ECE 385 – Digital Systems Laboratory

Lecture 10 – SLC-3 Microprocessor cont.

Zuofu Cheng, Deming Chen

Spring 2016

Link to Course Website

ECE ILLINOIS

ILLINOIS

Lab 6: Goals (Week 1)

- Create SLC3 (Simplified LC3) microprocessor in SystemVerilog
  - 16-Bit Data Path
  - Memory-mapped I/O (only mapped peripheral is HEX displays using Mem2IO)
  - Register File (8 registers with control)
  - Other Registers
    - PC, IR, MAR, MDR, nzp status register
  - ALU and Memory Instructions
    - Add, Sub, Logical Ops, Load, Store
  - Control Flow instructions
    - Branch and Jump Subroutine
- Week 1: Demo only FETCH operation
  - Simulated and real memory
  - May use SV arithmetic operators (e.g. a = a + 1;)
  - Must pass timing and work at 50 MHz

Week 1 Demo

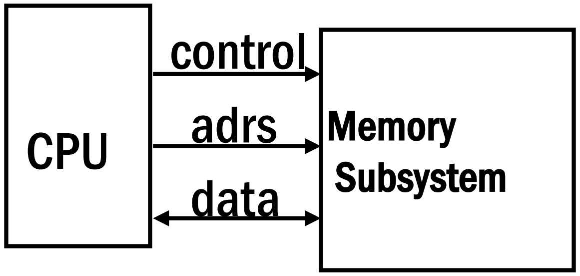
- Simulation of PC loading into MAR and PC incrementing. (1 points)
  - Use test_memory.sv
- Simulation of MDR loading into IR. (1 points)
  - Use test_memory.sv
- Correct FETCH operation on the board, showing IR on the hex displays.
  - Must use the physical memory (test_programs_image.ram) instead of the test memory (test_memory.sv). (3 point)
  - Mem2IO block takes up 4 HEX displays as I/O peripheral, use other 4 for displaying IR
  - Should halt after each FETCH so correct instruction can be seen on display
- **Even though demo is simple, plan on finishing at least data-path this week, or week 2's assignment will be impossible!**
  - Create all of the components in block diagram (register file, other registers, MUXes, ALU, branch logic, sign and zero extension blocks etc…) this week
  - Dedicate next week to control unit state machine (IDSU) and debugging

Available Documentation

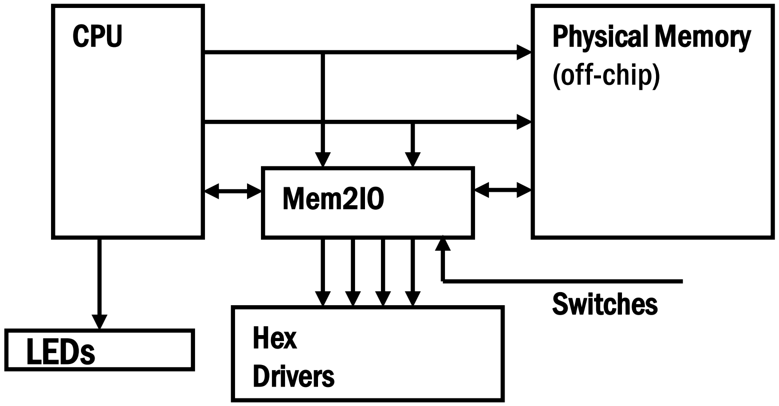
- Lab 6 Materials in Lab Manual
  - ISA breakdown (instruction coding for all 11 instructions)
  - Execution summary (RTL description for FETCH, DECODE, EXEC) for each instruction
  - Simplified block diagram
- Appendix C from P&P
  - Detailed ISA description of LC3
  - Full block diagram (with MUXes & individual registers)
  - Full state diagram
- Appendix A from P&P
  - Detailed programming guide for LC3
  - Explains instruction encoding and has examples for each instruction

Top Level Block Diagram

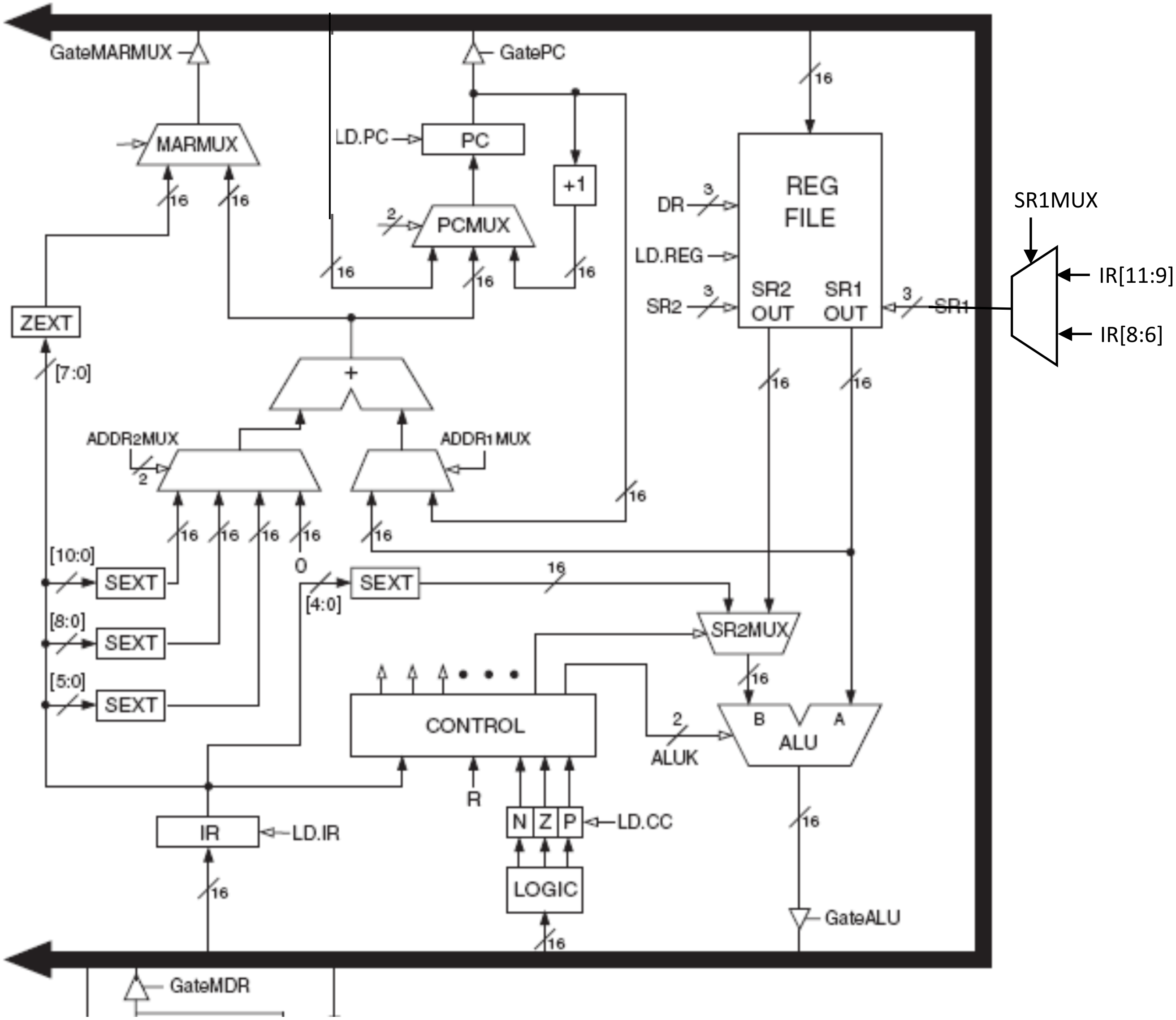
- Block diagram of FPGA top level
  - CPU only has control (r/w), address, data
  - I/O provided by Mem2IO block (memory controller & I/O mapper)
  - Focus on control/addr/data signals



Physical I/O Device	Type	Memory Address	"Memory Contents"
DE2 Board Hex Display	Output	0xFFFF	Hex Display Data
DE2 Board Switches	Input	0xFFFF	Switches (15:0)



SLC-3 Block Diagram



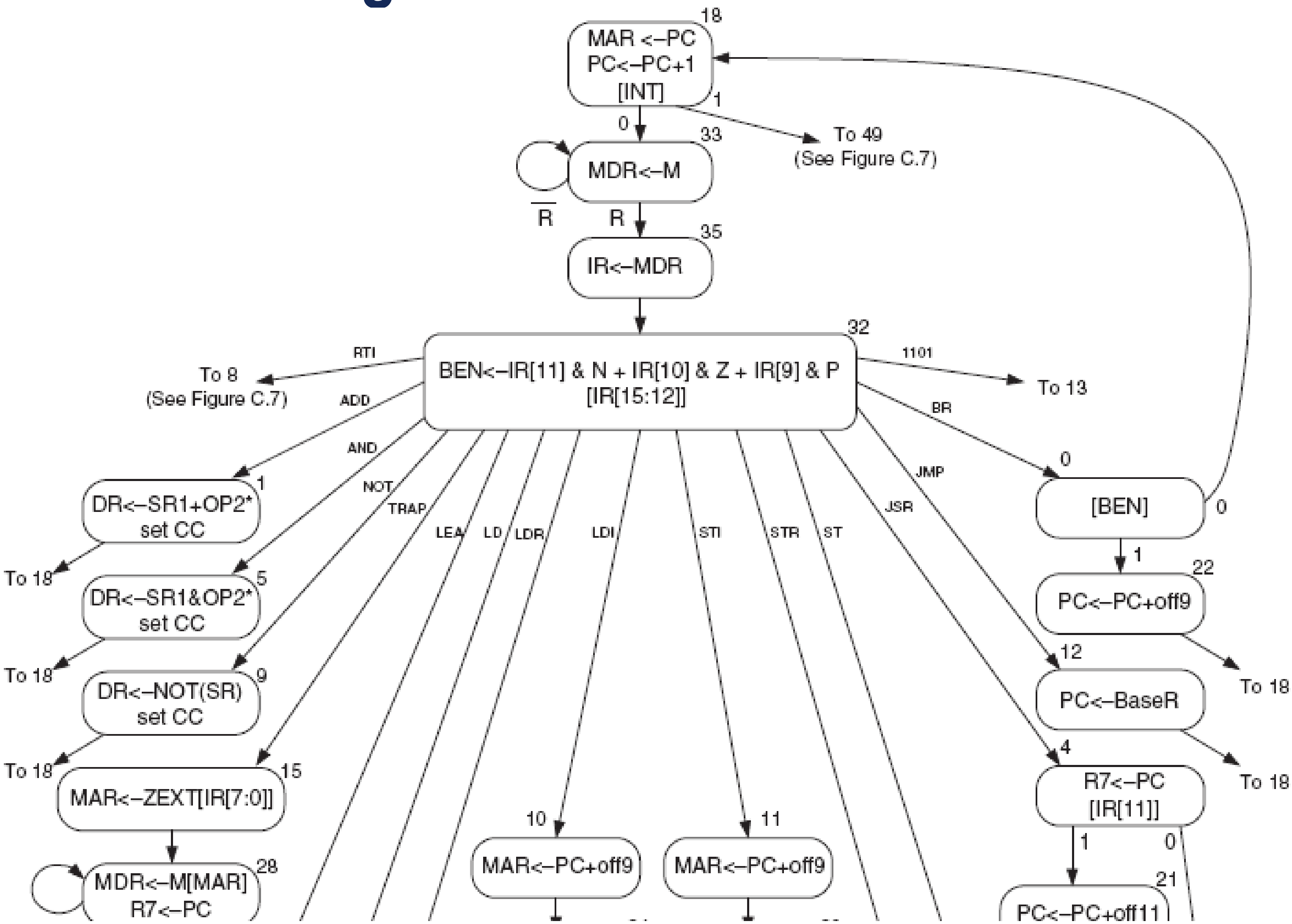
MDR is down here

For complete diagram check out the online materials

SLC-3 ISA – Subset of LC-3 ISA

Instruction	Instruction(15 downto 0)						Operation
ADD	0001	DR	SR1	0	00	SR2	R(DR) ← R(SR1) + R(SR2)
ADDi	0001	DR	SR	1	imm5		R(DR) ← R(SR) + SEXT(imm5)
AND	0101	DR	SR1	0	00	SR2	R(DR) ← R(SR1) AND R(SR2)
ANDi	0101	DR	SR	1	imm5		R(DR) ← R(SR) AND SEXT(imm5)
NOT	1001	DR	SR	111111			R(DR) ← NOT R(SR)
BR	0000	N Z P	PCoffset9				if ((nzp AND NZP) != 0) PC ← PC + 1 + SEXT(PCoffset9)
JMP	1100	000	BaseR	000000			PC ← R(BaseR)
JSR	0100	1	PCoffset11				R(7) ← PC + 1; PC ← PC + 1 + SEXT(PCoffset11)
LDR	0110	DR	BaseR	offset6			R(DR) ← M[R(BaseR) + SEXT(offset6)]
STR	0111	SR	BaseR	offset6			M[R(BaseR) + SEXT(offset6)] ← R(SR)
PAUSE	1101	ledVect12					LEDs ← ledVect12; Wait on Continue

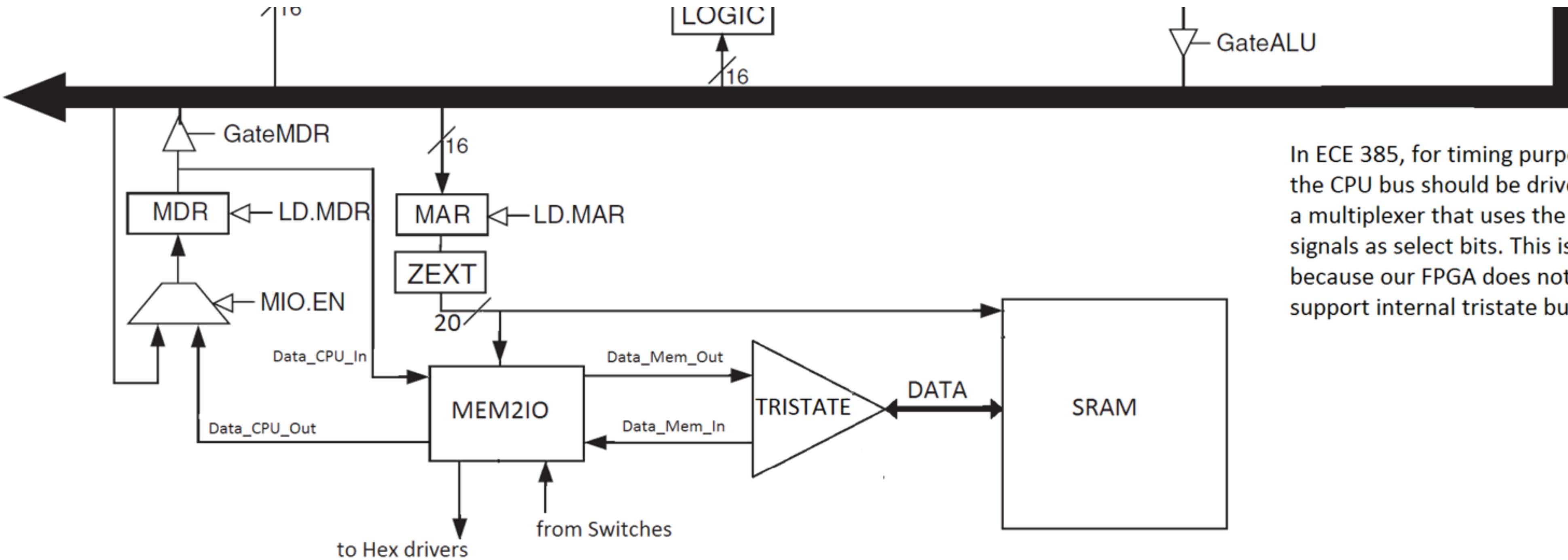
Partial State Diagram



For complete diagram check out the online materials

CPU to SRAM Configuration

- CPU to SRAM with Physical Memory



In ECE 385, for timing purposes, the CPU bus should be driven with a multiplexer that uses the Gate signals as select bits. This is because our FPGA does not support internal tristate buffers.

Instantiating Top Level Mem2IO & Tristate

- Mem2IO & Tristate blocks provided (top-level should have this in addition to ISDU (state machine)
- Note:

```
logic [15:0] MAR, MDR, MDR_In;
logic [15:0] Data_Mem_In, Data_Mem_Out; //bidirectional data from SRAM
assign ADDR = { 4'b00, MAR }; //SRAM is 1Mx16 as opposed to 64Kx16
tristate #(.N(16)) tr0(
    .Clk(Clk), .OE(~WE), .In(Data_Mem_Out), .Out(Data_Mem_In), .Data(Data)
);

Mem2IO memory_subsystem(
    .*, .Reset(Reset_ah), .A(ADDR), .Switches(S),
    .HEX0(hex_4[0]), .HEX1(hex_4[1]), .HEX2(hex_4[2]), .HEX3(hex_4[3]),
    .Data_CPU_In(MDR), .Data_CPU_Out(MDR_In)
);
```

Understanding Tristate Buffers

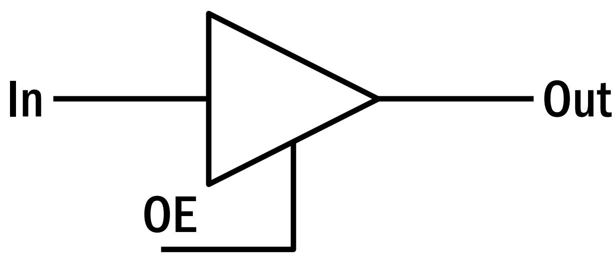
- Both external SRAM interface and CPU block diagram has several tristate buffers
- Tristate.sv as provided only to be used for SRAM

```
module tristate #(N = 16) (
    input wire Clk, OE,
    input [N-1:0] In,
    output logic [N-1:0] Out,
    inout wire [N-1:0] Data
);
logic [N-1:0] a, b;

assign Data = OE ? a : {N{1'bZ}};
assign Out = b;

always_ff @(posedge Clk)
begin
    b <= Data;
    a <= In;
end

endmodule
```

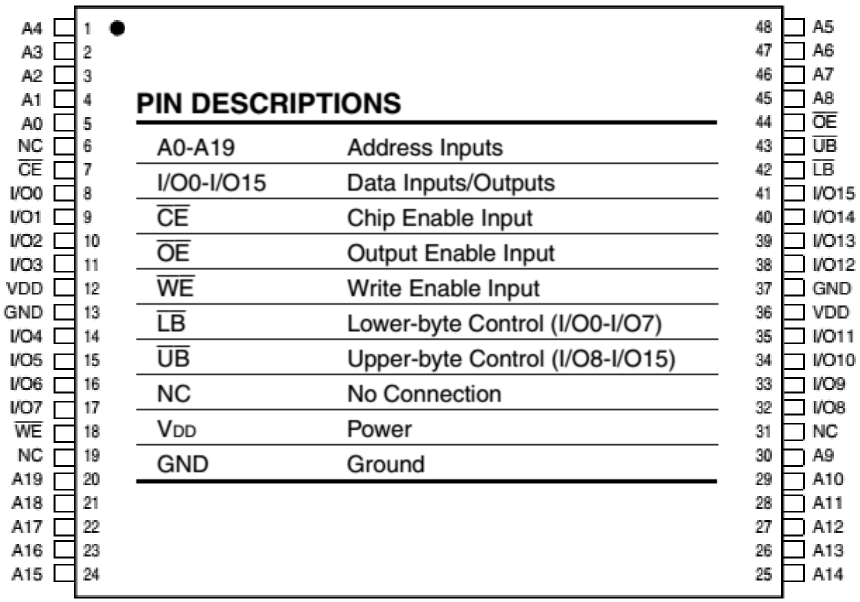


Internal Tristate Buffers

- Modern FPGAs do not have internal tristate buffers
  - Floating signals inside a chip are bad for power and reliability reasons
  - Synthesis tools are aware of this and will replace tristate buffers with other elements
- Block diagram implies tristate buffer for driving the data bus
  - Option 1: use a MUX which decides which component drives the bus (recommended)
  - Option 2: use tristate buffers anyway and have synthesis tools automatically replace it with other elements
    - On Cyclone IV it **should** replace with a MUX
    - On other devices, it may route out to IOB and route back (highly inefficient)

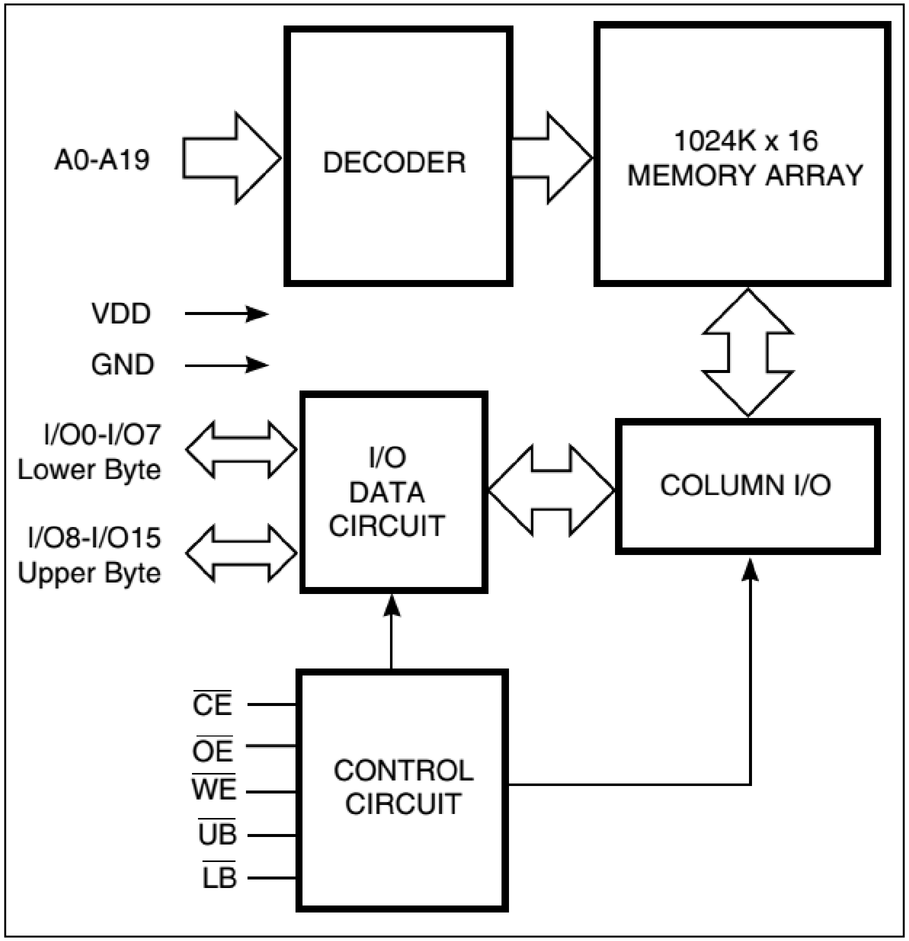
External SRAM

- 1M x 16 (2 Mbyte) organization
- Asynchronous (Access time = 10ns)
- 16 bit organization
- Byte access via UB/LB
- [Datasheet here]



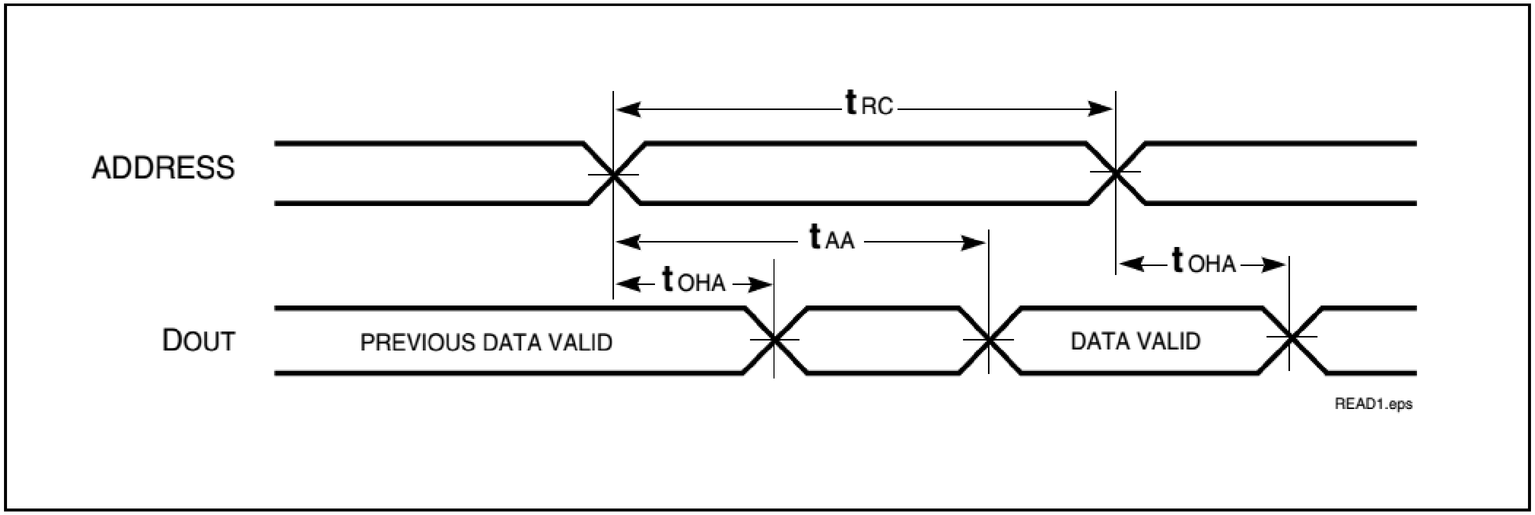
PIN DESCRIPTIONS

A0-A19	Address Inputs
I/O0-I/O15	Data Inputs/Outputs
$\overline{CE}$	Chip Enable Input
$\overline{OE}$	Output Enable Input
$\overline{WE}$	Write Enable Input
$\overline{LB}$	Lower-byte Control (I/O0-I/O7)
$\overline{UB}$	Upper-byte Control (I/O8-I/O15)
NC	No Connection
V_{DD}	Power
GND	Ground



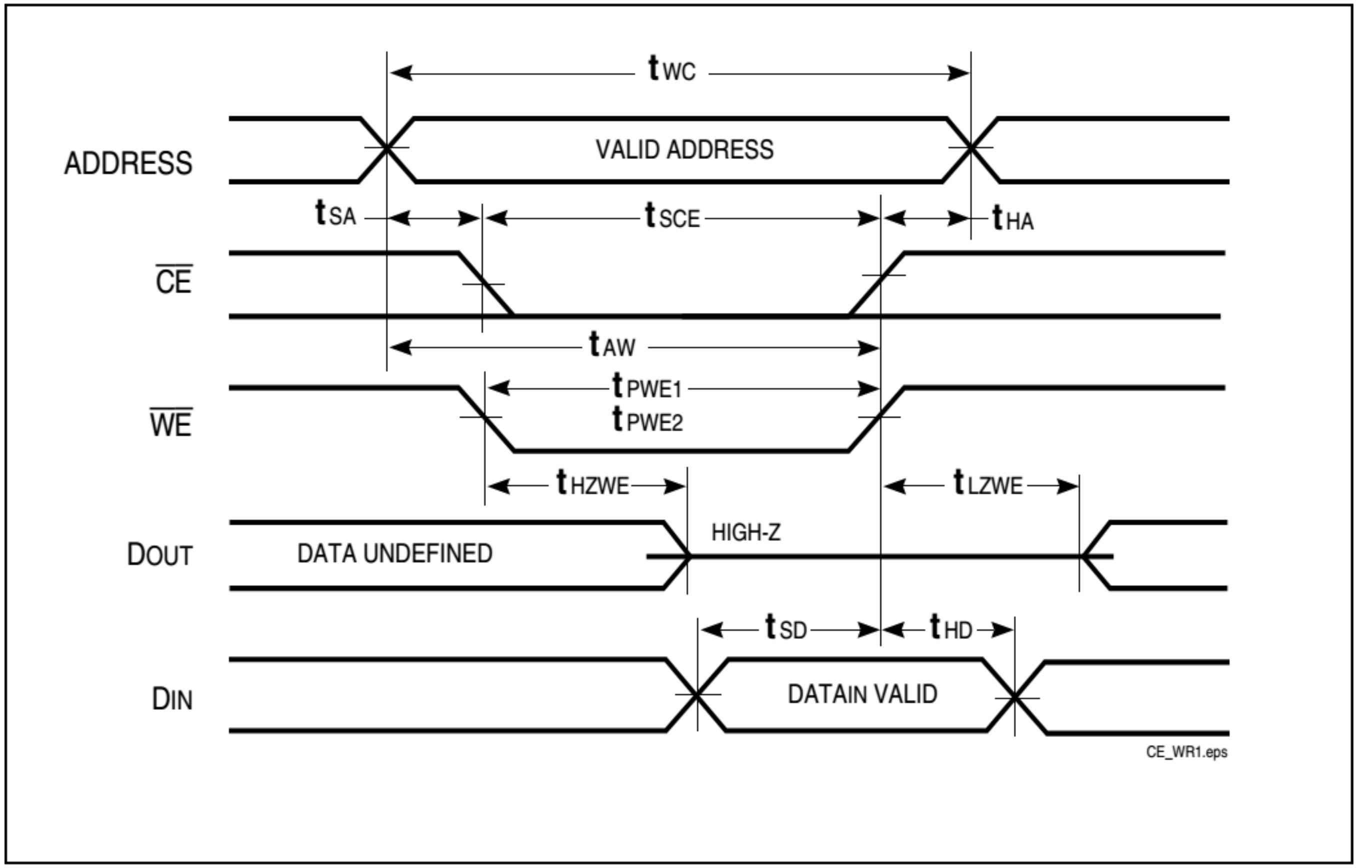
External SRAM Timing (Read)

- External SRAM is **asynchronous**
- !CE = !OE = 0 (in diagram below)
- Data is valid 10ns after address is valid
- If CPU (and state machine) running at 50 Mhz, data guaranteed to be valid by next cycle (from address being valid)
- Tristate has internal flip flop for synchronization, so wait a total of 2 cycles in R



External SRAM Timing (Write)

- OE and WE have to be driven from your state machine
- OE and WE drive asynchronous SRAM, so they need to be synchronized!



Synchronization of Asynchronous Signals

- Our memory is asynchronous (as are pushbuttons, switches, etc)
- Why do inputs need to be synchronized?
- How do we synchronize inputs?

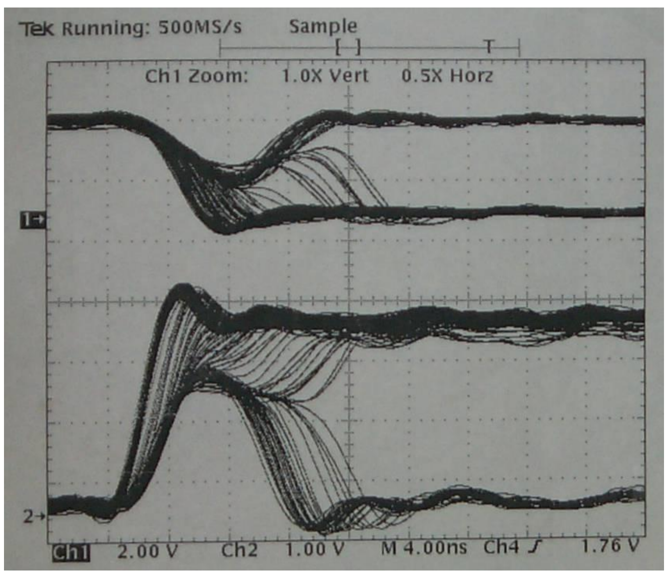
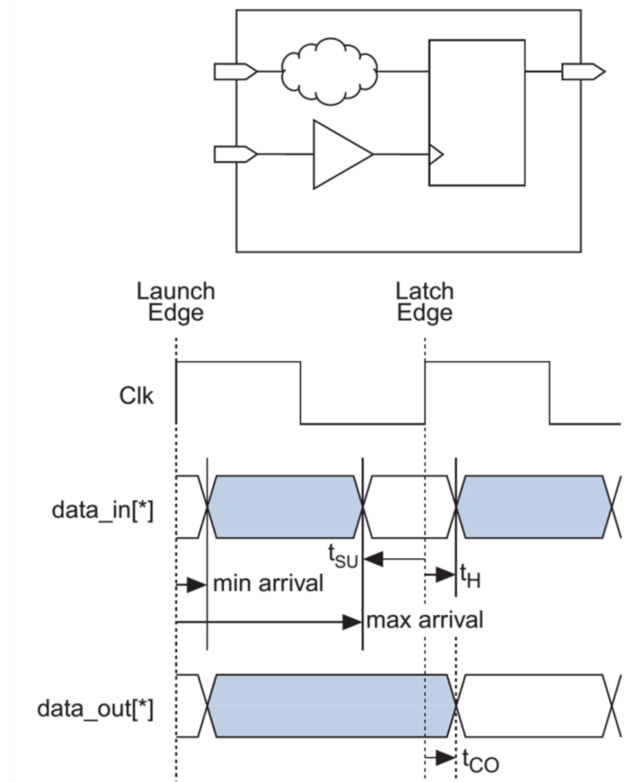


Image courtesy of Philip Freidin - EETimes



Synchronization of Asynchronous Signals

- What about outputs?
- Does the FPGA always generate glitch free logic?
- How do we synchronize?

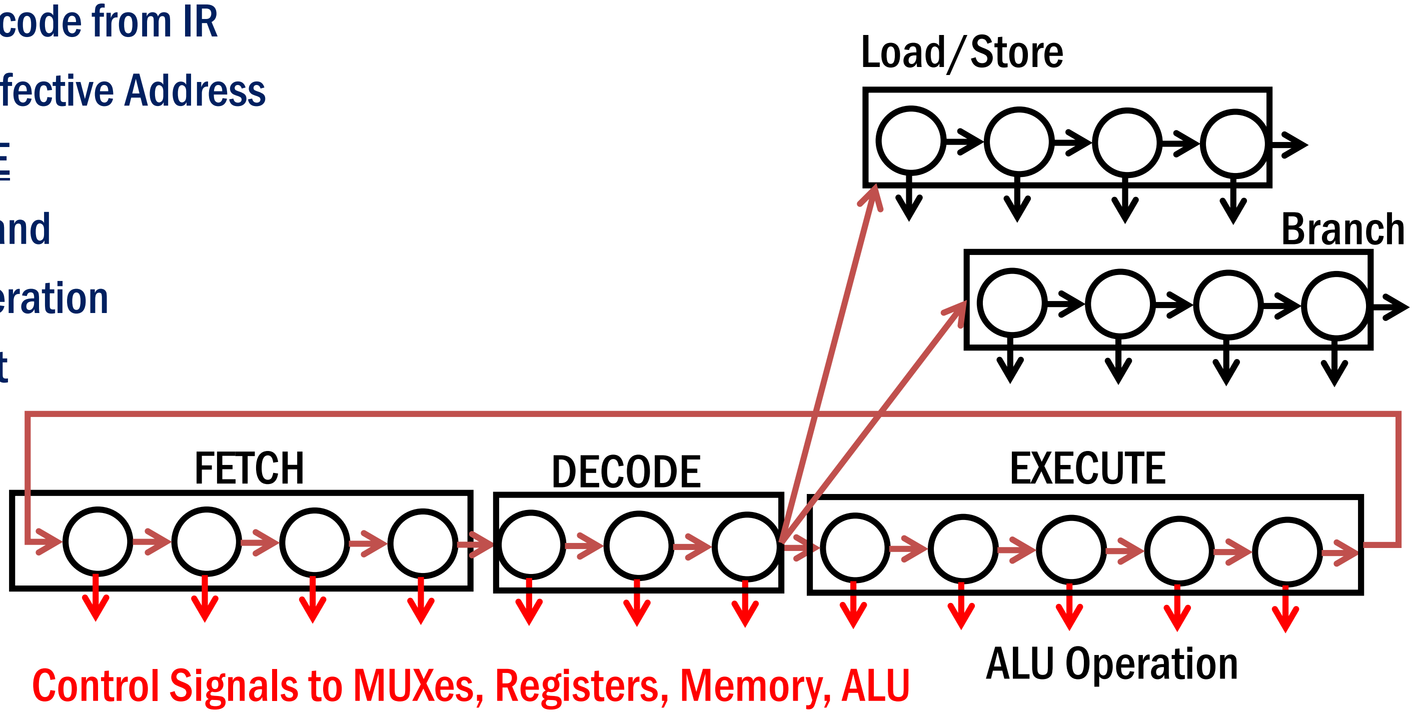
Debugging Memory Issues

- Always start with simulation memory (test_memory.sv)
  - This is **not** how the provided code is set up, you will need to figure this out
  - test_memory.sv emulates SRAM chip reasonably well, but is not 100% accurate
- If you have physical memory issues (but not with simulated memory)
  - Make sure you are accounting for 1 cycle delay in tristate.sv
  - Make sure you synchronize inputs and outputs from physical memory (input already is synchronized through tristate.sv, need to synchronize OE, WE, etc)
  - Make sure you are meeting access time for memory read
  - Try using SignalTap built in logic analyzer
    - Check out "Using SignalTap II with Verilog Designs" documents
    - Will be required to used for Lab 8, but can be helpful earlier than that

Instruction Cycle

- Think of instruction cycle in three main phases
  - <u>Phase1: FETCH</u>
    - MAR <- PC;
    - IR <- Read Memory;
    - PC <- PC+1;
  - <u>Phase2: DECODE</u>
    - Decode op-code from IR
    - Compute Effective Address
  - <u>Phase3: EXECUTE</u>
    - Fetch Operand
    - Execute operation
    - Store Result

Note: Cycle counts in diagram not accurate



Understanding the Instruction Cycle

- My advice:
  - Print out ~10 copies of SLC block diagram
  - With different colored highlighter, trace out direction each MUX goes for each clock cycle of in FETCH, DECODE and EXECUTE
  - FETCH should be common for all instructions
  - DECODE & EXECUTE will be different depending on instruction, start with a blank copy of block diagram the DECODE & EXECUTE cycles for each instruction

FETCH Phase

- state1: MAR ← PC
- state2: MDR ← M(MAR); -- *assert Read Command on the RAM*
- state3: IR ← MDR;

  PC ← PC+1; -- "+1" inserts an incrementer/counter

  instead of an adder.

  Go to decode state – or halt (in the case of week 1)

<u>More details:</u>

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

Provided IDSU Template

```
unique case (State)
…
S_33_2 : Next_state = S_35;          // Second cycle of mem FETCH (needed for SRAM)
S_35 :   Next_state = PauseIR1;      // Only for Week1
                                     // Bypass PauseIR in Week 2:
                                    // Next_state <= S_32;
PauseIR1 :            // Pause to display IR on HEX. (Week 1)
        if (~ContinueIR) Next_state = PauseIR1;
        else             Next_state = PauseIR2;
PauseIR2 :            // Wait for ContinueIR to be released. (Week 1)
         if (ContinueIR) Next_state = PauseIR2;
         else            Next_state = S_32 S_18; // Loop FETCH for Week 1

S_32 :
         case (Opcode) …


…
```

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
```