

CIS 372

Computer Organization and Design Lab

Prof. Amir Roth

Unit 2: Verilog Simulation and Debugging

CSE 372 (Martin/Roth): Simulation

1

Lab 0 Post Mortem

- How was it?
- What you should have learned
 - Xilinx error messages are not helpful
 - Logic optimization gone wild
 - Work off C:\user\ not S:\
 - Helps to draw the circuit before coding it
 - Be as specific as you can with widths of constants
 - The lecture notes are your friends

CSE 372 (Martin/Roth): Simulation

3

Plan For Today

- Lab 0 post mortem
- Lab 1 pre mortem
- Verilog simulation and debugging
 - You will need this for Lab1

CSE 372 (Martin/Roth): Simulation

2

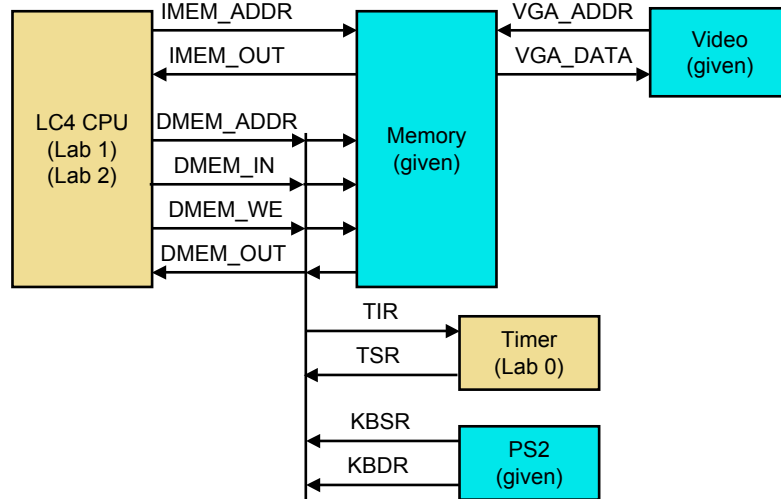
Lab 1 Pre Mortem

- Single-cycle LC4 datapath and control
 - 150–200 lines of Verilog
- Not as hard as it looks, every group finished this last year
 - But not something you can do in an afternoon
 - Mid-project milestone due next Friday
- Still hard enough that you can't debug by eyeball
 - Will have to use some debugging tools
 - Simulation (learn today)
 - Hardware debugging using the daughter-board (web docs)

CSE 372 (Martin/Roth): Simulation

4

LC4 System Block Diagram



CSE 372 (Martin/Roth): Simulation

5

Xilinx (372) != PennSim (240) LC4

- No **DIV** and **MOD** instructions
 - Xilinx will not synthesize / and % operators
 - Rewrote maelstrom not to use them
- No ASCII display device
 - Removed calls to `lc4_puts` from maelstrom
- Keyboard is not buffered
 - Fine for games, and fast enough you won't notice
- Video is not double buffered and only 128x120
 - Modified maelstrom to manually delete previous frame
 - FPGA is so fast that you don't notice
 - Haven't modified PennSim or maelstrom (yet), video is cut off
- Instructions and data in one 64K-word memory
 - Modified PennSim to do this

CSE 372 (Martin/Roth): Simulation

6

LC4 Datapath Skeleton (lc4_single.v)

```

module lc4_single (CLK, RST, GLOBAL_WE,
                  IMEM_ADDR, IMEM_IN,
                  DMEM_ADDR, DMEM_IN, DMEM_WE, DMEM_OUT);
    input CLK, RST, GLOBAL_WE;
    input [15:0] IMEM_OUT, DMEM_OUT;
    output DMEM_WE;
    output [15:0] IMEM_ADDR, DMEM_ADDR, DMEM_IN;

    // to get you started
    wire [15:0] PC, PC_IN;
    Nbit_reg #(16, 16'h8200) PC_reg(.in(PC_IN), .out(PC),
                                  .we(1'b1), .gwe(GLOBAL_WE),
                                  .rst(RST), .clk(CLK));

```

What is this?

```

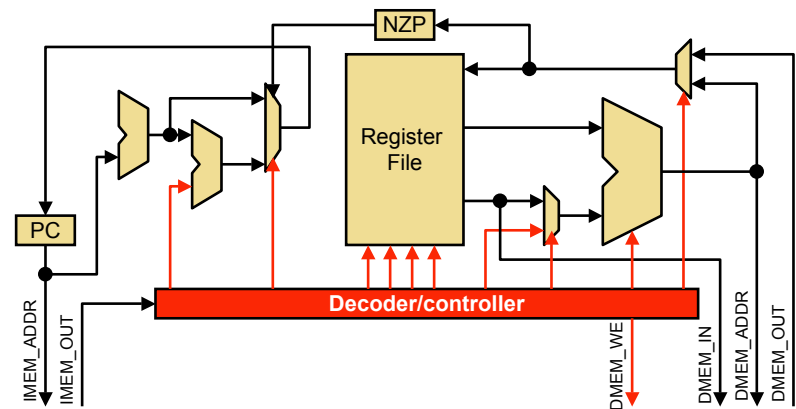
// YOUR CODE GOES HERE
endmodule // lc4_single

```

CSE 372 (Martin/Roth): Simulation

7

LC4 Single Cycle Datapath



- Caveat I: fairly incomplete
- Caveat II: other datapaths possible

CSE 372 (Martin/Roth): Simulation

8

Combinational Pieces

- Muxes
 - Easy, can use ? :
- Adders and incrementers
 - Also easy, can use +
- Sign and zero extension units
 - Just wire concatenation
- ALU
 - More involved, will have to write your own shifter
 - But can use +, *, |, &, ~, ^
 - Tricky: signed and unsigned comparisons (**CMP** vs. **CMPU**)
- **Decoder/controller**
 - This is the hard one
 - More details in a bit

CSE 372 (Martin/Roth): Simulation

9

Sequential Pieces

- PC: 16-bit Nbit_reg
- NZP: 3-bit Nbit_reg
- Register file: 8 16-bit Nbit_reg's
 - 2 read ports / 1 write port (2 reads and 1 write per cycle)

```
module lc4_regfile (R1SEL, R1DATA, R2SEL, R2DATA,
                  WSEL, WDATA, WE,
                  GWE, CLK, RST);
    input WE, GWE, CLK, RST;
    input [2:0] R1SEL, R2SEL, WSEL;
    input [15:0] WDATA; output [15:0] R1DATA, R2DATA;
    // YOUR CODE GOES HERE
endmodule
```

CSE 372 (Martin/Roth): Simulation

10

The "New" Nbit_reg

```
module Nbit_reg (out, in, we, gwe, rst, clk);
    parameter n = 1;
    parameter r = 0;

    output [n-1:0] out;
    input [n-1:0] in;
    input clk, we, gwe, rst;
    reg [n-1:0] state;
    assign #(1) out = state; // simulates propagation delay
    always @(posedge clk)
        begin
            if (gwe & rst)
                state = r;
            else if (gwe & we)
                state = in;
        end
endmodule
```

What is this?

CSE 372 (Martin/Roth): Simulation

11

Single-Cycle or Multi-Cycle?

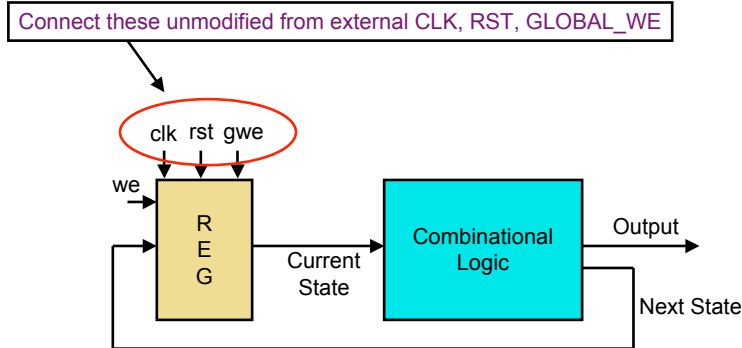
- Xilinx block RAMs (memory) only read on a clock edge
 - How do you do a single-cycle datapath?
 - How can you fetch instructions and load data in same cycle?
- Hack solution: use two clocks
 - "Big-clock" for registers (slow)
 - "Little-clock" for memory (fast)
 - 1 big-clock period = 4 little-clock periods
 - Fetch on big-clock + 1 little-clock
 - Data load on big-clock + 3 little-clock
 - Data store on big-clock
 - Implemented using "global write enable" (gwe) on registers
 - Same system used to implement single-stepping

CSE 372 (Martin/Roth): Simulation

12

372 Design Rule

- Separate combinational logic from sequential state
 - Not enforced by Verilog, but a very good idea
 - Possible exceptions: counters, shift registers



CSE 372 (Martin/Roth): Simulation

13

Clock

- The clock signals are **not** normal signals
 - Travel on dedicated "clock" wires
 - Reach all parts of the FPGA
 - Special "low-skew" routing
- Messing with the clock can cause a errors
 - Often can only be found using timing simulation
- Never do logic operations on the clocks
 - Always pass them unmodified

CSE 372 (Martin/Roth): Simulation

14

Decoder/Controller Designs

- Control signals
 - Mux selectors
 - Write enables
- How we generate them? Two styles
- **Controller**
 - Generates control signals directly (**DMEM_WE**)
- **Decoder**
 - Generate signals that can be combined into control (**IS_STORE**)
 - My personal preference
 - Makes logic more apparent in datapath
 - Can reuse (mostly) in different implementations
 - YMMV

CSE 372 (Martin/Roth): Simulation

15

Decoder Example

```
`define IS_CALL [0]
`define IS_REG_TARG [1]
`define DECODE_SIZE 2

module lc4_decoder(INSN, DECODE);
    input [15:0] INSN;
    output [DECODE_SIZE-1:0] DECODE;
    wire is_JSR = (INSN[15:11] == 5'b01001);
    wire is_JSRR = (INSN[15:11] == 5'b01000);
    wire is_JMPR = (INSN[15:11] == 5'b11000);
    wire is_TRAP = (INSN[15:12] == 4'b1111);
    assign DECODE`IS_CALL = (is_JSR | is_JSRR | is_TRAP);
    assign DECODE`IS_REG_TARG = (is_JSRR | is_JMPR);
endmodule // lc4_decoder
```

This is how you fake a "struct" in Verilog

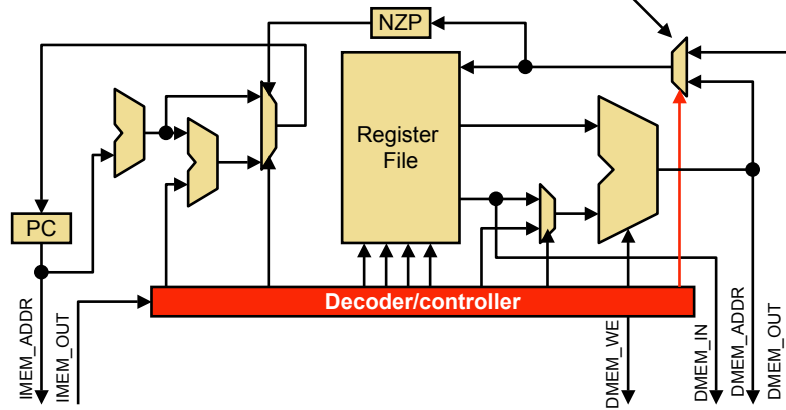
- Keep decode signals together in a wire vector
- Use macros to name signals symbolically

CSE 372 (Martin/Roth): Simulation

16

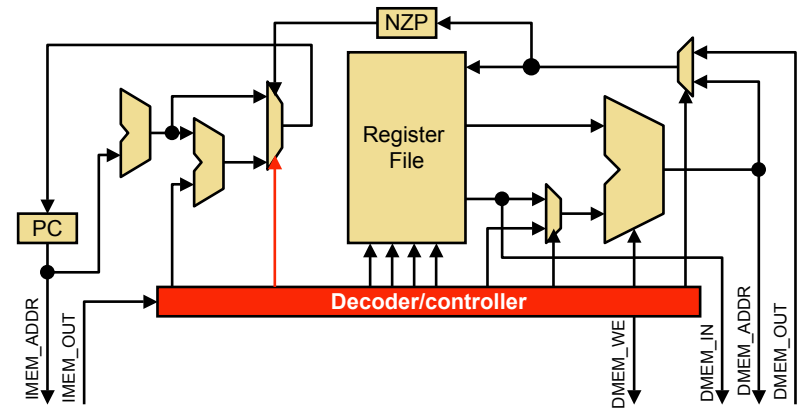
Using Decode Signals

I said this was incomplete



- Use **IS_CALL** to control destination register value mux
- ```
assign rdval = DECODE`IS_CALL ? PC + 1 ;
```

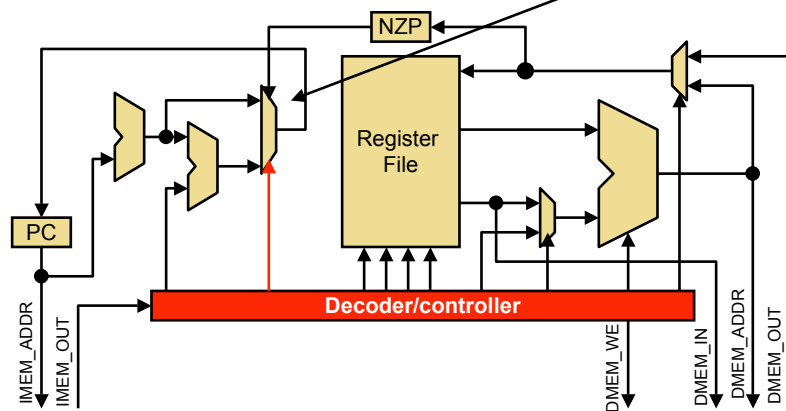
## Using Decode Signals



- Use **IS\_REG\_TARG** to control next PC mux
- ```
assign next_pc = DECODE`IS_REG_TARG ? rsval ;
```

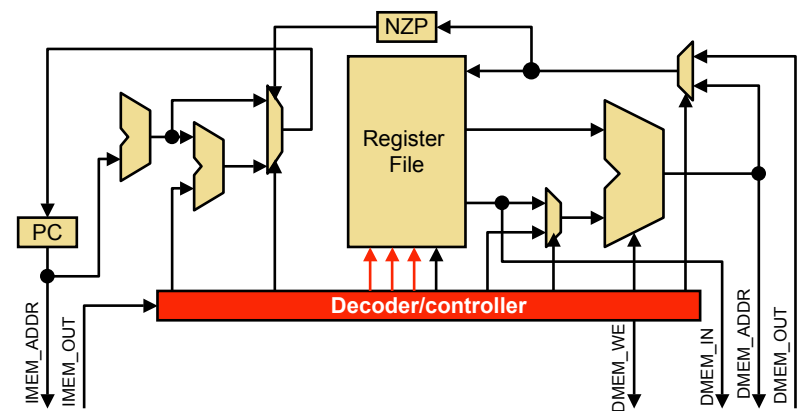
Using Decode Signals

I said this was incomplete



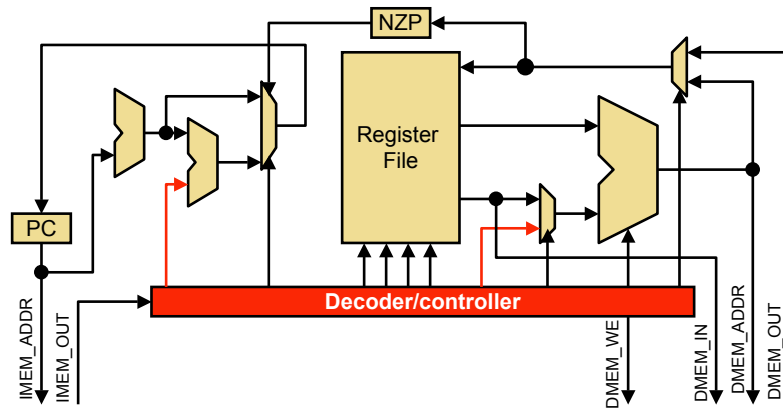
- To me, this is more readable than
- ```
assign next_pc = DECODE`NEXT_PC_MUX == 2'b00 ? rsval ;
```

## More About Decoder Design



- Are **registers specifiers** decode signals?
  - Why not? makes life easier (IMNHO)

## More About Decoder Design



- Are **immediate values** decode signals?
  - Advantage is less clear

## Testing Verilog Code

- How do you test code?
- In C/Java?
  - The true test is the program itself
  - But for a big program it helps to test smaller pieces: **unit testing**
  - Write test code in C/Java to test C/Java: **test harness**
  - Wrote one in 240 to test maelstrix (before maelstrom was written)
- In Verilog?
  - Same thing, the true test is the design on the board
  - Write test code in Verilog to test Verilog

## Debugging Verilog Code

- How do you debug code?
- In C/Java?
  - With `printf` (by printing out internal values)
  - Better yet, with a debugger (e.g., gdb)
    - Provides single-stepping, breakpoints, other useful features
- In Verilog?
  - Would be good if there were `printf` and a debugger for Verilog
  - There are!

## Synthesis and Simulation

- HDL can specify a hardware design
  - Specification can be **synthesized** ("compiled" to hardware)
- HDL can also **simulate** a hardware design
  - Also useful in development: debugging, unit testing, etc.
    - Switches and LEDs useful only for small "toy" designs
  - Can simulate things that cannot be synthesized
    - Helps too: can get design "working" quickly
  - Verilog has "SDL"-style simulation-only features
    - Types, control structures, I/O routines
  - Many free tools (e.g., Icarus) do simulation only

## A Skeleton Verilog Test Module

```
`timescale 1ns / 1ps
module test_full_adder_v;
 reg a, b, cin; // inputs
 wire s, cout; // outputs
 my_fa uut (a, b, cin, s, cout); // unit under test
 initial begin
 #125; a = 1'b1; b = 1'b0; cin = 1'b0;
 #25; a = 1'b0; b = 1'b1; cin = 1'b0;
 end
endmodule
```

- Some things we haven't seen before
  - What are `reg`, `initial`, `#100`, etc.?

## Simulated Time

```
`timescale 1ns / 1ps
module test_full_adder_v;
 reg a, b, cin; // inputs
 wire s, cout; // outputs
 my_fa uut (a, b, cin, s, cout); // unit under test
 initial begin
 #125; a = 1'b1; b = 1'b0; cin = 1'b0;
 #25; a = 1'b0; b = 1'b1; cin = 1'b0;
 end
endmodule
```

- Synthesized hardware has no notion of time:  $H \neq S$ 
  - Other than circuit delay (and clock for synchronous circuits)
- Simulated hardware does: simulated through ... time

```
`timescale <reference_time_step>/<precision_time_step>
#N // advance N reference steps
```

## Registers and Register Assignment

```
`timescale 1ns / 1ps
module test_full_adder_v;
 reg a, b, cin; // inputs
 wire s, cout; // outputs
 my_fa uut (a, b, cin, s, cout); // unit under test
 initial begin
 #125; a = 1'b1; b = 1'b0; cin = 1'b0;
 #25; a = 1'b0; b = 1'b1; cin = 1'b0;
 end
endmodule
```

- Wires `assign`'ed "continuously", can't reuse like C-variables
  - `reg`: interface-less bit (not a latch or flip-flop, no "enable" input)
    - Procedural assignment: value remains until next assignment
    - Designated by absence of `assign` keyword (like C)
    - Will synthesize (to something)

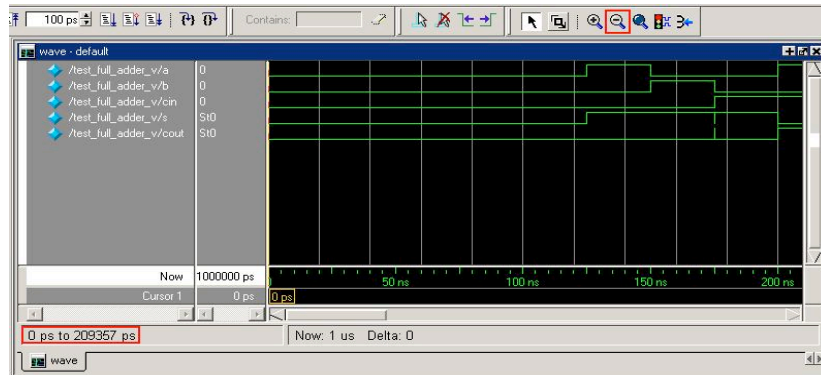
## Behavioral Invocations

```
`timescale 1ns / 1ps
module test_full_adder_v;
 reg a, b, cin; // inputs
 wire s, cout; // outputs
 my_fa uut (a, b, cin, s, cout); // unit under test
 initial begin
 #125; a = 1'b1; b = 1'b0; cin = 1'b0;
 #25; a = 1'b0; b = 1'b1; cin = 1'b0;
 end
endmodule
```

`initial`: invokes behavioral code when simulation begins

## ModelSim

- How do you get from simulation code to simulation output?
  - ModelSim: graphical output using waveforms
  - Can “drill” into modules (why modules must be named)



CSE 372 (Martin/Roth): Simulation

29

## Behavioral Simulation

- Behavioral (Functional) Simulation
  - Simulates Verilog abstractly
  - + Fast
  - No timing information, can’t detect “timing bugs”
- Errors functional simulation will catch
  - Not assigning a wire a value (will show up as Z)
  - Assigning a wire a value more than once (will show up as X)
  - Implicit wire declarations (default to type “wire” 1-bit wide)
    - Disable using ``default_nettype none`
    - Does not work with ModelSim

CSE 372 (Martin/Roth): Simulation

30

## Testing Large Designs

```
reg a, b, cin; // inputs
wire s, cout; // outputs
my_fa uut (a, b, cin, s, cout); // unit under test
initial begin
 #25; a = 1'b0; b = 1'b0; cin = 1'b0;
 #25; a = 1'b0; b = 1'b0; cin = 1'b1;
 #25; a = 1'b0; b = 1'b1; cin = 1'b0;
 #25; a = 1'b0; b = 1'b1; cin = 1'b1;
 #25; a = 1'b1; b = 1'b0; cin = 1'b0;
 #25; a = 1'b1; b = 1'b0; cin = 1'b1;
 #25; a = 1'b1; b = 1'b1; cin = 1'b0;
 #25; a = 1'b1; b = 1'b1; cin = 1'b1;
end
```

- Can completely test full-adder: only 8 input combinations
  - 8-bit adder: 64K (65,536) input combinations
  - 16-bit adder: 4M input combinations

CSE 372 (Martin/Roth): Simulation

31

## Industrial Strength Behavioral Simulation

```
reg [7:0] a, b; reg cin; // inputs
wire [7:0] s; wire cout; // outputs
my_adder8 uut (a, b, cin, s, cout);
initial begin
 cin = 1'd0;
 for (a = 8'd0; a <= 8'd255; a = a + 1)
 for (b = 8'd0; b <= 8'd255; b = b + 1)
 #25;
end
```

- New feature
  - Behavioral looping: `for (<stmt>;<expr>;<stmt>) <stmt>`
  - Careful: No ++ operator in Verilog
  - No {} to group statements either (`begin-end` instead)

CSE 372 (Martin/Roth): Simulation

32

## Oops!

```
reg [7:0] a, b; reg cin; // inputs
wire [7:0] s; wire cout; // outputs
my_adder8 uut (a, b, cin, s, cout);
initial begin
 cin = 1'd0;
 for (a = 8'd0; a <= 8'd255; a = a + 1)
 for (b = 8'd0; b <= 8'd255; b = b + 1)
 #25;
end
```

- Actually, this code has a subtle bug
  - Conditions `a <= 8'd255` and `b <= 8'd255` are always true!
  - Because `a` and `b` are 8-bit unsigned numbers
  - Loops will not terminate

## Take 2

```
integer ia, ib;
reg [7:0] a, b; reg cin; // inputs
wire [7:0] s; wire cout; // outputs
my_adder8 uut (a, b, cin, s, cout);
initial begin
 cin = 1'd0;
 for (ia = 8'd0; ia <= 8'd255; ia = ia + 1)
 for (ib = 8'd0; ib <= 8'd255; ib = ib + 1)
 begin a = ia; b = ib; #25; end
end
```

- Several ways to fix this, but here is a general one
  - Auxiliary variables: `integer ia, ib; // signed 32-bit`
  - Use these for loop control

## Auxiliary Variables

- C style variables that are used procedurally
    - Understood to be “program helpers”, not pieces of hardware
- ```
integer i; // signed 32-bit (not int)
real r; // double precision FP
time t; // unsigned 64-bit
```
- C-like arrays
- ```
integer iarray[63:0]; // array of 64 integers
```
- Example
- ```
integer i;
for (i = 0; i < 64; i = i + 1)
  iarray[i] = 0;
```

No Bug, But Still...

```
integer ia, ib;
reg [7:0] a, b; reg cin; // inputs
wire [7:0] s; wire cout; // outputs
my_adder8 uut (a, b, cin, s, cout);
initial begin
  cin = 1'd0;
  for (ia = 8'd0; ia <= 8'd255; ia = ia + 1)
    for (ib = 8'd0; ib <= 8'd255; ib = ib + 1)
      begin a = ia; b = ib; #25; end
end
```

- Still have to look through all 64K cases on wave-form?

Behavioral Statements

```
integer ia, ib, errors;
my_adder8 uut (a, b, cin, s, cout);
initial begin
    cin = 1'd0;
    errors = 0;
    for (ia = 8'd0; ia <= 8'd255; ia = ia + 1)
        for (ib = 8'd0; ib <= 8'd255; ib = ib + 1) begin
            a = ia; b = ib; #25;
            if (s != a + b) errors = errors + 1;
        end
    end
end
```

- A few new things
 - Compare (!==) structural adder with behavioral adder (+)
 - Conditional statement: if (<expr>) <stmt> else if <stmt>
 - + Now can just look at wave-form value of errors

It Gets Better

```
integer ia, ib, errors;
my_adder8 uut (a, b, cin, s, cout);
initial begin
    cin = 1'd0;
    errors = 0;
    for (ia = 8'd0; ia <= 8'd255; ia = ia + 1)
        for (ib = 8'd0; ib <= 8'd255; ib = ib + 1) begin
            a = ia; b = ib; #25;
            if (s != a + b) errors = errors + 1;
        end
    $display("Simulation finished with %d errors", errors);
end
```

- **VPI: Verilog Programming Interface**
 - Start with \$
 - Verilog equivalent of C library
 - VPI \$display analog of C-library printf

You Want More? You Got It

```
integer ia, ib, errors;
my_adder8 uut (a, b, cin, s, cout);
initial begin
    cin = 1'd0;
    errors = 0;
    for (ia = 8'd0; ia <= 8'd255; ia = ia + 1)
        for (ib = 8'd0; ib <= 8'd255; ib = ib + 1) begin
            a = ia; b = ib; #25;
            if (s != a + b) begin
                errors = errors + 1;
                $display("Error at time %dns", $time);
                $finish;
            end
        end
    end
end
```

- Variable \$time: simulator time (64-bit unsigned)
- Command \$finish: terminate simulation
- Command \$stop: break simulation (can continue)

Testing The Entire Processor

- Need a little bit more to test the entire processor
 - First thing you need is a program to test
 - Open file include/bram.v (memory module)
 - You will see this line at the top

```
`define MEMORY_IMAGE_FILE "code/maelstrom.hex"
```
 - And these lines inside the memory module

```
reg[15:0] RAM [65535:0];
initial begin
    $readmemh(`MEMORY_IMAGE_FILE, RAM, 0, 65535);
end
```
 - The first line is how you define a memory in verilog
 - The second is how you define its initial contents
 - Xilinx embeds this into the .bit programming file
 - Change MEMORY_IMAGE_FILE to test different programs

Creating Test Programs

- I will give you a memory image for maelstrom
 - You can use PennSim to create images of smaller programs
 - First: write a small program in LC4 assembly
 - Second: assemble using PennSim `as` command
 - Third: load into PennSim memory using `ld` command
 - Fourth: create memory image using PennSim `dump` command
 - Example using file `test1.asm`

```
as test1 test1
ld test1
dump -readmemh 0 xFFFF test1.hex
```

CSE 372 (Martin/Roth): Simulation

41

Full Processor Test Harness

```
// Global signals
reg CLK, RST, GWE;
// Processor-memory interconnect
wire DMEM_WE;
wire [15:0] IMEM_ADDR, IMEM_OUT, DMEM_ADDR, DMEM_OUT, DMEM_IN;

// Instantiate processor and memory, but not devices
bram memory (.idclk(CLK ),
             .iaddr(IMEM_ADDR), .iout(IMEM_OUT),
             .daddr(DMEM_ADDR), .dout(DMEM_OUT),
             .din(DMEM_IN), .dwe(DMEM_WE));

lc4_single lc4 (.CLK(CLK), .RST(RST), .GLOBAL_WE(GWE),
               .IMEM_ADDR(IMEM_ADDR), .IMEM_OUT(IMEM_OUT),
               .DMEM_ADDR(DMEM_ADDR), .DMEM_IN(DMEM_IN),
               .DMEM_OUT(DMEM_OUT), .DMEM_WE(DMEM_WE));
```

CSE 372 (Martin/Roth): Simulation

42

Full Processor Test Harness (Cont'd)

```
// generate clock and global write enable signal
always #5 CLK <= ~CLK;
always #10 GWE <= ~GWE;

// Initialize system
initial begin
    CLK = 0;
    GWE = 0;
    RST = 0;
    // Reset
    #20; RST = 1;
    #125; RST = 0;
end
```

CSE 372 (Martin/Roth): Simulation

43

Full Processor Test Harness (Cont'd)

```
// At every "big clock" edge
always @(posedge GWE)
begin
    // Display instruction PC and bits
    $display("PC: %h INSN: %h", IMEM_ADDR, IMEM_OUT);
    // Break, wait for continue
    $stop;
end
```

- That's it
 - If you want to display other "variables"
 - Pull them out of `lc4_single`
 - Or embed an `always` block within `lc4_single`

CSE 372 (Martin/Roth): Simulation

44

Testing with ICARUS

- We have an installation of **ICARUS**
 - Free Verilog simulator (but not synthesis tool)
 - Can use it to simulate and debug at home (rather than in lab)
 - Two step process

- First: compile with `iverilog`

```
> ~cse372/public-bin/iverilog -otest test.tf
```

- Then: run with `vvp`

```
> ~cse372/public-bin/vvp test
```

Coding and Compiling for ICARUS

- ICARUS is not an integrated project environment
 - Have to manually include files
 - E.g., for single-cycle test fixture

```
`ifdef ICARUS
`include "include/bram.v"
`include "include/clock_util.v"
`include "lc4_single.v"
`endif
```

- Then on the compilation command line

```
> ~cse372/public-bin/iverilog -DICARUS -otest test.tf
```

Board Testing and Debugging

- Board (and daughterboard) have test/debug features
 - "Big clock" single-stepping
 - View various values on hex display
 - See web tutorial on this

Lab 1 Implementation and Test Plan

- First milestone
 - Implement and test regfile, demo
 - Pencil and paper drawing of datapath
 - All paths labeled
 - Values of all control signals for the following instructions
 - ADD, HICONST, CMPU, LDR, STR, BRn, JSR, TRAP
 - Due Friday 2/20, 6:30pm
- Final project
 - Implement and test LC4 datapath, demo on maelstrom
 - Final writeup
 - Due Friday 2/27, 6:30pm