

# ECE 385 – Digital Systems Laboratory

Lecture 11 – Experiment 6.2 – SLC-3 CPU  
Zuofu Cheng

Spring 2018

[Link to Course Website](#)



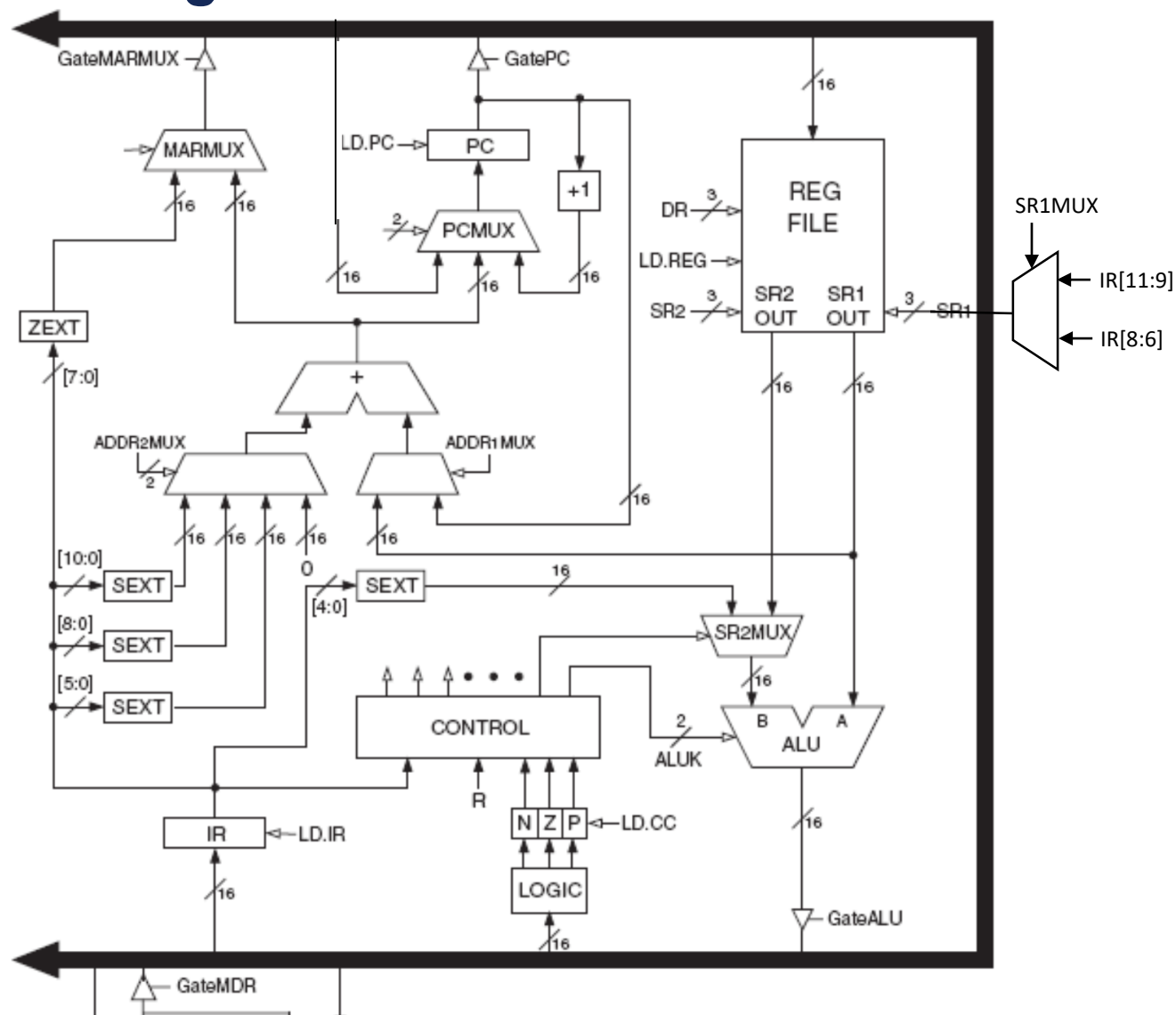
# Week 2 Demo

- Basic I/O Test 1. (1.0 point)
  - Displays LED on switches (continuous)
- Basic I/O Test 2. (0.5 points)
  - Displays LED on switches, pause and wait for continue
- Self-Modifying Code Test. (1.0 point)
- Multiplication Test. (1.0 point)
- Sort Test. (1.0 point)
- Correct “Act Once” Behavior. (0.5 points)
  - Note: DE2 Control Panel application is unreliable - program at least two times
  - Alternative: Use the SD Card SRAM programmer (reliable, only need to program once)

# SLC-3 ISA – Subset of LC-3 ISA

Instruction	Instruction(15 downto 0)										Operation
ADD	0001	DR	SR1	0	00	SR2					$R(DR) \leftarrow R(SR1) + R(SR2)$
ADDi	0001	DR	SR	1	imm5						$R(DR) \leftarrow R(SR) + \text{SEXT}(\text{imm5})$
AND	0101	DR	SR1	0	00	SR2					$R(DR) \leftarrow R(SR1) \text{ AND } R(SR2)$
ANDi	0101	DR	SR	1	imm5						$R(DR) \leftarrow R(SR) \text{ AND } \text{SEXT}(\text{imm5})$
NOT	1001	DR	SR		11111						$R(DR) \leftarrow \text{NOT } R(SR)$
BR	0000	N	Z	P	PCOffset9						if ((nzp AND NZP) != 0) $PC \leftarrow PC + 1 + \text{SEXT}(\text{PCOffset9})$
JMP	1100	000	BaseR		000000						$PC \leftarrow R(\text{BaseR})$
JSR	0100	1	PCOffset11								$R(7) \leftarrow PC + 1;$ $PC \leftarrow PC + 1 + \text{SEXT}(\text{PCOffset11})$
LDR	0110	DR	BaseR		offset6						$R(DR) \leftarrow M[R(\text{BaseR}) + \text{SEXT}(\text{offset6})]$
STR	0111	SR	BaseR		offset6						$M[R(\text{BaseR}) + \text{SEXT}(\text{offset6})] \leftarrow R(SR)$
PAUSE	1101	ledVect12									$\text{LEDs} \leftarrow \text{ledVect12}; \text{ Wait on Continue}$

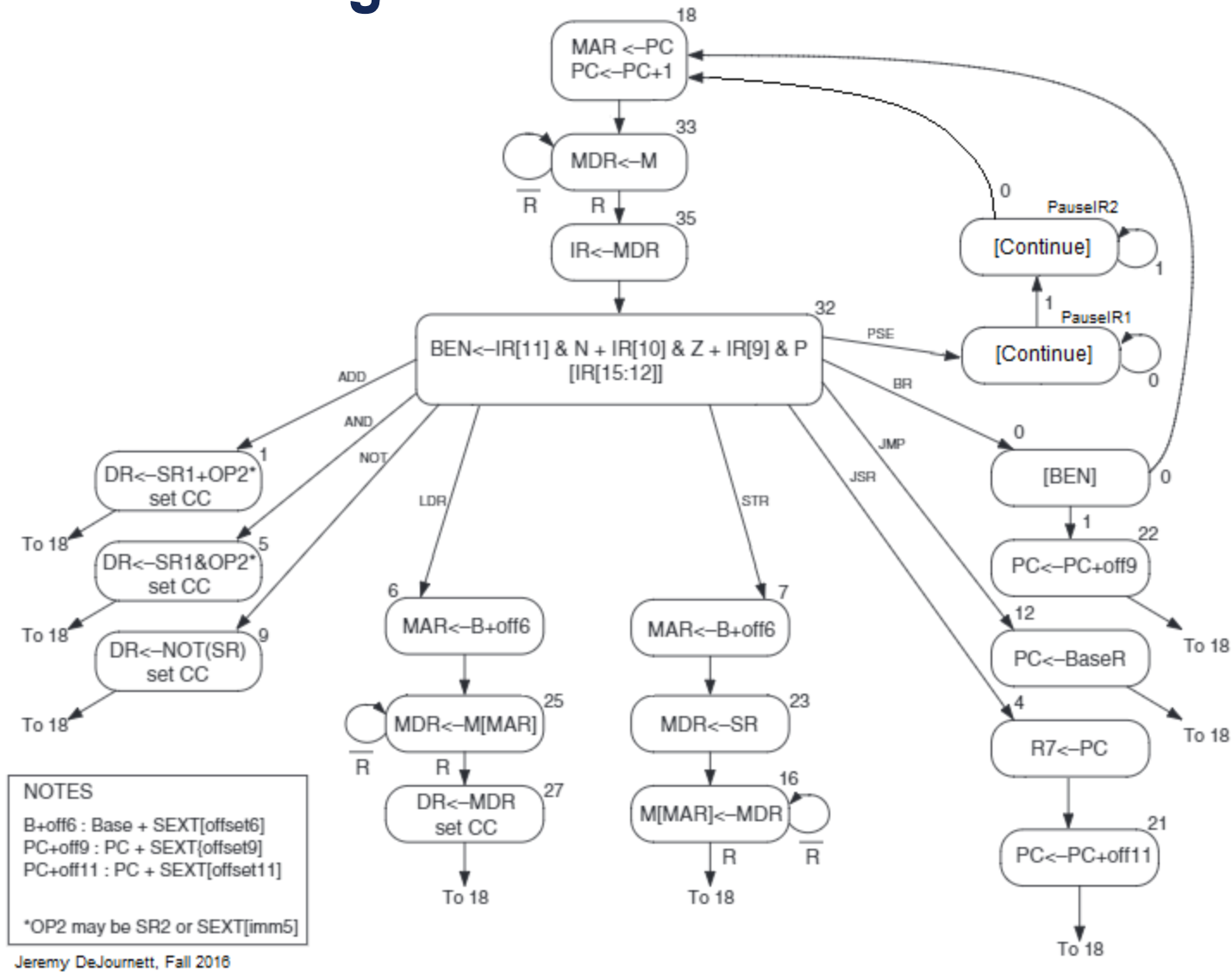
# SLC-3 Block Diagram



MDR is down here

*For complete diagram check out the online mater*

# Updated State Diagram



# FETCH Phase

- state1:  $MAR \leftarrow PC$
- state2:  $MDR \leftarrow M(MAR)$ ; -- *assert Read Command on the RAM*
- state3:  $IR \leftarrow MDR$ ;  
 $PC \leftarrow PC + 1$ ; -- "+1" inserts an incrementer/counter instead of an adder.

Go to decode

## More details:

- $MAR \leftarrow PC$ ;  $MAR$  = memory address to read the instruction from
- $MDR \leftarrow M(MAR)$ ;  $MDR$  = Instruction read from memory
- $IR \leftarrow MDR$ ;  $IR$  = Instruction to decode
- $PC \leftarrow (PC + 1)$

# EXECUTE Stage for Load/Store

## LOAD:

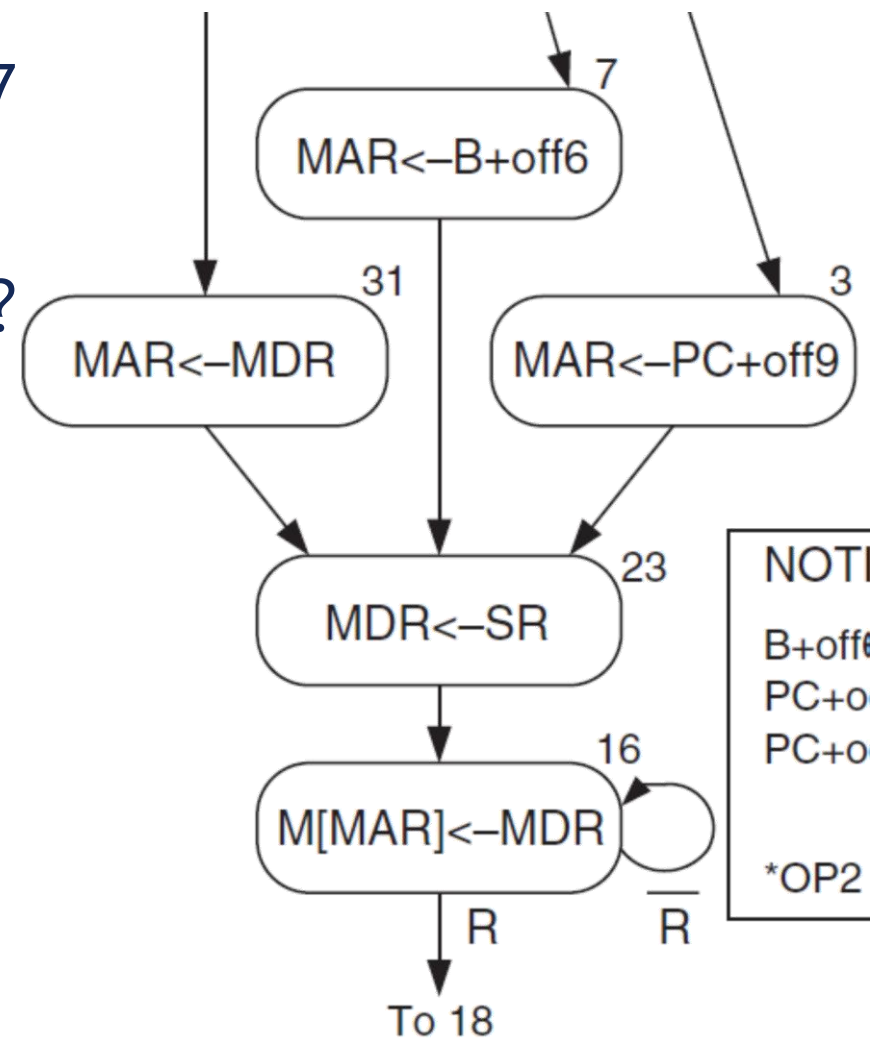
- Load\_state1:  $MAR \leftarrow (BaseR + SEXT(offset6))$  from ALU & AGU
- Load\_state2:  $MDR \leftarrow M(MAR)$ ; -- *assert Read Command on the RAM*
- Load\_state3:  $R(DR) \leftarrow MDR$ ;

## STORE:

- Store\_state1:  $MAR \leftarrow (BaseR + SEXT(offset6))$  from ALU & AGU
- Store\_state2:  $MDR \leftarrow R(SR)$
- Store\_state3:  $M(MAR) \leftarrow MDR$ ; -- *assert Write Command on the RAM*

# State Diagram: Store

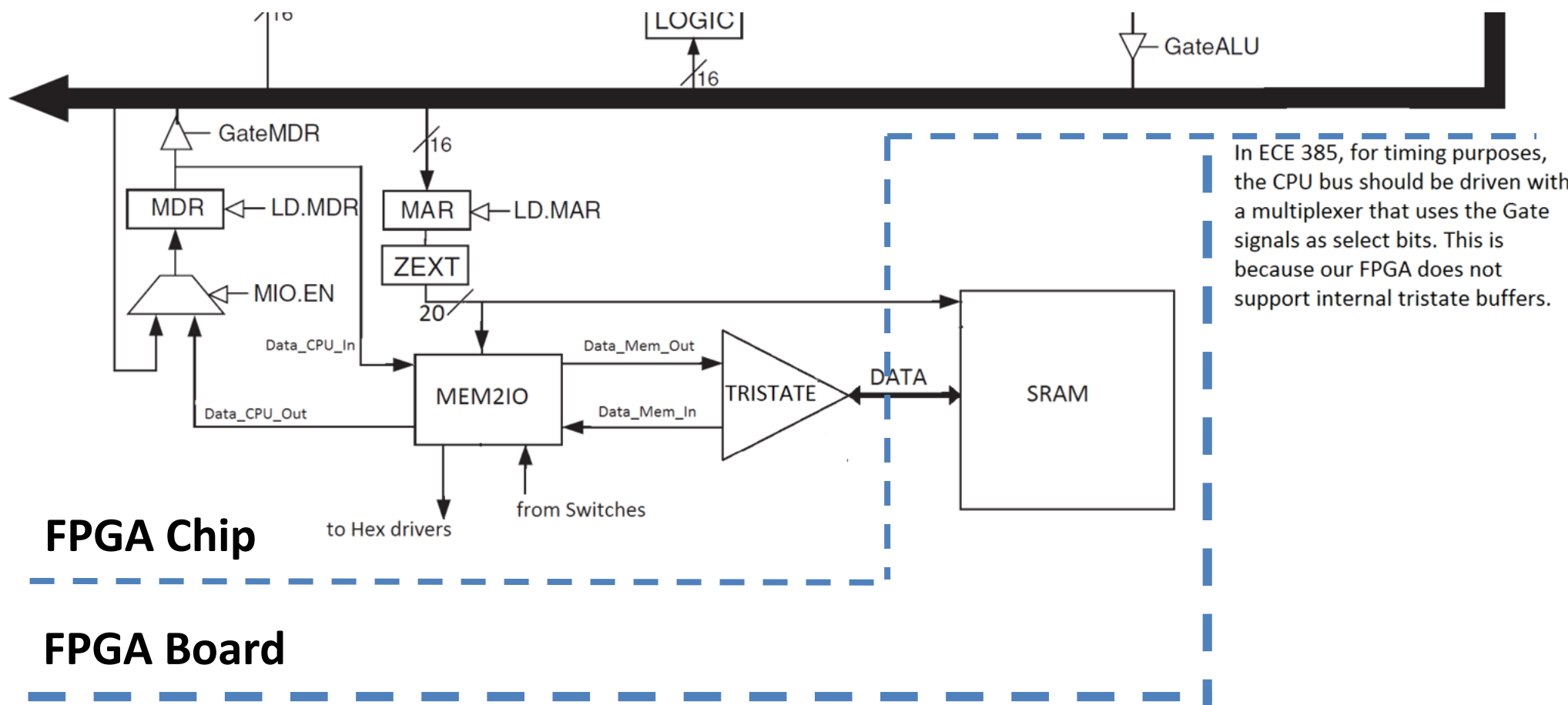
- Store Decodes into state 7
- What is R?
- How many cycles are in R?





# CPU to SRAM Configuration

- CPU to SRAM with Physical Memory



# Interfacing: I/O addressing

- A microprocessor communicates with other devices using some of its pins
  - Port-based I/O (parallel I/O)
    - Processor has one or more N-bit ports
    - Processor's software reads and writes a port just like a register
    - E.g.,  $P0 = 0xFF$ ;  $v = P1.2$ ; -- P0 and P1 are 8-bit ports
    - Often has special instructions for I/O
  - Bus-based I/O
    - Processor has address, data and control ports that form a single bus
    - Communication protocol is built into the processor
    - A single instruction carries out the read or write protocol on the bus
    - Often looks the same as Load/Store

# Memory Mapped Bus-Based I/O

- We are implementing memory mapped bus based I/O
- Primary function of the Mem2IO block
  - Detects load from address 0xFFFF as a load from switches (instead of real SRAM word 0xFFFF)
  - Detects store to address 0xFFFF as store out to HEX displays (stores this value in internal register and displays it instead of storing to SRAM)
- Note:
  - Don't need to implement any special instructions for I/O (Load/Store sufficient)
  - Things get potentially messy if CPU has cache (won't have to deal with for now)

# A Reminder About Endianness

- All LC-3 CPUs (LC-3, LC-3b, SLC-3) are little endian
- Little Endian -> least significant byte stored at lowest address
- Big Endian -> most significant byte stored at lowest address
- This may result in bytes appearing to be in wrong order when viewing on hex editor

# Implementing Register File

- Can use 8 16-bit registers (extend reg8 to reg16)
- What goes to D (data input port), what goes to DR?
- What about Q1, Q2 (data output ports), SR1, SR2?
- Alternatively, can use unpacked array of packed logic:
  - `logic [15:0] reg_file [8];`
- Different approaches illustrate difference between behavioral HDL and structural HDL

# Understanding Test Programs

- Make sure you understand the test programs for Week 2
- Note: you can write your own test programs by using `test_memory.sv` and `slc3_2.sv`
  - `slc3_2.SV` is a SystemVerilog library which has un-synthesizable functions which act as an assembler
  - This allows `test_memory.sv` to have assembly language syntax
  - Note: all `test_memory.sv` has same contents as RAM image

```
mem_array[ 0 ] <= opCLR(R0) ; // Clear the
    register so it can be used as a base
mem_array[ 1 ] <= opLDR(R1, R0, inSW) ; // Load switches
mem_array[ 2 ] <= opJMP(R1) ; // Jump to the
    start of a program
```

# Basic I/O Test 1

- Starts at address 0x03
  - Should be able to set switch to 3 and reset to run Basic I/O Test 1
  - Note – all SLC-3 addresses are word addresses
  - LC-3 addresses are byte addressable (hence PC+2 on some other LC-3 materials)
- I/O Test displays contents of switches on HEX displays
  - Tests load from 0xFFFF (-1 sign extended – maps to switches)
  - Tests store to 0xFFFF (-1 sign extended – maps to HEX displays)

```
mem_array[ 3 ] <=    opLDR(R1, R0, inSW)        ;        // Load switches
mem_array[ 4 ] <=    opSTR(R1, R0, outHEX)       ;        // Output
mem_array[ 5 ] <=    opBR(nzp, -3)              ;        // Repeat
```

# Another Test Program (Basic I/O Test 2)

- Make sure you understand the test programs for Week 2 Basic I/O Test 2
- The first pause instruction (checkpoint 1) will ask for input on the switches
- The second and subsequent pauses will display x02 on the LED, and ask for input and report that an output is present on the HEX display
- When operating correctly, each press of the Continue button will transfer the value from the switches to the HEX display, but the HEX display will not change until Continue is pressed

```
mem_array[ 6 ] <=      opPSE(12'h801)          ;          // Checkpoint 1 -  
    prepare to input  
mem_array[ 7 ] <=      opLDR(R1, R0, inSW)      ;          // Load switches  
mem_array[ 8 ] <=      opSTR(R1, R0, outHEX)    ;          // Output  
mem_array[ 9 ] <=      opPSE(12'hC02)          ;          // Checkpoint 2 -  
    read output, prepare to input  
mem_array[ 10 ] <=     opBR(nzp, -4)            ;          // Repeat
```

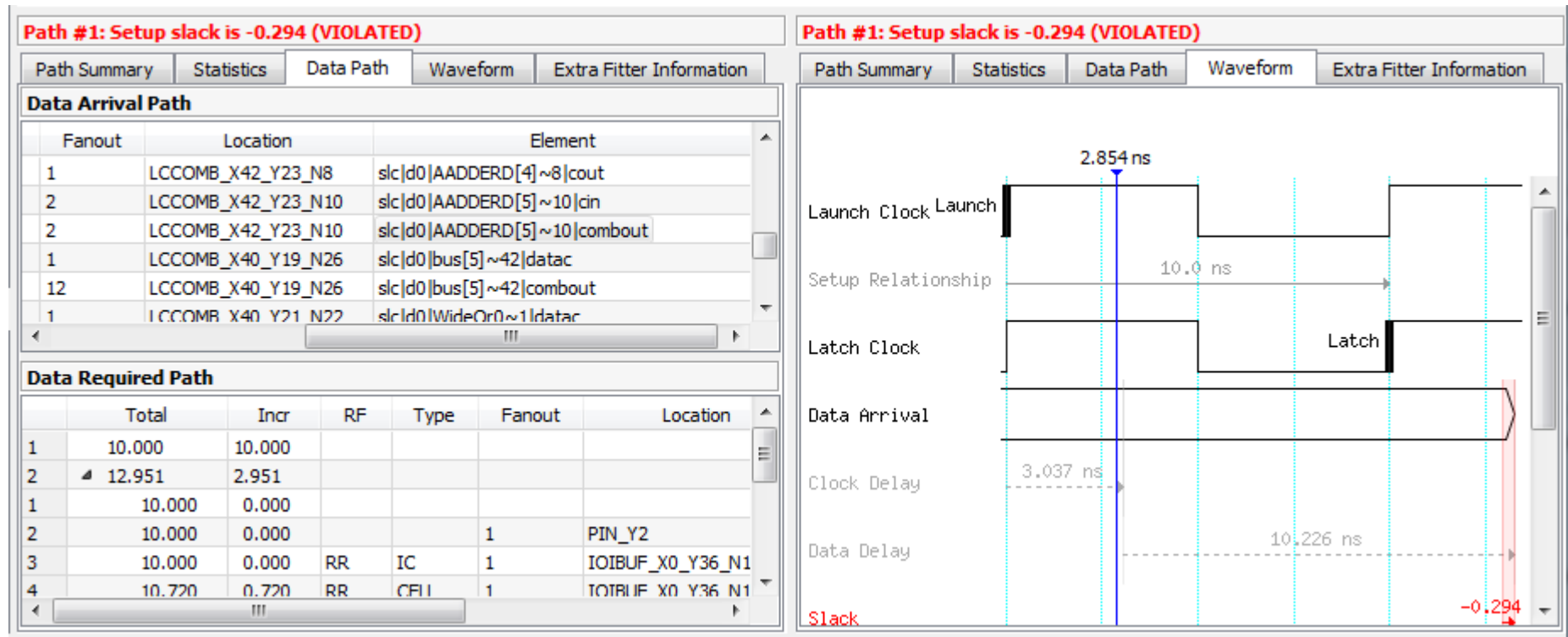


# Hints for Meeting Timing Constraints

- Make sure you have .SDC file with clock constraint
  - `create_clock -period 20 [get_ports clk]`
- Note that place & route tool will only do minimum effort necessary to pass timing
  - If  $f_{\max}$  is very close to 50 MHz, try increasing target clock constraint
  - Increasing target clock speed too much will cause timing to fail or cause place & route to take a long time to complete
- Only optimize combinational logic by hand if timing fails
  - Take a look at *Advanced Synthesis Cookbook* for ideas
  - Should be unnecessary, since data-path is specified in detail
  - Most common reason for missing timing is misunderstanding of data-path (e.g. putting combinational logic where there should be a register)

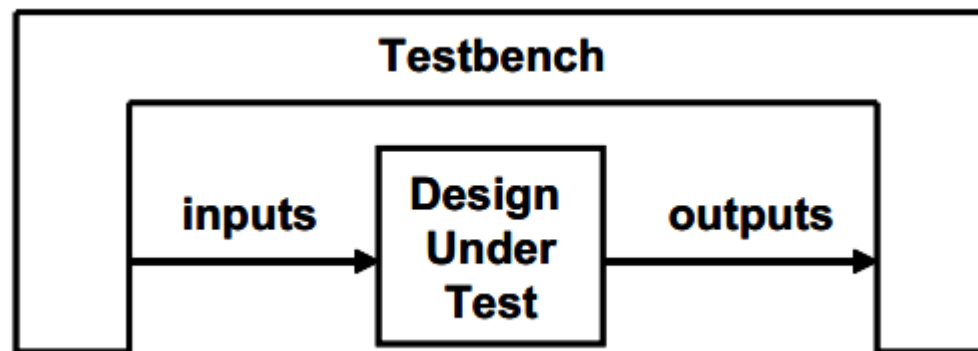
# Hints for Meeting Timing Constraints

- TimeQuest will give you list of worst slack times (negative slack = failed timing)
- Optimize worst paths by hand by tracing through and following on RTL
- Turns out in this case, delay is in address generation



# SystemVerilog Test-benches

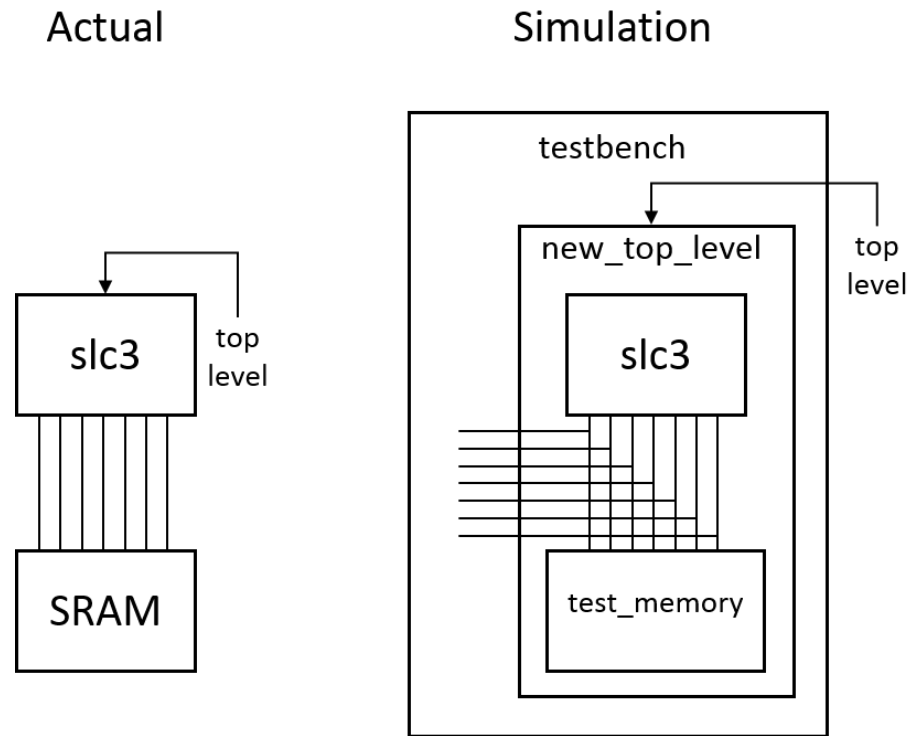
- Test-bench is used to determine the correctness of the Design Under Test (DUT)
  - Generate stimulus
  - Apply stimulus to the DUT
  - Capture the response
  - Check for correctness
  - Measure progress against the overall verification goals
- Test-bench wraps around the DUT just as a hardware tester connects to a physical chip.



[1] C. Spear, SystemVerilog for Verification. New York, NY: Springer, 2008.

# SystemVerilog Test-benches

- When simulating (only when simulating) you will use test\_memory.sv
- Make a separate top-level for simulation which includes the regular SLC3 (datapath, control, tristate, mem2IO) and additionally instantiates test\_memory
- Otherwise ModelSim may complain about dependencies



# Flat vs. Layered Test-bench

- Flat test-bench

```
initial begin
    Reset = 0;
    LoadA = 1;

    #2 Reset = 1;

    #2 LoadA = 0;
    #2 LoadA = 1;
    .....
end
```

Similar to unit testing a single module

- Layered test-bench

- Break down into tasks
- Reusable
- Modularized
- Scalable

- Note: both allow you to use the large portion of non-synthesizable SystemVerilog!

# Example Flat Test-bench (Register File)

```
module RegisterFileTest();

    timeunit 10ns; // Half clock cycle at 50 MHz
                  // This is the amount of time represented by #1

    timeprecision 1ns;

    // Internal signals
    logic Clk = 0;
    logic [2:0] SR1, SR2, DR;
    logic Reset, LD_REG;
    logic [15:0] D, SR1_OUT, SR2_OUT;

    // A counter to count the instances where simulation results
    // do not match with expected results
    integer ErrorCnt = 0;

    // Instantiating the DUT
    RegisterFile r0(.*);

    // Toggle the clock
    // #1 means wait for a delay of 1 timeunit
    always #1 Clk = ~Clk;

    initial begin
        Reset = 1;
        SR1 = 3'd0;
        SR2 = 3'd0;
        DR = 3'd0;
        LD_REG = 1'b0;
        D = 0;
        #2 Reset = 0;

        DR = 3'd0;
        D = -1;
        #2 LD_REG = 1'b1;
        #2 LD_REG = 1'b0;

        DR = 3'd1;
        D = -2;
        #2 LD_REG = 1'b1;
        #2 LD_REG = 1'b0;

        DR = 3'd2;
        D = -3;
        #2 LD_REG = 1'b1;
        #2 LD_REG = 1'b0;

        DR = 3'd3;
        D = -4;
        #2 LD_REG = 1'b1;
        #2 LD_REG = 1'b0;

        #2 SR1 = 3'd0; SR2 = 3'd1;
        #2 SR1 = 3'd2; SR2 = 3'd3;
        #2 SR1 = 3'd4; SR2 = 3'd5;
        #2 SR1 = 3'd6; SR2 = 3'd7;
        #2 SR1 = 3'd0; SR2 = 3'd0;

        if (ErrorCnt == 0)
            $display("Success!");
        else
            $display("Try again!");
        $stop;
    end
endmodule
```

# Delay Token (#)

- Delay token (#) is used to model delays in simulation
- Delay token is not synthesizable (it is simply ignored by synthesis tool, will still successfully synthesize)
- Important to understand simulation workflow
- **Note: different procedures (always\_comb, always\_ff, always) run concurrently in unknown order!**
- Each simulation timeunit (is divided into several queues)
- First queue: (can happen in any order, but always before the second queue)
  - All RHS of non-blocking assignments are evaluated
  - All RHS and LHS of blocking and continuous (assign procedure) assignment
  - Inputs and outputs evaluated
  - \$display and \$write processed
- Second queue: (happens after first queue)
  - Change LHS of all non-blocking assignment
- See (Understanding Verilog Blocking Understanding Verilog Blocking and Non and Non -blocking Assignments – S. Sutherland)

# When do the assignments happen?

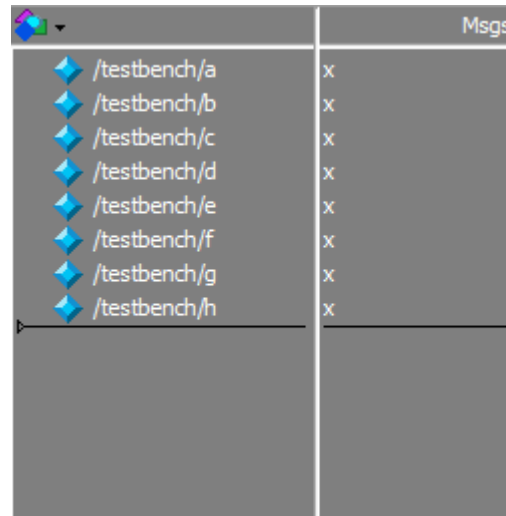
- Which one is different, and why?

```
initial begin: TEST_A
    #10 a = 1'b1;
    #20 b = 1'b0;
end
```

```
initial begin: TEST_B
    #10 c <= 1'b1;
    #20 d <= 1'b0;
end
```

```
initial begin: TEST_C
    e = #10 1'b1;
    f = #20 1'b0;
end
```

```
initial begin: TEST_D
    g <= #10 1'b1;
    h <= #20 1'b0;
end
```



	Msgs
/testbench/a	x
/testbench/b	x
/testbench/c	x
/testbench/d	x
/testbench/e	x
/testbench/f	x
/testbench/g	x
/testbench/h	x



# Monitoring Internal Signals

- Sometimes useful to be able to monitor/force internal signals
- Prevents the need to break out “test signals” from interior modules
- Hierarchical references are addressed using . (period)

```
logic ALU_out;  
always_comb begin: INTERNAL_MONITORING  
    ALU_out = processor0.F_A_B;  
end
```

```
initial begin: INTERNAL_FORCES  
    ...  
    #1    force processor0.F_A_B = 1'b0;
```