EEC 485 Computer Organization

Introduction to Verilog

Reading: Appendix B.4, 5.8
Note: "For More Practice" problems are on the CDROM

Some slides come from Prof. John Nestor, Lafayette College, PA

Quick Tutorial (alu.v)

```
module alu (ALUctl, A, B, ALUOut, Zero);    // file "alu.v"

input [3:0] ALUctl;
input [31:0] A,B;
output [31:0] ALUOut;
output Zero;

reg [31:0] ALUOut;
assign Zero = (ALUOut==0); //Zero is true if ALUOut is 0

always @(ALUctl, A, B) begin //reevaluate if these change
    case (ALUctl)
            0: ALUOut <= A & B;
            1: ALUOut <= A | B;
            2: ALUOut <= A + B;
            6: ALUOut <= A - B;
            7: ALUOut <= A < B ? 1 : 0;
            12: ALUOut <= ~(A | B); // result is nor
            default: ALUOut <= 0;
    endcase
end

endmodule
```

Quick Tutorial (sim_alu.v)

```
module sim_alu;                                    // file "sim_alu.v"

reg [3:0] ALUctl;
reg [31:0] A,B;
wire [31:0] ALUOut;
wire Zero;

alu ALU1 (ALUctl, A, B, ALUOut, Zero);

initial begin
    ALUctl = 4'd2;
    A = 32'd20;
    B = 32'd12;
    #1;
    $display("ALUOut = ", ALUOut);
    $display("Zero = ", Zero);
    $finish;
end

endmodule
```


Outline - Introduction to Verilog

- What is HDL-Based Design?
- A First Example
- Operators in Verilog
- Data Types in Verilog
- Structure of a Verilog Program
  - Module and Port Declarations
  - Modeling with Hierarchy
  - Modeling with Continuous Assignments
  - Modeling with `always` blocks
- Project #2

What is HDL-Based Design?

- Model hardware for
  - Simulation - predict how hardware will behave
  - Synthesis - generate optimized hardware
- Provide a concise text description of circuits
- Support design of very large systems

What is HDL-Based Design?

- Verilog can specify both behavioral and structural definition of a digital system
  - Behavioral specification describes how a digital system functionally operates (but cannot be used for synthesis) (see Fig. 5.8.1)
  - Structural specification describes the detailed organization of a digital system usually using a hierarchical description (see Figs. 5.8.2 & 5.8.3)

A First Example

- **Full Adder :**


```
module fulladder(a, b, cin, sum, cout);
  input a, b, cin;
  output sum, cout;

  assign sum = a ^ b ^ cin;
  assign cout = a & b | a & cin | b & cin;
endmodule
```


Comments about the First Example

- **Verilog describes a circuit as a set of modules**
- **Each module has input and output ports**
  - **Single bit**
  - **Multiple bit - array syntax**
- **Each port can take on a digital value (0, 1, X, Z)**

Operators in Verilog

- **Supports most of C operators**
  - **Bitwise operators**
  - **Bitwise reduction operators (&, |, ^)**
  - **Conditional operators**
  - **Others**

Operators in Verilog - Bitwise Operators

- **Basic bitwise operators: identical to C/C++/Java**

```
module inv(a, y);
  input  [3:0] a;
  output [3:0] y;

  assign y = ~a;
endmodule
```

4-bit Ports

Unary Operator: NOT

Operators in Verilog - Reduction Operators

- Apply a single logic function to multiple-bit inputs

```
module and8(a, y);
  input  [7:0] a;
  output       y;

  assign y = &a;
endmodule
```

Reduction Operator: AND
equivalent to:
a[7] & a[6] & a[5] & a[4] & a[3] & a[2] & a[2] & a[2] & a[0]

Operators in Verilog - Conditional Operators

- Like C/C++/Java Conditional Operator

```
module mux2(d0, d1, s, y);
  input  [3:0] d0, d1;
  input        s;
  output [3:0] y;

  assign y = s ? d1 : d0;
  // output d1 when s=1, else d0
 endmodule
```

} Comment

Yes, this is a MUX implementation!

Operators in Verilog - More Operators

- **Equivalent to C/C++/Java Operators**
  - **Arithmetic:** `+` `-` `*` `/` `&`
  - **Comparison:** `==` `!=` `<` `<=` `>` `>=`
  - **Shifting:** `<<` `>>`
- **Example:**

```
module adder(a, b, y);
  input  [31:0]      a, b;
  output [31:0]      y;

  assign y = a + b;
endmodule
```

- **Warning: small expressions can make big hardware if complex operators are used!**

Operators in Verilog - Concatenation

- **{ } is the concatenation operator**

```
module adder(a, b, y, cout);
  input  [31:0]      a, b;
  output [31:0]      y;
   output            cout;

  assign {cout,y} = a + b;
endmodule
```

Concatenation (33 bits)

Operators in Verilog - Replication

- `{ n {pattern} }` replicates a pattern *n* times

```
module signextend(a, y);
  input  [15:0]     a;
  output [31:0]     y;

  assign y = {16{a[15]}, a[15:0]};
endmodule
```

Copies sign bit 16 times

Lower 16 Bits

Data Types in Verilog

- **Internal signals can be declared using wire and reg**
  - **"wire" specifies a combination signal, e.g. wire [31:0] X**
  - **"reg" holds a value, which can vary with time , e.g. reg [31:0] Y**
- **Possible values for a register or a wire are 0, 1, x, z**
- **Constant values can be specified as follows**
  - **4'b0100 : 4-bit data with binary value of 0100**
  - **-8'h4 : 8-bit data with hexa value of -4**

Data Types – Example

```
module fulladder(a, b, cin, s, cout);
  input        a, b, cin;
  output s, cout;

  wire         prop;

  assign prop = a ^ b;
  assign s = prop ^ cin;
  assign cout = (a & b) | (cin & (a | b));
endmodule
```

Important point: these statements "execute" in parallel

Structure of a Verilog Program

- Consists of a set of modules, each of which specifies its input and output ports (incoming and outgoing signals)
- The body of a module consists of
  - "initial" constructs (initialize "reg" variable)
  - Instances of other modules (hierarchy)
  - Continuous assignments using "assign"
  - "always" construct

Structure - Modeling with Hierarchy

- Create instances of submodules
- Example: Create a 4-input Mux using `mux2` module
- Original `mux2` module:

```
module mux2(d0, d1, s, y);
  input  [3:0] d0, d1;
  input        s;
  output [3:0] y;
  assign y = s ? d1 : d0;
 endmodule
```


Structure - Modeling with Hierarchy

- Example: Create a 4-input Mux using `mux2` module (continued)

```
module mux4(d0, d1, d2, d3, s, y);
  input  [3:0] d0, d1, d2, d3;
  input  [1:0] s;
  output [3:0] y;

  wire   [3:0] low, high;

  mux2 lowmux(d0, d1, s[0], low);
  mux2 highmux(d2, d3, s[0], high);
  mux2 finalmux(low, high, s[1], y);
endmodule
```

Instance Names Connections

Structure - Larger Hierarchy Example

- **Use full adder to create an n-bit adder**

```
module add8(a, b, sum, cout);
  input [7:0] a, b;
  output [7:0] sum;
  output cout;

  wire [7:0] c;  // used for carry connections

  assign c[0]=0;
  fulladder f0(a[0], b[0], c[0], sum[0], c[1]);
  fulladder f1(a[1], b[1], c[1], sum[1], c[2]);
  fulladder f2(a[2], b[2], c[2], sum[2], c[3]);
  fulladder f3(a[3], b[3], c[3], sum[3], c[4]);
  fulladder f4(a[4], b[4], c[4], sum[4], c[5]);
  fulladder f5(a[5], b[5], c[5], sum[5], c[6]);
  fulladder f6(a[6], b[6], c[6], sum[6], c[7]);
  fulladder f7(a[7], b[7], c[7], sum[7], cout);
endmodule
```

New internal variable as in general C programming. Here, they are used to make Connections among fulladder's.

Structure of a Verilog Program

- **The body of a module consists of**
  - **"initial" constructs, which initializes reg variables**
  - **Instances of other modules, which are used to implement the module being defined (hierarchy)**
  - **<u>Continuous assignments using "assign", which define only combinational logic</u>**
  - **<u>"always" construct, which can define either sequential or combination logic</u>**

**Generally, sequential.
But it can be implemented
by combination circuit
if no registers on the RHS.**

But why? – modular programming

Structure – “assign” vs “always”

- “assign”
  - Output is continuously assigned the value, and a change in the input values is reflected immediately in the output value
- Combination logic modeling with “assign”

```
module fulladder(a, b, cin, sum, cout);
  input a, b, cin;
  output sum, cout;

  assign sum = a ^ b ^ cin;
  assign cout = a & b | a & cin | b & cin;
endmodule
```


Structure – “assign” vs “always”

- “always”: Procedural modeling
  - Allowed to include control constructs such as if-then-else, case, for, and repeat statements
  - Reevaluated if any of the listed signals changes values
- Basic syntax:

```
always @(sensitivity-list)
     statement
```

Signal list - change activates block

Procedural statement (=, `if`/`else`, etc.)

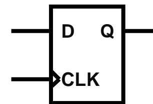
or

```
always @(sensitivity-list)
begin
     statement-sequence
end
```

Compound Statement - sequence of procedural statements

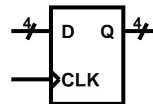
“always” – Examples (Flip-Flop, Register)

```
module flipflop_1(d, clk, q);
  input     d;
  input     clk;
  output    q;
  reg       q;
  always @(posedge clk)
      q <= d;
endmodule
```



for positive edge-trigger

```
module register_4(clk, d, q);
  input            clk;
  input       [3:0] d;
  output      [3:0] q;
  reg         [3:0] q;
  always @(posedge clk)
       q <= d;
endmodule
```



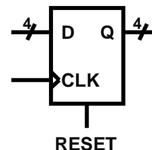
“always” – Example (Register with Reset)

- Synchronous - resets on clock edge if reset=1

```
module register_4(clk, reset, d, q);
  input         clk;
  input         reset;
  input   [3:0] d;
  output  [3:0] q;

  reg     [3:0] q;

  always @(posedge clk)
   if (reset) q <= 4'b0;
   else q <= d;
endmodule
```



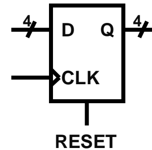
“always” – Example (Register with Reset)

- Asynchronous - resets immediately if reset=1

```
module register_4(clk, reset, d, q);
  input        clk;
  input        reset;
  input  [3:0] d;
  output [3:0] q;

  reg    [3:0] q;

  always @(posedge clk or posedge reset)
   if (reset) q <= 4'b0;
   else q <= d;
endmodule
```



“always”

- reg variables may only be assigned inside an “always” block
  - using <u>procedural assignment</u> statement
  - either blocking (=) or non-blocking (<=)

“wire” specifies a combination signal and “reg” holds a value as a register. Therefore, we need to initialize "reg" variable but not necessarily the case with "wire" variable. Also, "reg" variables may only be assigned inside an “always” block because it is important for “reg" variables to know when they can get updated.

All control signals such as ALUSrc may be declared as "wire" variable while all internal registers such as MDR and IR (in multicycle implementation) should be defined as "reg" variables.

"always" – Example (Shift Register)

```
module shiftreg(clk, sin, q);
  input         clk;
  input               sin;
  output [3:0] q;

  reg           [3:0] q;

  always @(posedge clk)
   begin
     q[3] <= q[2];
     q[2] <= q[1];
     q[1] <= q[0];
     q[0] <= sin;
   end
endmodule
```

Non-blocking assignments (immediate update)

Review : Verilog Module Declaration

- Describes the external interface of a single module
  - Name
  - Ports - inputs and outputs
- General Syntax:

```
module modulename ( port1, port2, ... );
  port1 direction declaration;
  port2 direction declaration;
  reg declarations;

  module body - "parallel" statements
endmodule   // note no semicolon (;) here!
```

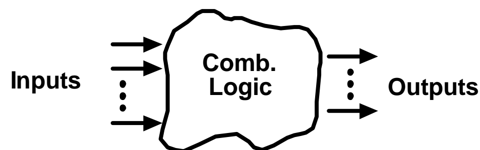
Review : Combinational Logic in Verilog

- **Combinational Logic in `assign`**

```
assign output = expression;
```

- **Combinational Logic specified in `always`**

```
always @(sensitivity-list)
begin
  out1 = expr1;
  . . .
  out2 = expr2;
  . . .
end
```



Review : Comb. Design with `always`

- **Example: 4-input mux behavioral model**

```
module mux4(d0, d1, d2, d3, s, y);
    input        d0, d1, d2, d3;
    input  [1:0] s;
    output       y;
    reg          y;

    always @(d0 or d1 or d2 or d3 or s)
        case (s)
            2'd0 : y = d0;
            2'd1 : y = d1;
            2'd2 : y = d2;
            2'd3 : y = d3;
            default : y = 1'bx;
        endcase
endmodule
```

<u>Blocking</u> assignments (immediate update)

Review : Sequential Design with `always`

- **Describe edge-triggered behavior using:**
  - **`always` block with "edge event"**

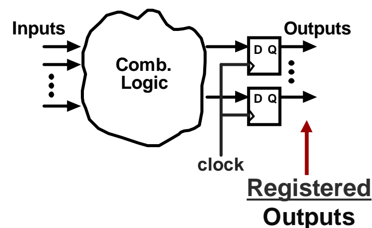
    ```
    always @(posedge clock-signal)
    always @(negedge clock-signal)
    ```

    for positive edge-trigger

    for negative edge-trigger
  - **Nonblocking assignments (<=)**

    ```
    @always(posedge clock-signal)
    begin
      output1 <= expression1;
      . . .
      output2 <= expression2;
      . . .
    end
    ```

    Non-Blocking Assignments



Registered Outputs

What is HDL-Based Design? (Revisited)

- **Verilog can specify both behavioral and structural definition of a digital system**
  - **Behavioral specification describes how a digital system functionally operates (but cannot be used for synthesis) (see Fig. 5.8.1)**
  - **Structural specification describes the detailed organization of a digital system usually using a hierarchical description (see Figs. 5.8.2 & 5.8.3)**

Behavioral Specification (Fig. 5.8.1)

```
module CPU (clock);
parameter LW = 6'b100011, SW = 6'b101011, BEQ=6'b000100, J=6'd2;
input clock; //the clock is an external input
// The architecturally visible registers and scratch registers for implementation
reg [31:0] PC, Regs[0:31], Memory [0:1023], IR, ALUOut, MDR, A, B;
reg [2:0] state; // processor state
...
//The state machine--triggered on a rising clock
always @(posedge clock) begin
    Regs[0] = 0; //make R0 0 //short-cut way to make sure R0 is always 0
   case (state) //action depends on the state
      1: begin // first step: fetch the instruction, increment PC, go to next state
         IR <= Memory[PC>>2];
         PC <= PC + 4;
         state=2; //next state
      end
  ...
```

FIGURE 5.8.1 A behavioral specification of the multicycle MIPS design. This has the same cycle behavior as the multicycle design, but is purely for simulation and specification. It cannot be used for synthesis.

Structural Specification (Fig. 5.8.3)

```
module CPU (clock);
   parameter LW = 6'b100011, SW = 6'b101011, BEQ=6'b000100, J=6'd2; //constants
   input clock; reg [2:0] state;
   wire [1:0] ALUOp, ALUSrcB, PCSource; wire [5:0] opcode;
   wire RegDst, MemRead, MemWrite, IorD, RegWrite, IRWrite, PCWrite, PCWriteCond,
          ALUSrcA, MemoryOp, IRWwrite, Mem2Reg;

assign PCWrite = (state==1) | ((state==3)&(opcode==J));
assign PCWriteCond = (state==3)&(opcode==BEQ);
assign ALUSrcA = ((state==1)|(state==2)) ? 0 :1;
assign ALUSrcB = ((state==1) | ((state==3)&(opocde==BEQ))) ? 2'b01 : (state==2) ? 2'b11 :
                   ((state==3)&MemoryOp) ? 2'b10 : 2'b00; // memory operation or other
assign PCSource = (state==1) ? 2'b00 : ((opcode==BEQ) ? 2'b01 : 2'b10);

always @(posedge clock) begin // all state updates on a positive clock edge
   case (state)
   1: state = 2;   //unconditional next state
   2: state = 3;  //unconditional next state
   3: // third step: jumps and branches complete
      state = ((opcode==BEQ) | (opcode==J)) ? 1 : 4;// branch or jump go back else next state
   4: state = (opcode==LW) ? 5 : 1; //R-type and SW finish
   5: state =1; // go back
    endcase
```

FIGURE 5.8.3 The MIPS CPU using the datapath from Figure 5.8.2.

Structural Specification (Fig. 5.8.2)

```
module Datapath (ALUOp, RegDst, MemtoReg, MemRead, MemWrite, IorD, RegWrite, IRWrite, PCWrite,
        PCWriteCond, ALUSrcA, ALUSrcB, PCSource, opcode, clock); // the control inputs + clock
 input [1:0] ALUOp, ALUSrcB, PCSource; // 2-bit control signals
input RegDst, MemtoReg, MemRead, MemWrite, IorD, RegWrite, IRWrite, PCWrite, PCWriteCond, ALUSrcA,
   clock; // 1-bit control signals
output [5:0] opcode ;// opcode is needed as an output by control


// Inputs are ALUResultOut (the incremented PC) , ALUOut (the branch address), the jump target address
   // PCSource is the selector input and PCValue is the multiplexor output
  Mult3to1 PCdatasrc (ALUResultOut,ALUOut,{PC[31:28],IR[25:0], 2'b00}, PCSource , PCValue);


   // Inputs are ALUCtl (the ALU control), ALU value inputs (ALUAin, ALUBin)
   // Outputs are ALUResultOut (the 32-bit output) and Zero (zero detection output)
MIPSALU ALU (ALUCtl, ALUAin, ALUBin, ALUResultOut,Zero); //the ALU

always @(posedge clock) begin   if (MemWrite) Memory[ALUOut>>2] <= B; // Write memory--must be a store
   ALUOut <= ALUResultOut; //Save the ALU result for use on a later clock cycle
   if (IRWrite) IR <= MemOut; // Write the IR if an instruction fetch
   MDR <= MemOut; // Always save the memory read value
// The PC is written both conditionally (controlled by PCWrite) and conditionally
   if (PCWrite || (PCWriteCond & Zero)) PC <=PCValue;
```

FIGURE 5.8.2 A Verilog version of the multicycle MIPS datapath that is appropriate for synthesis. This datapath relies on several units from Appendix B. Initial statements do not synthesize, and a version used for synthesis would have to incorporate a reset signal that had this effect. Also note that resetting R0 to 0 on every clock is not the best way to ensure that R0 stays 0; instead modifying the register file module to produce 0 whenever R0 is read and to ignore writes to R0 would be a more efficient solution.

Project #1 (Due: September 18)

- Goal: Implement ALU, ALU control and main control circuit of MIPS CPU that performs add, sub, slt, and, or, nor, lw, sw, beq, bne, and j instructions using the Icarus Verilog HDL. Their designs are shown in Figs. B.5.12, B.5.13 and 5.18, respectively.
- Modules (all combinational)
  - ALU module (alu.v)
  - ALU control (aluctrl.v)
  - Main control (control.v)
- Test program is defined in sim modules
  - A simulation module (sim_alu.v) instantiates an ALU with appropriate inputs
  - Another simulation module (sim_aluctrl.v) instantiates an ALU control with appropriate inputs
  - A third simulation module (sim_control.v) instantiates a main control with appropriate inputs
  - The last simulation module (sim_three.v) instantiates an ALU, an ALU control and a main control, connect them, and provide appropriate inputs

Project #1 (Due: September 18)

- Simulation steps
  - To compile the design with the test file:
    C:\> iverilog –o sim.o alu.v sim_alu.v
  - To run (simulate) the executable file "sim.o":
    C:\> vvp sim.o
  - Repeat with aluctrl.v and control.v instead of alu.v
  - Finally
    C:\> iverilog –o sim.o alu.v aluctrl.v control.v sim_three.v
    C:\> vvp sim.o
  - sim.v uses "$display" and "$monitor" commands