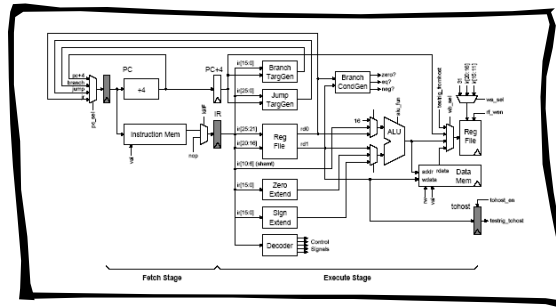
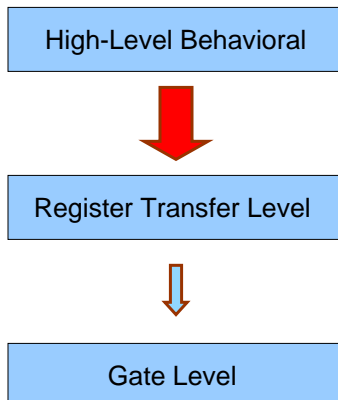


Verilog 2 - Design Examples



6.375 Complex Digital Systems
February 11, 2008

Verilog can be used at several levels



A common approach is to use C/C++ for initial behavioral modeling, and for building test rigs.

automatic tools to synthesize a low-level gate-level model.

Writing Good Synthesizable Verilog

- Use only positive-edge triggered flip-flops for state
- Do not assign the same variable from more than one always block
- Describe combinational logic using continuous assignments (`assign`) and `always@(*)` blocks with blocking assignments (`=`)

```
assign C_in = B_out + 1;

always @(*)
begin
    out = 2'd0;
    if (in1 == 1)
        out = 2'd1;
    else if (in2 == 1)
        out = 2'd2;
end
```

- Describe sequential logic using `always @(posedge clk)` and non-blocking assignments (`<=`)
- Only leaf modules should have functionality; use higher-level modules only for wiring together sub-modules

```
always @(posedge clk)
    C_out <= C_in;
```

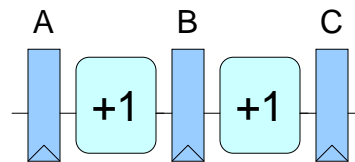
6.375 Spring 2008 • L03 Verilog 2 • 3

An example

```
wire A_in, B_in, C_in;
reg A_out, B_out, C_out;

always @(posedge clk)
begin
    A_out <= A_in;
    B_out <= B_in;
    C_out <= C_in;
end

assign B_in = A_out + 1;
assign C_in = B_out + 1;
```



The order of non-blocking assignments does not matter!

6.375 Spring 2008 • L03 Verilog 2 • 4

Another style – multiple always blocks

```

wire A_in, B_in, C_in;
reg  A_out, B_out, C_out;

always @( posedge clk )
  A_out <= A_in;

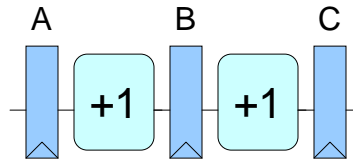
assign B_in = A_out + 1;

always @( posedge clk )
  B_out <= B_in;

assign C_in = B_out + 1;

always @( posedge clk )
  C_out <= C_in;

```



Does it have the same functionality

6.375 Spring 2008 • L03 Verilog 2 • 5

Execution semantics of Verilog

```

wire A_in, B_in, C_in;
reg  A_out, B_out, C_out;

always @( posedge clk )
  A_out <= A_in;

assign B_in = A_out + 1;

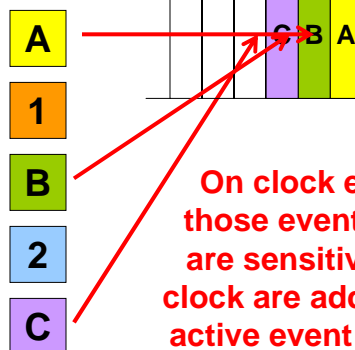
always @( posedge clk )
  B_out <= B_in;

assign C_in = B_out + 1;

always @( posedge clk )
  C_out <= C_in;

```

Active Event Queue



On clock edge all those events which are sensitive to the clock are added to the active event queue in any order!

6.375 Spring 2008 • L03 Verilog 2 • 6

Semantics of non-blocking assignment requires two event queues

```
wire A_in, B_in, C_in;
reg A_out, B_out, C_out;
```

```
always @( posedge clk )
  A_out <= A_in;
```

A

```
assign B_in = A_out + 1;
```

1

```
always @( posedge clk )
  B_out <= B_in;
```

B

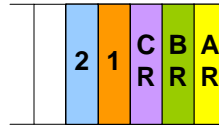
```
assign C_in = B_out + 1;
```

2

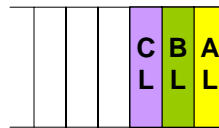
```
always @( posedge clk )
  C_out <= C_in;
```

C

Active Event Queue



Non-Blocking Queue



Variables in RHS of always blocks are not updated until all inputs (e.g. LHS + dependencies) are evaluated

6.375 Spring 2008 • L03 Verilog 2 • 7

Behavioral Verilog is richer

- Characterized by heavy use of sequential blocking statements in large always blocks
- Many constructs are **not synthesizable** but can be useful for behavioral modeling
 - Data dependent for and while loops
 - Additional behavioral datatypes: `integer`, `real`
 - Magic initialization blocks: `initial`
 - Magic delay statements: `#<delay>`
 - System calls: `$display`, `$assert`, `$finish`

6.375 Spring 2008 • L03 Verilog 2 • 8

System tasks are used for test harnesses and simulation management

```
reg [ 1023:0 ] exe_filename;

initial
begin

    // This turns on VCD (plus) output
    $vcdpluson(0);

    // This gets the program to load into memory from the command line
    if ( $value$plusargs( "exe=%s", exe_filename ) )
        $readmemh( exe_filename, mem.m );
    else
    begin
        $display( "ERROR: No executable specified! (use +exe=<filename>)" );
        $finish;
    end

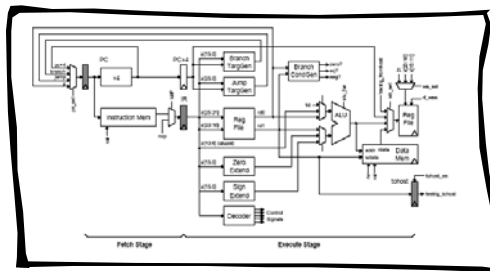
    // Strobe reset
    #0 reset = 1;
    #38 reset = 0;

end
```

6.375 Spring 2008 • L03 Verilog 2 • 9

Verilog Design Examples

- **Greatest Common Divisor**
- Unpipelined SMIPsv1 processor



6.375 Spring 2008 • L03 Verilog 2 • 10

GCD in C

```
int GCD( int inA, int inB)
{
    int done = 0;
    int A = inA;
    int B = inB;
    while ( !done )
    {
        if ( A < B )
        {
            swap = A;
            A = B;
            B = swap;
        }
        else if ( B != 0 )
            A = A - B;
        else
            done = 1;
    }
    return A;
}
```

Such a GCD description can be easily written in Behavioral Verilog

It can be simulated but it will have nothing to do with hardware, i.e. it won't synthesize.

6.375 Spring 2008 • L03 Verilog 2 • 11

Behavioral GCD in Verilog

```
module gcdGCDUnit_behav#( parameter W = 16 )
(
    input [W-1:0] inA, inB,
    output [W-1:0] out
);
    reg [W-1:0] A, B, out, swap;
    integer done;
    always @(*)
    begin
        done = 0;
        A = inA; B = inB;
        while ( !done )
        begin
            if ( A < B )
                swap = A;
                A = B;
                B = swap;
            else if ( B != 0 )
                A = A - B;
            else
                done = 1;
        end
        out = A;
    end
endmodule
```

User simply sets the input operands and checks the output; the answer will appear immediately, like a combinational circuit

Data dependent loop, "done"

6.375 Spring 2008 • L03 Verilog 2 • 12

You have to be careful in using such behavioral models

```
module exGCDTestHarness_behav;  
  
    reg [15:0] inA, inB;  
    wire [15:0] out;  
  
    exGCD_behav#(16) gcd_unit( .inA(inA), .inB(inB), .out(out) );  
  
    initial  
    begin  
  
        // 3 = GCD( 27, 15 )  
        inA = 27;  
        inB = 15;  
        #10;  
        if ( out == 3 )  
            $display( "Test ( gcd(27,15) ) succeeded, [ %x == %x ]", out, 3 );  
        else  
            $display( "Test ( gcd(27,15) ) failed, [ %x != %x ]", out, 3 );  
  
        $finish;  
  
    end  
endmodule
```

without some delay
out is bogus

6.375 Spring 2008 • L03 Verilog 2 • 13

Deriving an RTL model for GCD

```
module gcdGCDUnit_behav#( parameter W = 16 )  
(  
    input [W-1:0] inA, inB,  
    output [W-1:0] Y  
);  
  
    reg [W-1:0] A, B, Y, swap;  
    integer done;  
  
    always @(*)  
    begin  
        done = 0;  
        A = inA; B = inB;  
  
        while ( !done )  
        begin  
            if ( A < B )  
                swap = A;  
                A = B;  
                B = swap;  
            else if ( B != 0 )  
                A = A - B;  
            else  
                done = 1;  
        end  
  
        Y = A;  
    end  
endmodule
```

What does the RTL
implementation need?

State

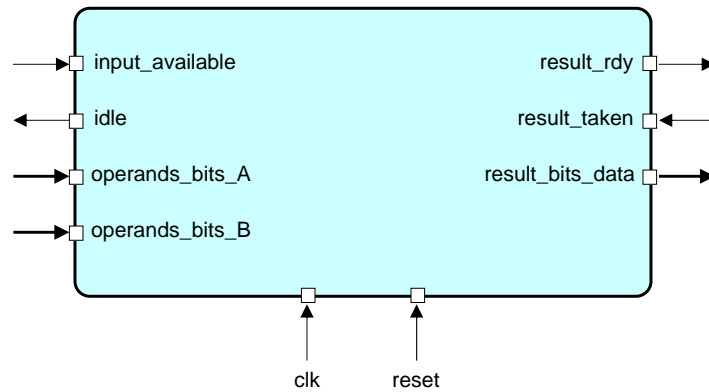
Less-Than Comparator

Equal Comparator

Subtractor

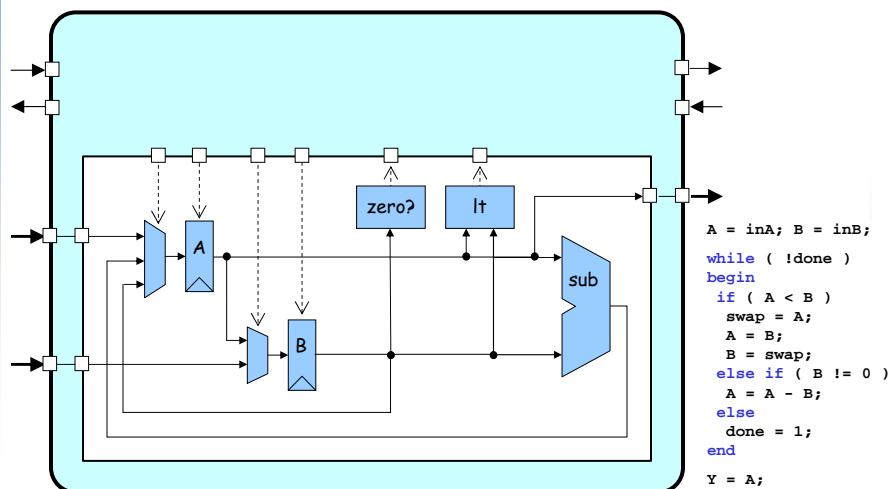
6.375 Spring 2008 • L03 Verilog 2 • 14

The first step is to carefully design an appropriate port interface



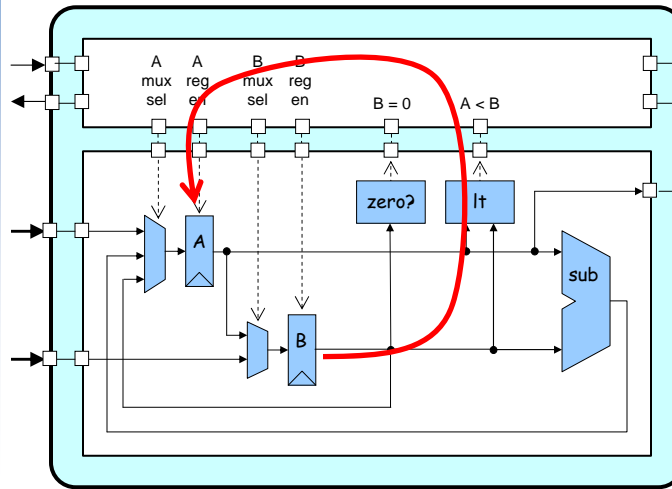
6.375 Spring 2008 • L03 Verilog 2 • 15

Next develop a datapath which has the proper functional units



6.375 Spring 2008 • L03 Verilog 2 • 16

Finally add the control unit to sequence the datapath



Control unit should be designed so that it is either busy or waiting for input or waiting for out to be picked up

```
A = inA; B = inB;
while ( !done )
begin
if ( A < B )
swap = A;
A = B;
B = swap;
else if ( B != 0 )
A = A - B;
else
done = 1;
end
Y = A;
```

6.375 Spring 2008 • L03 Verilog 2 • 17

Datapath module interface

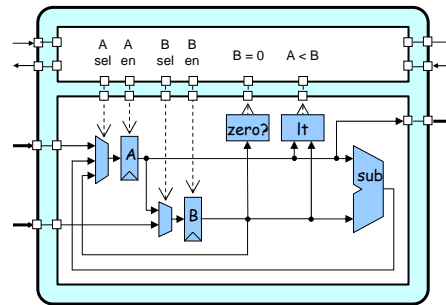
```
module gcdGCDUnitDpath_sstr#( parameter W = 16 )
(
    input    clk,

    // Data signals
    input  [W-1:0] operands_bits_A,
    input  [W-1:0] operands_bits_B,
    output [W-1:0] result_bits_data,

    // Control signals (ctrl->dpath)
    input  A_en,
    input  B_en,
    input  [1:0] A_mux_sel,
    input  B_mux_sel,

    // Control signals (dpath->ctrl)
    output B_zero,
    output A_lt_B

);
```



6.375 Spring 2008 • L03 Verilog 2 • 18

Connect the modules

```

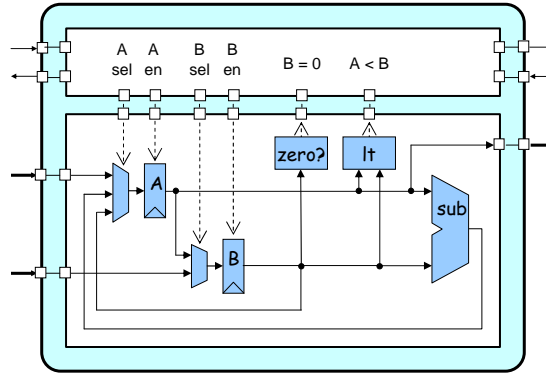
wire [W-1:0] B;
wire [W-1:0] sub_out;
wire [W-1:0] A_mux_out;

vcMux3#(W) A_mux
(
  .in0 (operands_bits_A),
  .in1 (B),
  .in2 (sub_out),
  .sel (A_mux_sel),
  .out (A_mux_out)
);

wire [W-1:0] A;

vcEDFF_pf#(W) A_pf
(
  .clk (clk),
  .en_p (A_en),
  .d_p (A_mux_out),
  .q_np (A)
);

```



6.375 Spring 2008 • L03 Verilog 2 • 19

Connect the modules ...

```

wire [W-1:0] B;
wire [W-1:0] sub_out;
wire [W-1:0] A_mux_out;

vcMux3#(W) A_mux
(
  .in0 (operands_bits_A),
  .in1 (B),
  .in2 (sub_out),
  .sel (A_mux_sel),
  .out (A_mux_out)
);

wire [W-1:0] A;

vcEDFF_pf#(W) A_pf
(
  .clk (clk),
  .en_p (A_en),
  .d_p (A_mux_out),
  .q_np (A)
);

vcMux2#(W) B_mux
(
  .in0 (operands_bits_B),
  .in1 (A),
  .sel (B_mux_sel),
  .out (B_mux_out)
);

vcEDFF_pf#(W) B_pf
(
  .clk (clk),
  .en_p (B_en),
  .d_p (B_mux_out),
  .q_np (B)
);

assign B_zero = ( B == 0 );
assign A_lt_B = ( A < B );
assign sub_out = A - B;
assign result_bits_data = A;

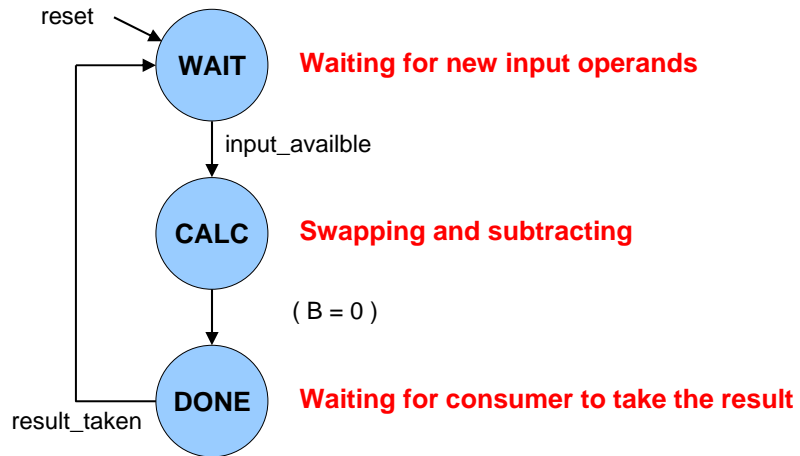
```

Using explicit state helps eliminate issues with non-blocking assignments

Continuous assignment combinational logic is fine

6.375 Spring 2008 • L03 Verilog 2 • 20

Control unit requires a simple state machine for valid/ready signals



6.375 Spring 2008 • L03 Verilog 2 • 21

Implementing the control logic finite state machine in Verilog

```
localparam WAIT = 2'd0;
localparam CALC = 2'd1;
localparam DONE = 2'd2;

reg [1:0] state_next;
wire [1:0] state;

vcrdff_pf#(2,WAIT) state_pf
(
  .clk      (clk),
  .reset_p  (reset),
  .d_p      (state_next),
  .q_np     (state)
);
```

Localparams are not really parameters at all. They are scoped constants.

Explicit state in the control logic is also a good idea!

6.375 Spring 2008 • L03 Verilog 2 • 22

Implementing the control signal outputs for the finite state machine

```

reg [6:0] cs;

always @(*)
begin

    // Default control signals
    A_mux_sel = A_MUX_SEL_X;
    A_en      = 1'b0;
    B_mux_sel = B_MUX_SEL_X;
    B_en      = 1'b0;
    input_available = 1'b0;
    result_rdy = 1'b0;

    case ( state )

        WAIT :
        ...
        CALC :
        ...
        DONE :
        ...
    endcase
end

WAIT :
begin
    A_mux_sel = A_MUX_SEL_IN;
    A_en      = 1'b1;
    B_mux_sel = B_MUX_SEL_IN;
    B_en      = 1'b1;
    input_available = 1'b1;
end

CALC :
if ( A_lt_B )
    A_mux_sel = A_MUX_SEL_B;
    A_en      = 1'b1;
    B_mux_sel = B_MUX_SEL_A;
    B_en      = 1'b1;
else if ( !B_zero )
    A_mux_sel = A_MUX_SEL_SUB;
    A_en      = 1'b1;
end

DONE :
    result_rdy = 1'b1;

```

6.375 Spring 2008 • L03 Verilog 2 • 23

Implementing the state transitions for the finite state machine

```

always @(*)
begin

    // Default is to stay in the same state
    state_next = state;

    case ( state )

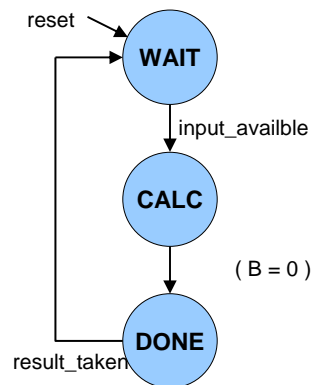
        WAIT :
            if ( input_available )
                state_next = CALC;

        CALC :
            if ( B_zero )
                state_next = DONE;

        DONE :
            if ( result_taken )
                state_next = WAIT;

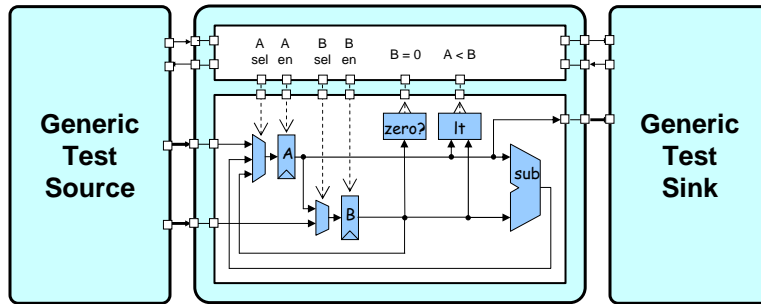
    endcase
end

```



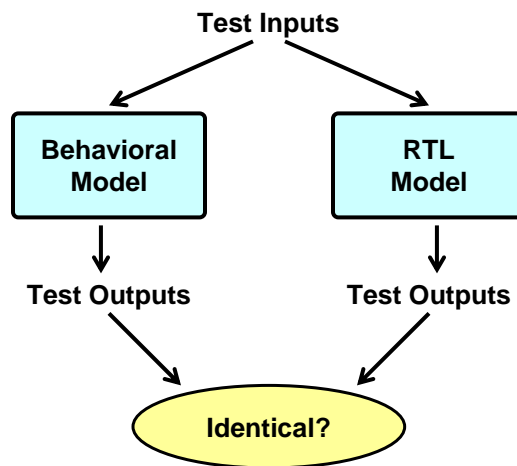
6.375 Spring 2008 • L03 Verilog 2 • 24

RTL test harness requires proper handling of the ready/valid signals



6.375 Spring 2008 • L03 Verilog 2 • 25

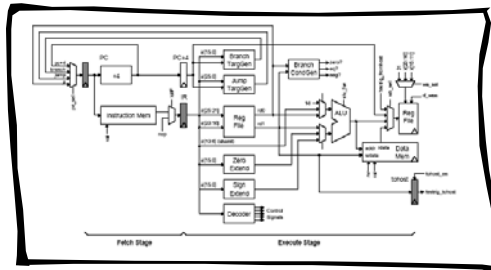
We can compare the behavioral and RTL implementations to verify correctness



6.375 Spring 2008 • L03 Verilog 2 • 26

Verilog Design Examples

- Greatest Common Divisor
- **Unpipelined SMIPsv1 processor**



6.375 Spring 2008 • L03 Verilog 2 • 27

SMIPS is a simple MIPS ISA which includes three variants

- SMIPsv1
 - 5 instructions
 - No exceptions/interrupts
 - Lecture examples
- SMIPsv2
 - 35 instructions
 - No exceptions/interrupts
 - ISA for lab assignments
- SMIPsv3
 - 58 instructions
 - Full system coproc with exceptions/Interrupts
 - Optional ISA for projects

31	26	25	21	20	16	15	11	10	6	5	0	
opcode	rs	rt	rd	shamt	func							R-type
opcode	rs	rt	immediate									I-type
opcode	target											J-type
Load and Store Instructions												
100011	base	dest	signed offset									LW rd, offset(rs)
101011	base	dest	signed offset									SW rt, offset(rs)
I-Type Computational Instructions												
001001	src	dest	signed immediate									ADDIU rt, rs, signed-imm.
001010	src	dest	signed immediate									SLTI rt, rs, signed-imm.
001011	src	dest	signed immediate									SLTIU rt, rs, signed-imm.
001100	src	dest	zero-ext.	immediate								ANDI rt, rs, zero-ext-imm.
001101	src	dest	zero-ext.	immediate								ORI rt, rs, zero-ext-imm.
001110	src	dest	zero-ext.	immediate								XORI rt, rs, zero-ext-imm.
001111	00000	dest	zero-ext.	immediate								LUI rt, zero-ext-imm.
R-Type Computational Instructions												
000000	00000	src	dest	shamt	000000							SLL rd, rt, shamt
000000	00000	src	dest	shamt	000010							SRL rd, rt, shamt
000000	00000	src	dest	shamt	000011							SRA rd, rt, shamt
000000	rshamt	src	dest	00000	000100							SLLV rd, rt, rs
		dest	00000	000110							SRLV rd, rt, rs	
			000111							SRAV rd, rt, rs		

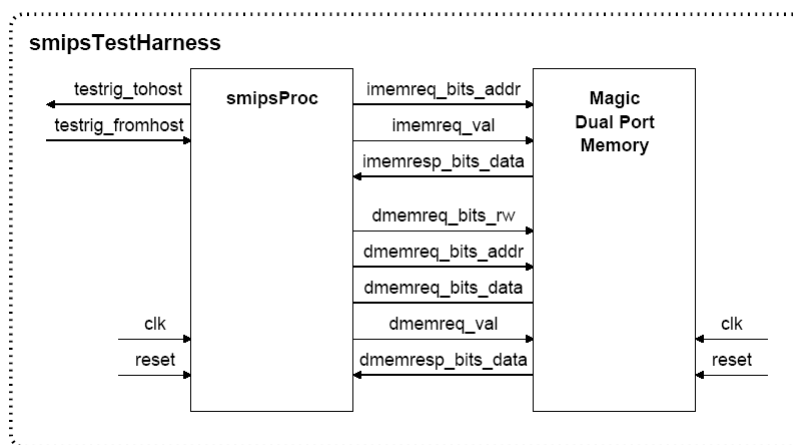
6.375 Spring 2008 • L03 Verilog 2 • 28

SMIPSV1 ISA

Instruction	Semantics	Hardware Requirements
<code>addiu rt, rs, imm</code>	$R[rt] := R[rs] + \text{sext}(imm)$	Needs adder, sext, 1w1r rf port
<code>bne rs, rt, offset</code>	if ($R[rs] \neq R[rt]$) $pc := pc + \text{sext}(offset) + 4$	Needs adder, sext, comparator, 2r rf port
<code>lw rt, offset(rs)</code>	$R[rt] := M[R[rs] + \text{sext}(offset)]$	Needs adder, sext, memory read port, 1r1w rf port
<code>sw rt, offset(rs)</code>	$M[R[rs] + \text{sext}(offset)] = R[rt]$	Needs adder, sext, memory write port, 1r1w port

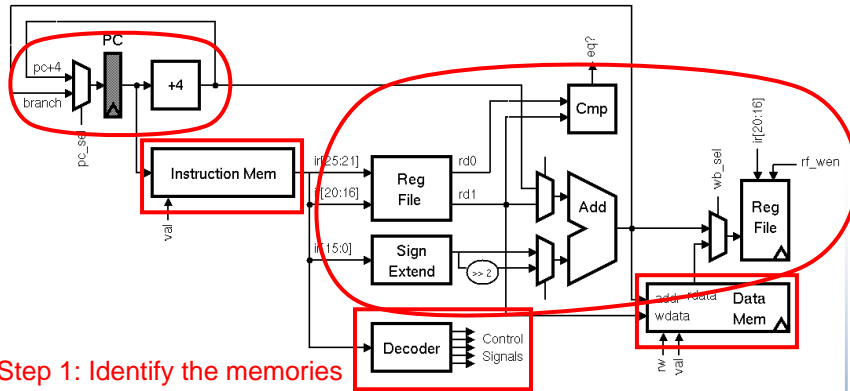
6.375 Spring 2008 • L03 Verilog 2 • 29

The first step is to carefully design an appropriate port interface



6.375 Spring 2008 • L03 Verilog 2 • 30

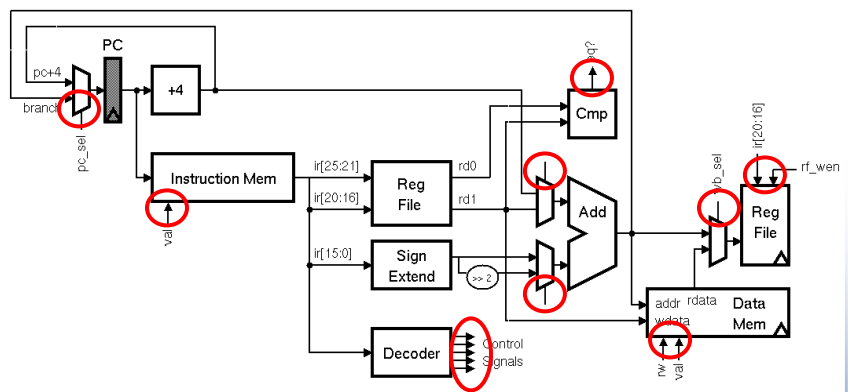
Identify memories, datapaths, and random logic



- Step 1: Identify the memories
- Step 2: Identify the datapaths
- Step 3: Everything else is random logic

6.375 Spring 2008 • L03 Verilog 2 • 31

Identify the signals to interface with the controller



6.375 Spring 2008 • L03 Verilog 2 • 32

SMIPSV1 datapath

```
module smipsProcDpath_pstr
(
    input clk, reset,
    // Memory ports
    output [31:0] imemreq_bits_addr,
    output [31:0] dmemreq_bits_addr,
    output [31:0] dmemreq_bits_data,
    input [31:0] dmemresp_bits_data,
    // Controls signals (ctrl->dpath)
    input pc_mux_sel,
    input [ 4:0] rf_raddr0,
    input [ 4:0] rf_raddr1,
    input rf_wen,
    input [ 4:0] rf_waddr,
    input op0_mux_sel,
    input op1_mux_sel,
    input [15:0] inst_imm,
    input wb_mux_sel,
    // Control signals (dpath->ctrl)
    output branch_cond_eq,
    output [7:0] tohost_next
);

    wire [31:0] branch_targ;
    wire [31:0] pc_plus4;
    wire [31:0] pc_mux_out;
    vcMux2#(32) pc_mux
    (
        .in0 (pc_plus4),
        .in1 (branch_targ),
        .sel (pc_mux_sel),
        .out (pc_mux_out)
    );
    wire [31:0] pc;
    vcRDFP_pf#(32,32'h0001000) pc_pf
    (
        .clk (clk),
        .reset_p (reset),
        .d_p (pc_mux_out),
        .q_np (pc)
    );
    assign imemreq_bits_addr = pc;
    vcInc#(32,32'd4) pc_inc4
    (
        .in (pc),
        .out (pc_plus4)
    );
```

6.375 Spring 2008 • L03 Verilog 2 • 33

Register file with two combinational read ports and one write port

```
module smipsProcDpathRegfile
(
    input clk,
    input [ 4:0] raddr0, // Read 0 address (combinational input)
    output [31:0] rdata0, // Read 0 data (combinational on raddr)
    input [ 4:0] raddr1, // Read 1 address (combinational input)
    output [31:0] rdata1, // Read 1 data (combinational on raddr)
    input wen_p, // Write enable (sample on rising clk edge)
    input [ 4:0] waddr_p, // Write address (sample on rising clk edge)
    input [31:0] wdata_p // Write data (sample on rising clk edge)
);

    // We use an array of 32 bit register for the regfile itself
    reg [31:0] registers[31:0];

    // Combinational read ports
    assign rdata0 = ( raddr0 == 0 ) ? 32'b0 : registers[raddr0];
    assign rdata1 = ( raddr1 == 0 ) ? 32'b0 : registers[raddr1];

    // Write port is active only when wen is asserted
    always @( posedge clk )
        if ( wen_p && (waddr_p != 5'b0) )
            registers[waddr_p] <= wdata_p;
endmodule
```

6.375 Spring 2008 • L03 Verilog 2 • 34

Verilog for SMIPsv1 control logic

```

`define LW      32'b100011_?????_?????_?????_?????_?????
`define SW      32'b101011_?????_?????_?????_?????_?????
`define ADDIU   32'b001001_?????_?????_?????_?????_?????
`define BNE     32'b000101_?????_?????_?????_?????_?????

localparam cs_sz = 8;
reg [cs_sz-1:0] cs;

always @(*)
begin
  cs = {cs_sz{1'b0}};
  casez ( imemresp_bits_data )
    //      op0 mux  opl mux  wb mux  rfile mreq  mreq tohost
    //      br type sel    sel    sel    wen  r/w    val  en
    `ADDIU : cs = { br_pc4, op0_sx, opl_rd0, wmx_alu, 1'b1, mreq_x, 1'b0, 1'b0 };
    `BNE   : cs = { br_neq, op0_sx2, opl_pc4, wmx_x, 1'b0, mreq_x, 1'b0, 1'b0 };
    `LW    : cs = { br_pc4, op0_sx, opl_rd0, wmx_mem, 1'b1, mreq_x, 1'b1, 1'b0 };
    `SW    : cs = { br_pc4, op0_sx, opl_rd0, wmx_x, 1'b0, mreq_w, 1'b1, 1'b0 };
    `MTCO  : cs = { br_pc4, op0_x,  opl_x,  wmx_x, 1'b0, mreq_x, 1'b0, 1'b1 };
  endcase
end

```

casez performs simple pattern matching and can be very useful when implementing decoders

6.375 Spring 2008 • L03 Verilog 2 • 35

Verilog for SMIPsv1 control logic

```

// Set the control signals based on the decoder output

wire br_type = cs[7];
assign pc_mux_sel = ( br_type == br_pc4 ) ? 1'b0
                  : ( br_type == br_neq ) ? ~branch_cond_eq
                  : 1'bx;

assign op0_mux_sel = cs[6];
assign opl_mux_sel = cs[5];
assign wb_mux_sel  = cs[4];
assign rf_wen      = ( reset ? 1'b0 : cs[3] );
assign dmemreq_bits_rw = cs[2];
assign dmemreq_val  = ( reset ? 1'b0 : cs[1] );
wire tohost_en     = ( reset ? 1'b0 : cs[0] );

// These control signals we can set directly from the instruction bits

assign rf_raddr0 = inst[25:21];
assign rf_raddr1 = inst[20:16];
assign rf_waddr  = inst[20:16];
assign inst_imm  = inst[15:0];

// We are always making an imemreq

assign imemreq_val = 1'b1;

```

6.375 Spring 2008 • L03 Verilog 2 • 36

Take away points

- Parameterized models provide the foundation for reusable libraries of components
- Use explicit state to prevent unwanted state inference and to more directly represent the desired hardware
- Begin your RTL design by identifying the external interface and then move on to partition your design into the **memories**, **datapaths**, and **control logic**