   or  (out, w0, w1);
endmodule // mux2
```

a) 4-input mux symbol b) 4-input mux implemented with 2-input muxes

```
module mux4 (in0, in1, in2, in3, select, out);
input in0,in1,in2,in3;
input [1:0] select;
output out;
wire w0,w1;
  mux2
    m0 (.select(select[0]), .in0(in0), .in1(in1), .out(w0)),
    m1 (.select(select[0]), .in0(in2), .in1(in3), .out(w1)),
    m3 (.select(select[1]), .in0(w0), .in1(w1), .out(out));
endmodule // mux4
```

Signal array. Declares select[1], select[0]

Named ports. Highly recommended.

Simple Behavioral Model

```
module foo (out, in1, in2);
  input         in1, in2;
  output        out;

    assign out = in1 & in2;

endmodule
```

"continuous assignment"

Connects out to be the "and" of in1 and in2.

Shorthand for explicit instantiation of "and" gate (in this case).

The assignment continuously happens, therefore any change on the rhs is reflected in out immediately (except for the small delay associated with the implementation of the &).

Not like an assignment in C that takes place when the program counter gets to that place in the program.

Continuous Assignment Examples

```
wire [3:0] X,Y,R;
wire [7:0] P;
wire r, a, cout, cin;
```

```
assign R = X | (Y & ~Z);
```
use of bit-wise Boolean operators

```
assign r = &X;
```
example reduction operator

```
assign R = (a == 1'b0) ? X : Y;
```
conditional operator

```
assign P = 8'hff;
```
example constants

```
assign P = X * Y;
```
arithmetic operators (use with care!)

```
assign P[7:0] = {4{X[3]}, X[3:0]};
```
(ex: sign-extension)

```
assign {cout, R} = X + Y + cin;
```
bit field concatenation

```
assign Y = A << 2;
```
bit shift operator

```
assign Y = {A[1], A[0], 1'b0, 1'b0};
```
equivalent bit shift

Verilog Operators

Verilog Operator	Name	Functional Group
( )	bit-select or part-select	
( )	parenthesis	
! ~ & | ~& ~| ^ ~^ or ^~	logical negation negation reduction AND reduction OR reduction NAND reduction NOR reduction XOR reduction XNOR	Logical Bit-wise Reduction Reduction Reduction Reduction Reduction Reduction
+ -	unary (sign) plus unary (sign) minus	Arithmetic Arithmetic
{ }	concatenation	Concatenation
{{ }}	replication	Replication
* / %	multiply divide modulus	Arithmetic Arithmetic Arithmetic
+ -	binary plus binary minus	Arithmetic Arithmetic
<< >>	shift left shift right	Shift Shift
>= < <=	greater than greater than or equal to less than less than or equal to	Relational Relational Relational Relational
== !=	logical equality logical inequality	Equality Equality
=== !==	case equality case inequality	Equality Equality
&	bit-wise AND	Bit-wise
^ ^~ or ~^	bit-wise XOR bit-wise XNOR	Bit-wise Bit-wise
|	bit-wise OR	Bit-wise
&&	logical AND	Logical
||	logical OR	Logical
?:	conditional	Conditional

Verilog Numbers

Constants:

14 ordinary decimal number

-14 2's complement representation

12'b0000_0100_0110 binary number ("_" is ignored)

12'h046 hexadecimal number with 12 bits

Signal Values:

By default, 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

wire signed [31:0] x;

Declares a signed (2's complement) signal array.

Non-continuous Assignments

A bit strange from a hardware specification point of view.

Shows off Verilog roots as a simulation language.

"always" block example:

```
module and_or_gate (out, in1, in2, in3);
  input     in1, in2, in3;
  output    out;
  reg       out;

  always @(in1 or in2 or in3) begin
     out = (in1 & in2) | in3;
  end

endmodule
```

"reg" type declaration. Not really a register in this case. Just a Verilog rule.

keyword

"sensitivity" list, triggers the action in the body.

brackets multiple statements (not necessary in this example.

Isn't this just: **assign out = (in1 & in2) | in3;**?

Why bother?

Always Blocks

Always blocks give us some constructs that are impossible or awkward in continuous assignments.

case statement example:

```
module mux4 (in0, in1, in2, in3, select, out);
   input in0,in1,in2,in3;
   input [1:0] select;
   output      out;
   reg         out;

  always @ (in0 in1 in2 in3 select)
      case (select)
            2'b00: out=in0;
            2'b01: out=in1;
            2'b10: out=in2;
            2'b11: out=in3;
      endcase
endmodule // mux4
```

keyword

The statement(s) corresponding to whichever constant matches "select" get applied.

Couldn't we just do this with nested "if"s?

Well yes and no!

Always Blocks

Nested if-else example:

```
module mux4 (in0, in1, in2, in3, select, out);
   input in0,in1,in2,in3;
   input [1:0] select;
   output      out;
   reg         out;

  always @ (in0 in1 in2 in3 select)
      if (select == 2'b00) out=in0;
      else if (select == 2'b01) out=in1;
           else if (select == 2'b10) out=in2;
                else out=in3;
endmodule // mux4
```

Nested if structure leads to "priority logic" structure, with different delays for different inputs (in3 to out delay > than in0 to out delay). Case version treats all inputs the same.

State Elements

Always blocks are the only way to specify the "behavior" of state elements. Synthesis tools will turn state element behaviors into state element instances.

D-flip-flop with synchronous set and reset example:

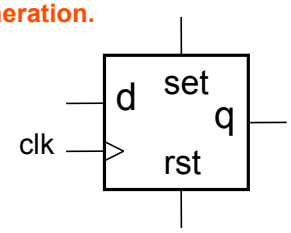
```
module dff(q, d, clk, set, rst);
  input d, clk, set, rst;
  output q;
  reg q;

  always @(posedge clk)
    if (rst)
      q <= 1'b0;
    else if (set)
      q <= 1'b1;
    else
      q <= d;
endmodule
```

keyword

"always @ (posedge clk)" is key to flip-flop generation.

This gives priority to reset over set and set over d.

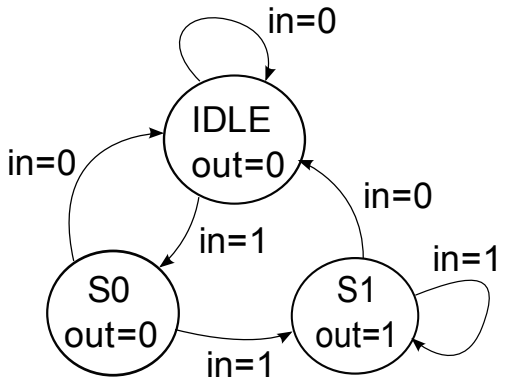


On FPGAs, maps to native flip-flop.

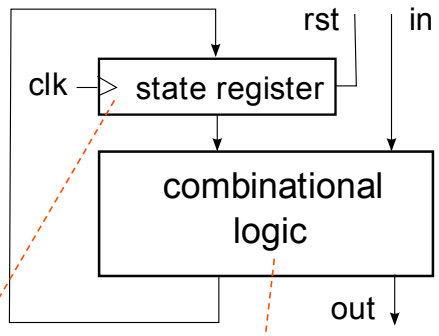
How would you add an CE (clock enable) input?

Finite State Machines

State Transition Diagram



Implementation Circuit Diagram



Holds a symbol to keep track of which bubble the FSM is in.

CL functions to determine output value and next state based on input and current state.

out = f(in, current state)

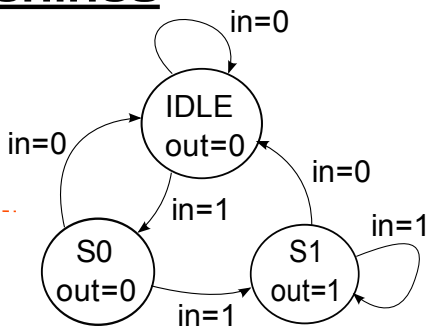
next state = f(in, current state)

Finite State Machines

```
module FSM1(clk, rst, in, out);
input clk, rst;
input in;
output out;

// Defined state encoding:
parameter IDLE = 2'b00;
parameter S0 = 2'b01;
parameter S1 = 2'b10;
reg out;
reg [1:0] state, next_state;

// always block for state register
always @(posedge clk)
      if (rst) state <= IDLE;
      else state <= next_state;
```



Must use reset to force to initial state.

reset not always shown in STD

Constants local to this module.

out not a register, but assigned in always block

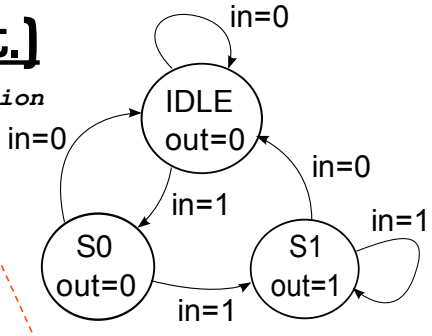
Combinational logic signals for transition.

THE register to hold the "state" of the FSM.

A separate always block will be used for combination logic part of FSM. Next state and output generation. (Always blocks in a design work in parallel.)

FSMs (cont.)

```
// always block for combinational logic portion
always @(state or in)
case (state)
// For each state def output and next
  IDLE    : begin
              out = 1'b0;
              if (in == 1'b1) next_state = S0;
              else next_state = IDLE;
            end
  S0      : begin
              out = 1'b0;
              if (in == 1'b1) next_state = S1;
              else next_state = S0;
            end
  S1      : begin
              out = 1'b1;
              if (in == 1'b1) next_state = S1;
              else next_state = IDLE;
  default: begin
              next_state = IDLE;
              data_out = 1'b0;
            end
endcase
endmodule
```



Each state becomes a case clause.

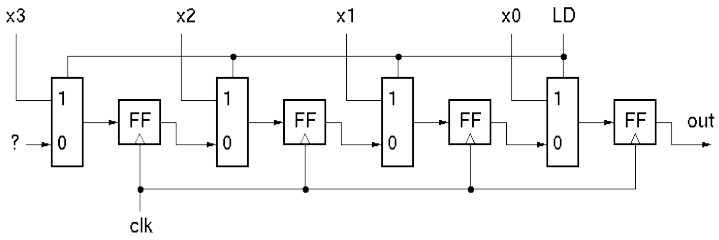
For each state define:

Output value(s)

State transition

Use "default" to cover unassigned state. Usually unconditionally transition to reset state.

Bonus Example




```
//Parallel to Serial converter
module ParToSer(LD, X, out, CLK);
  input [3:0] X;
  input LD, clk;
  output out;
  reg out;
  reg [3:0] Q;
  assign out = Q[0];
  always @ (posedge clk)
       if (LD) Q<=X;
       else Q <= Q>>1;
endmodule // mux2
```

"always @ (posedge CLK)" forces Q register to be rewritten every cycle.

">>" operator does right shift (shifts in a zero on the left).

Shifts can be done with concatenation. Continuous assign example:

```
wire [3:0] A, B;
assign B = {1'b0, A[3:1]}
```


Verilog in EECS150

- We will primarily use **behavioral modeling** along with **instantiation** to 1) build hierarchy and, 2) map to FPGA resources not supported by synthesis.
- Favor continuous assign and avoid always blocks unless:
  - no other alternative: ex: state elements, case
  - helps readability and clarity of code: ex: large nested if else
- Use named ports.
- Verilog is a big language. This is only an introduction.
  - Our text book is a good source. Read and use chapter 4.
  - Be careful of what you read on the web. Many bad examples out there.
  - We will be introducing more useful constructs throughout the semester. Stay tuned!

Final thoughts on Verilog Examples

Verilog looks like C, but it describes hardware

Multiple physical elements with parallel activities and temporal relationships.

A large part of digital design is knowing how to write Verilog that gets you the desired circuit. First understand the circuit you want then figure out how to code it in Verilog. If you do one of these activities without the other, you will struggle. These two activities will merge at some point for you.

Be suspicious of the synthesis tools! Check the output of the tools to make sure you get what you want.