

ECE 385 – Digital Systems Laboratory

Lecture 10 – Lab 6: SLC-3 Microprocessor
Zuofu Cheng

Spring 2019

[Link to Course Website](#)



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

SystemVerilog Parameters

- Parameterized modules are useful to reduce code duplication
- Consider: 2-1 MUX – CPU may require widths of 2-1 MUX (2-bit, 8-bit, 16-bit)
- Can make separate modules MUX2to1_16, MUX2to1_8, but lots of redundant code (code looks exactly same)

```
module mux2
    #(parameter width = 8)
    (input logic [width-1:0] d0, d1,
     input logic s,
     output logic [width-1:0] y);

    always_comb begin
        if (s)
            y = d1;
        else
            y = d0;
        end
    endmodule

//default with 8-bit width
mux2 my_8_mux (.d0(bus8_a), .d1(bus8_b), ...)

//16-bit mux instead of 8-bit (note first method doesn't work in SV)
mux2 #(width = 16) my_16_mux (.d0(bus16_a), .d1(bus16_b), ...)
mux2 #(16) my_16_mux (.d0(bus16_a), .d1(bus16_b), ...)
```

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;
```

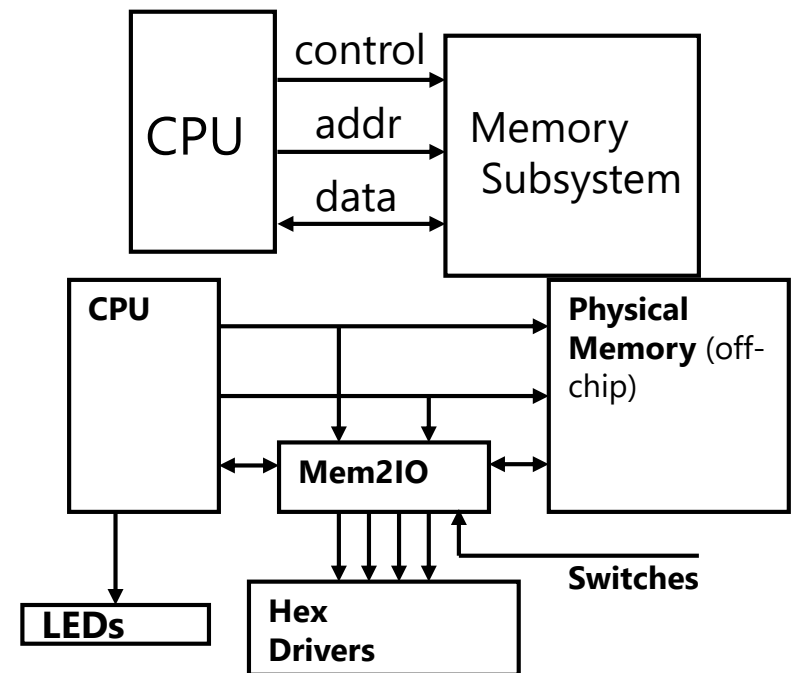
Available Documentation

- 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

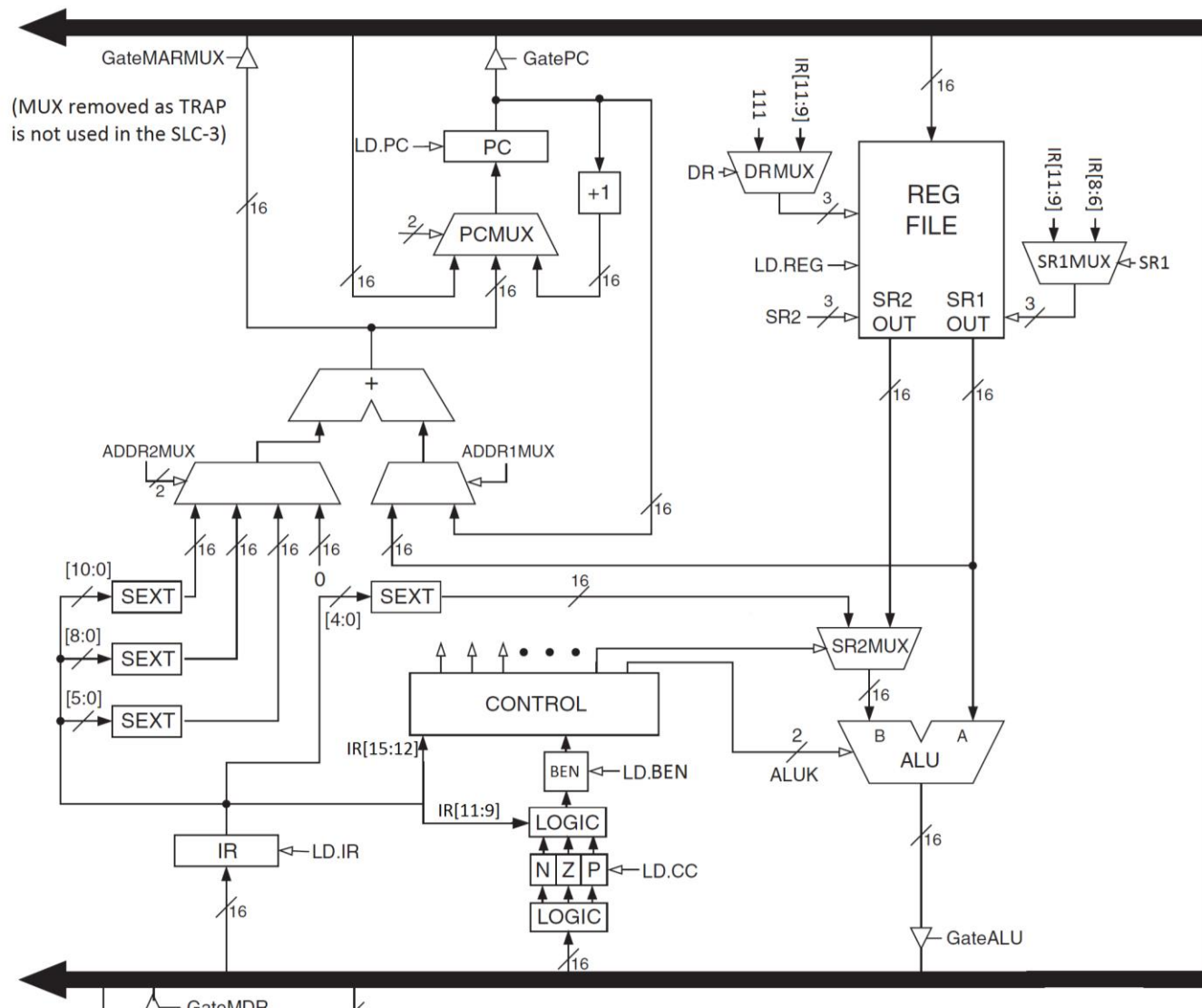
Top Level Block Diagram

- Block diagram of FPGA top level
 - CPU only has control (r/w), address, data
 - I/O provided by Mem2IO block (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



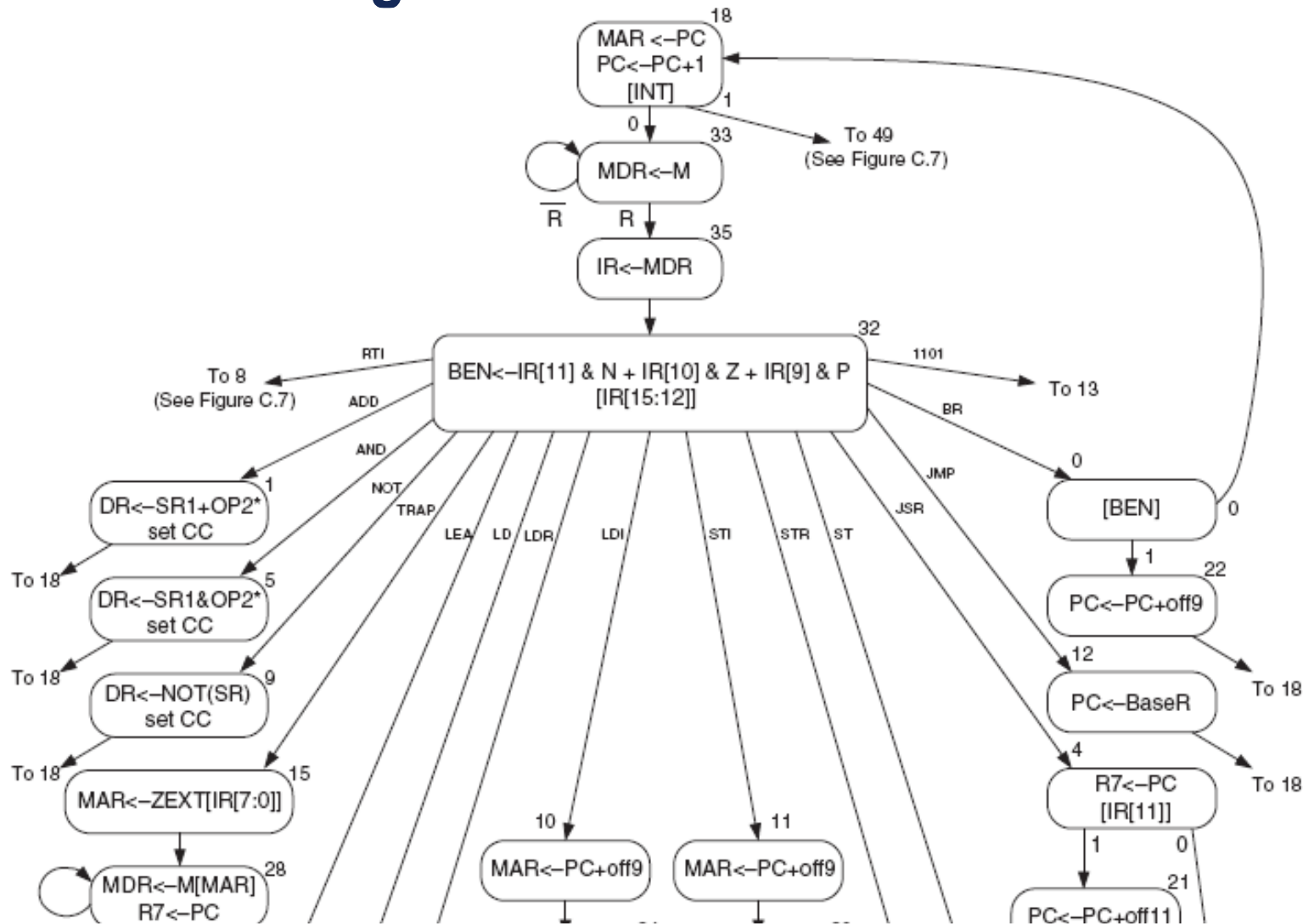
MDR is down here

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					$R(DR) \leftarrow R(SR1) + R(SR2)$
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									$\text{LEDs} \leftarrow \text{ledVect12}; \text{ Wait on Continue}$

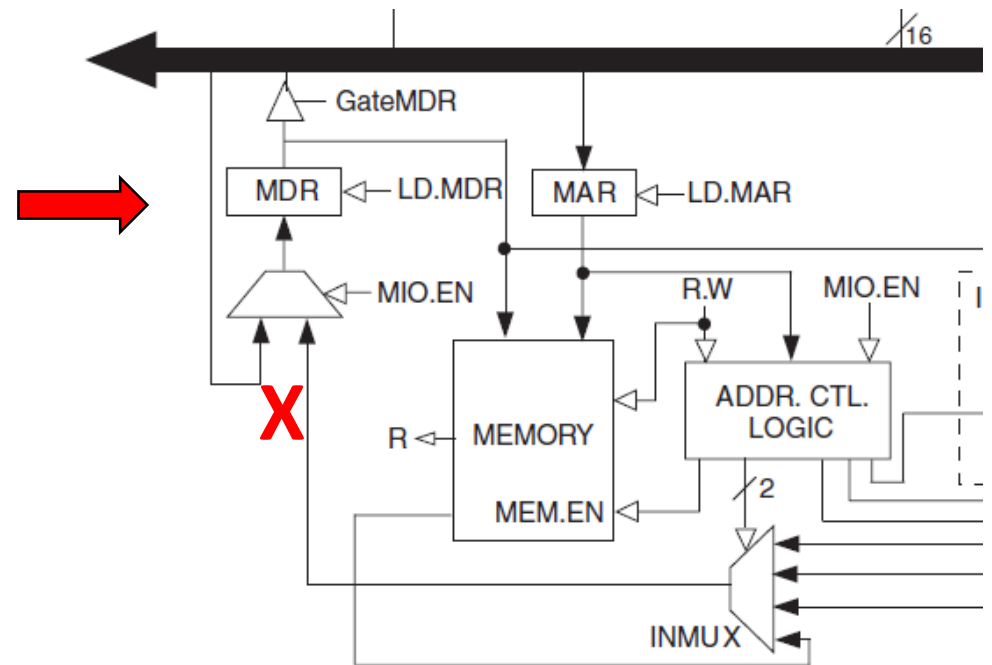
Partial State Diagram



For complete diagram check out the online materials

CPU to Mem2IO/SRAM Connection

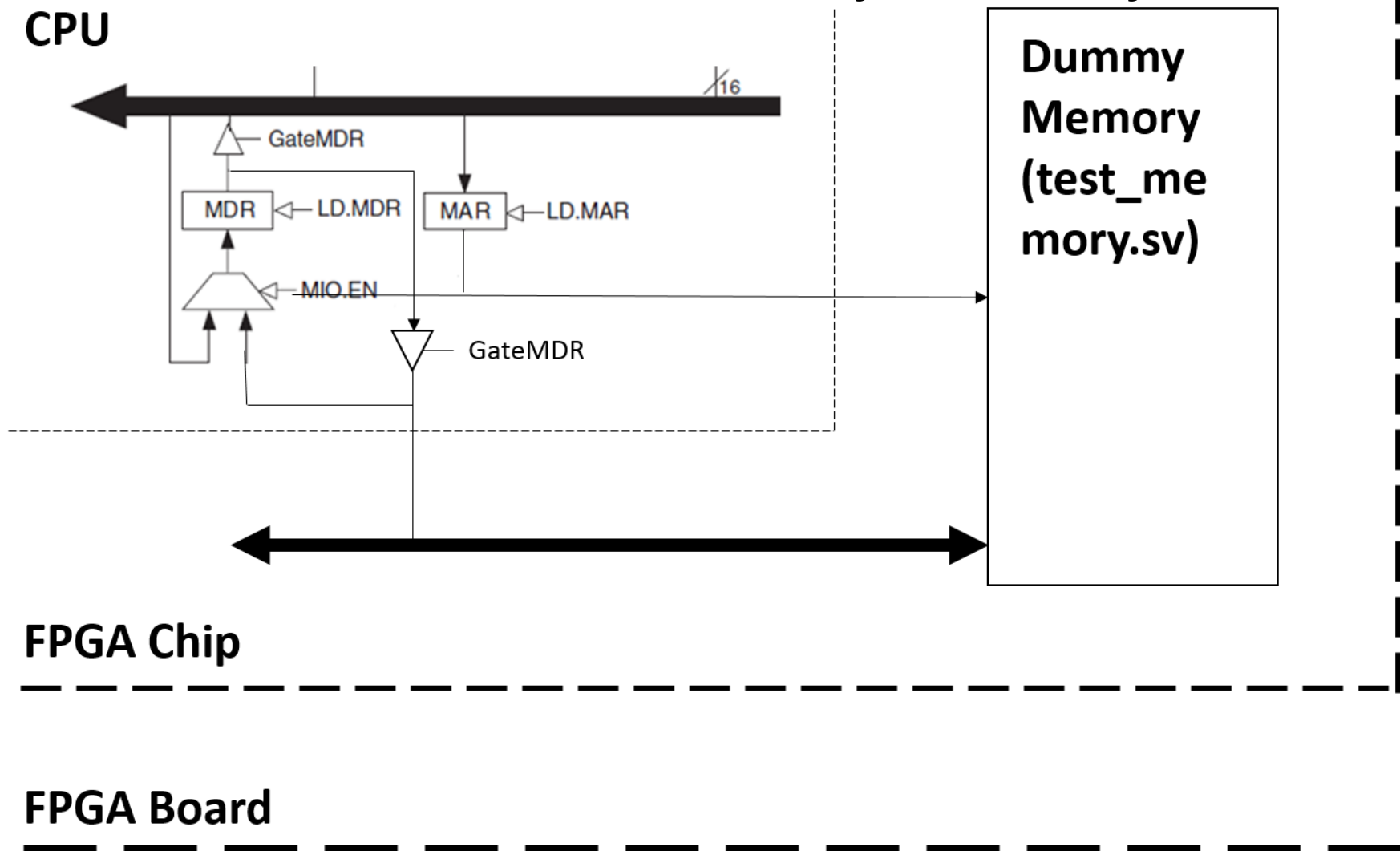
- **MEM2IO uses inout ports**
 - Appendix C has MDR directly connected to the memory.
 - **THIS IS NOT THE WAY**
 - We have to go through the tristate block, which converts uni-directional signals (internal to FPGA) to bidirectional signals



- MDR/MAR -> MEM2IO -> TRISTATE -> SRAM

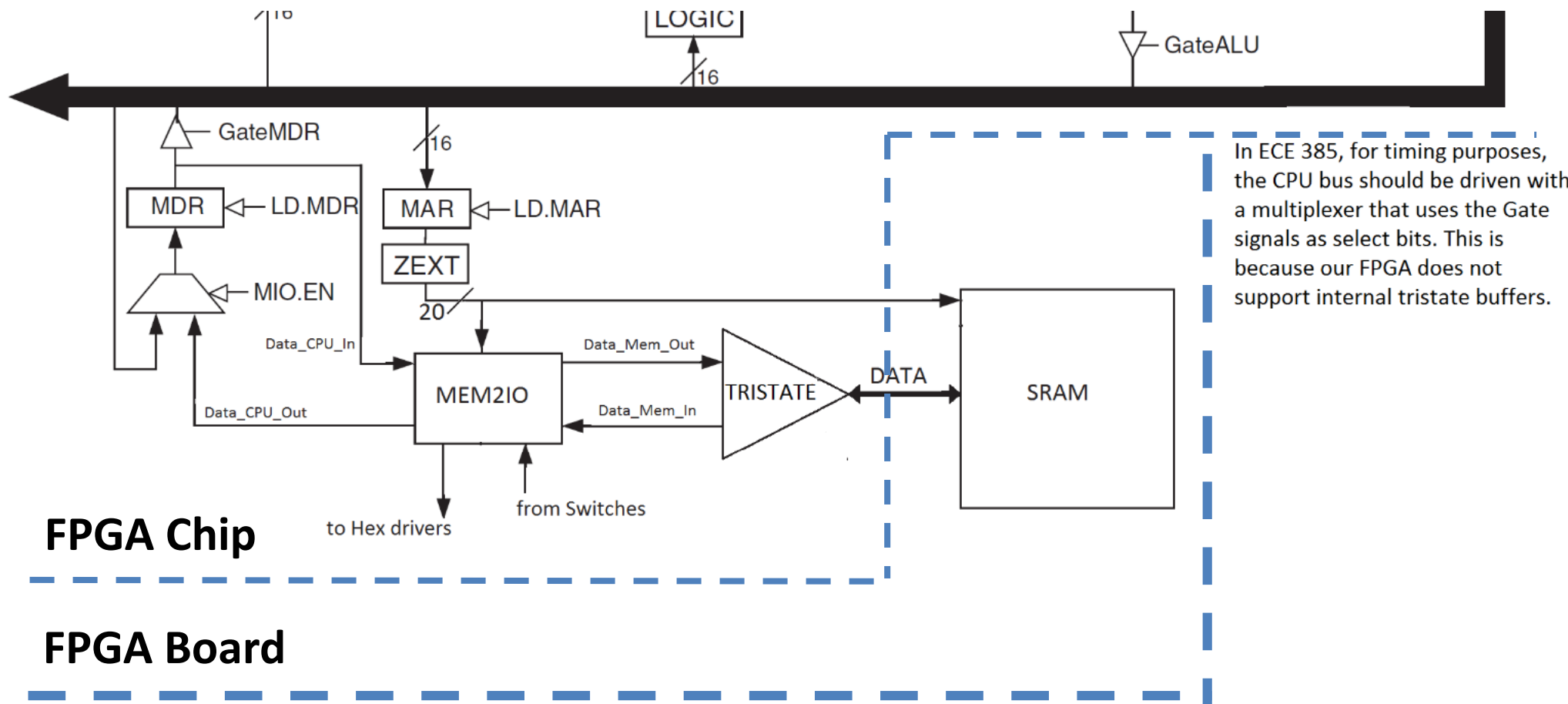
CPU to Dummy Memory Configuration

- CPU to SRAM with Dummy Memory



CPU to SRAM Configuration

- CPU to SRAM with Physical Memory



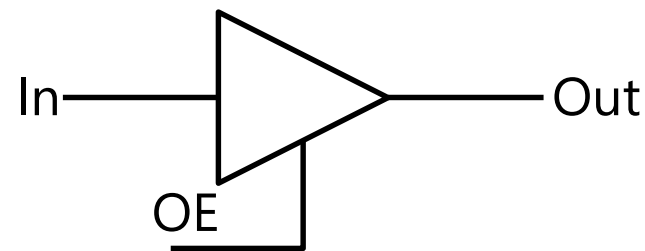
Instantiating Top Level Mem2IO & Tristate

- Mem2IO & Tristate blocks provided (top-level should have this in addition to ISDU - state machine)
- SRAM needs additional signals for operation (WE, OE, etc)
 - These signals should be driven by the ISDU in the appropriate states (when we need to read or write from SRAM)
- Remember:
 - MDR/MAR -> MEM2IO -> TRISTATE -> SRAM

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

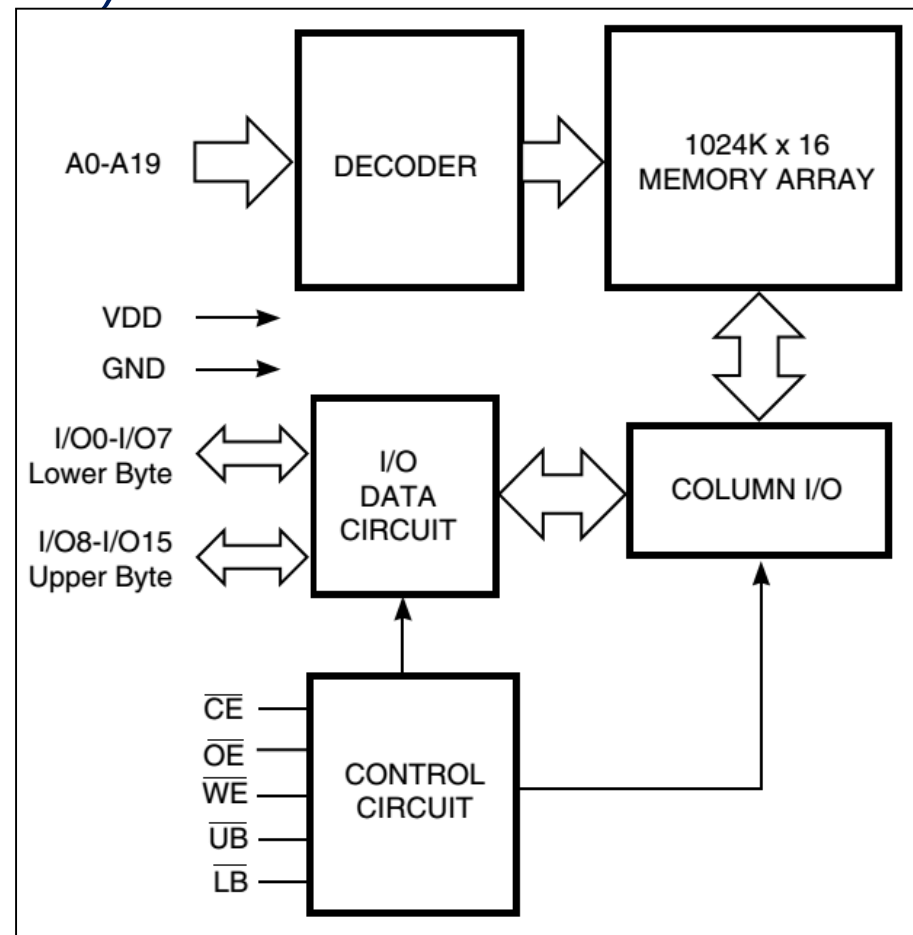
- Modern FPGAs do not have internal tristate buffers
 - Floating signals inside a chip are bad for power and reliability reasons
 - Synthesis tools are aware of this and will replace tristate buffers with other elements
- Block diagram implies tristate buffer for driving the data bus
 - Option 1: use 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](#)

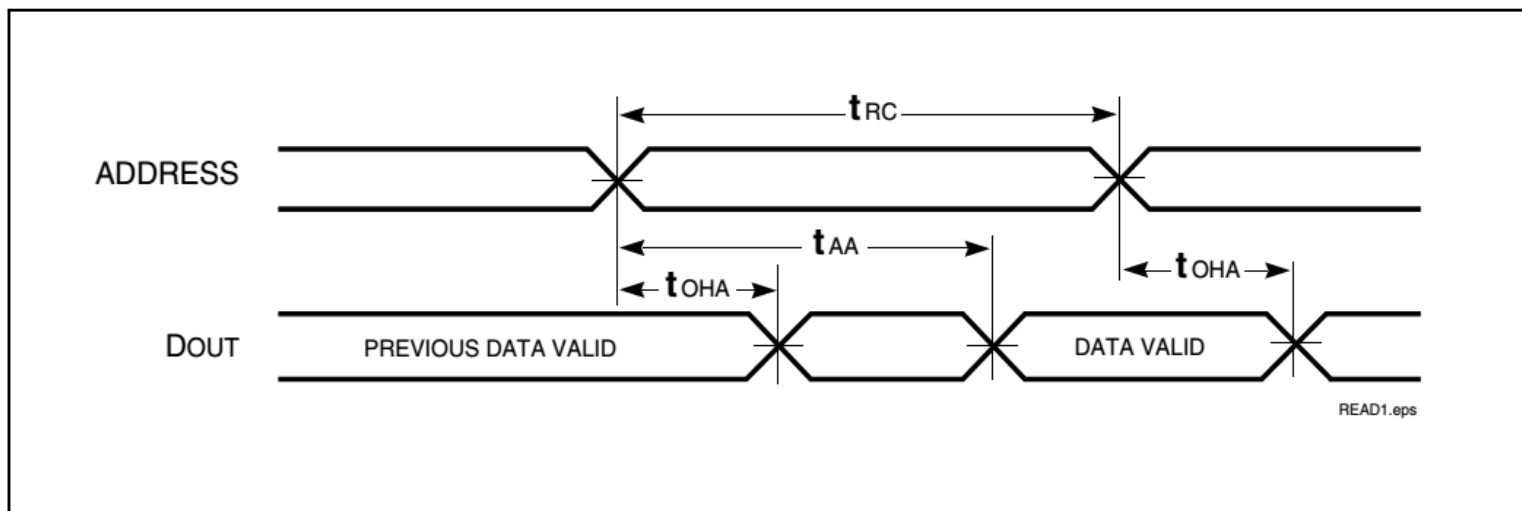
A4	1		48	A5
A3	2		47	A6
A2	3		46	A7
A1	4		45	A8
A0	5		44	OE
NC	6		43	UB
CE	7		42	LB
I/O0	8		41	I/O15
I/O1	9		40	I/O14
I/O2	10		39	I/O13
I/O3	11		38	I/O12
VDD	12		37	GND
GND	13		36	VDD
I/O4	14		35	I/O11
I/O5	15		34	I/O10
I/O6	16		33	I/O9
I/O7	17		32	I/O8
WE	18		31	NC
NC	19		30	A9
A19	20		29	A10
A18	21		28	A11
A17	22		27	A12
A16	23		26	A13
A15	24		25	A14

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	
VDD	Power	
GND	Ground	



External SRAM Timing (Read)

- External SRAM is **asynchronous**
- $\text{!CE} = \text{!OE} = 0$ (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



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?

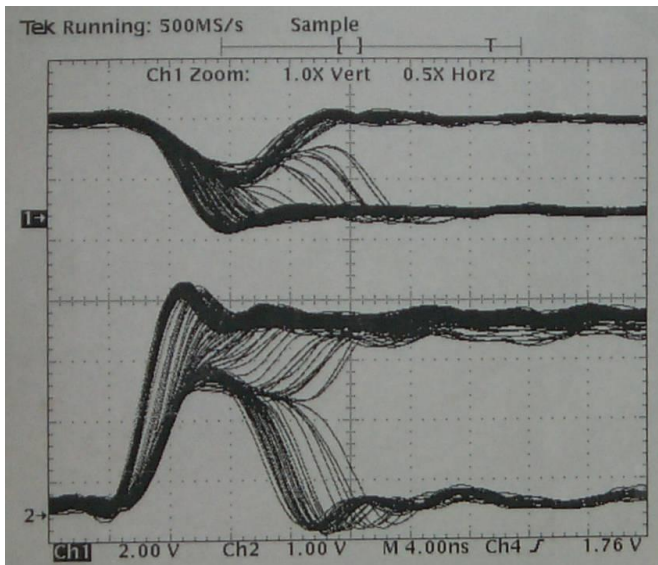
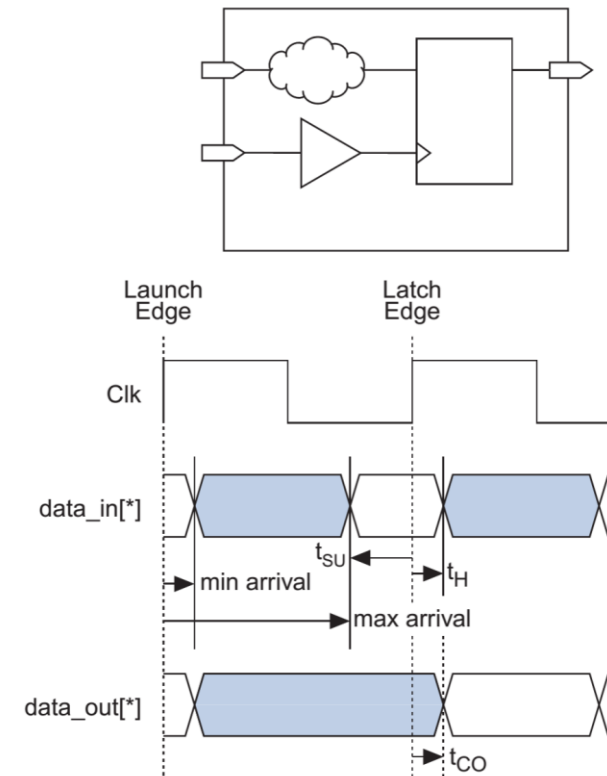


Image courtesy of Philip Freidin - EETimes



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

- Phase1: FETCH

- $MAR \leftarrow PC;$
- $IR \leftarrow \text{Read Memory};$
- $PC \leftarrow PC + 1;$

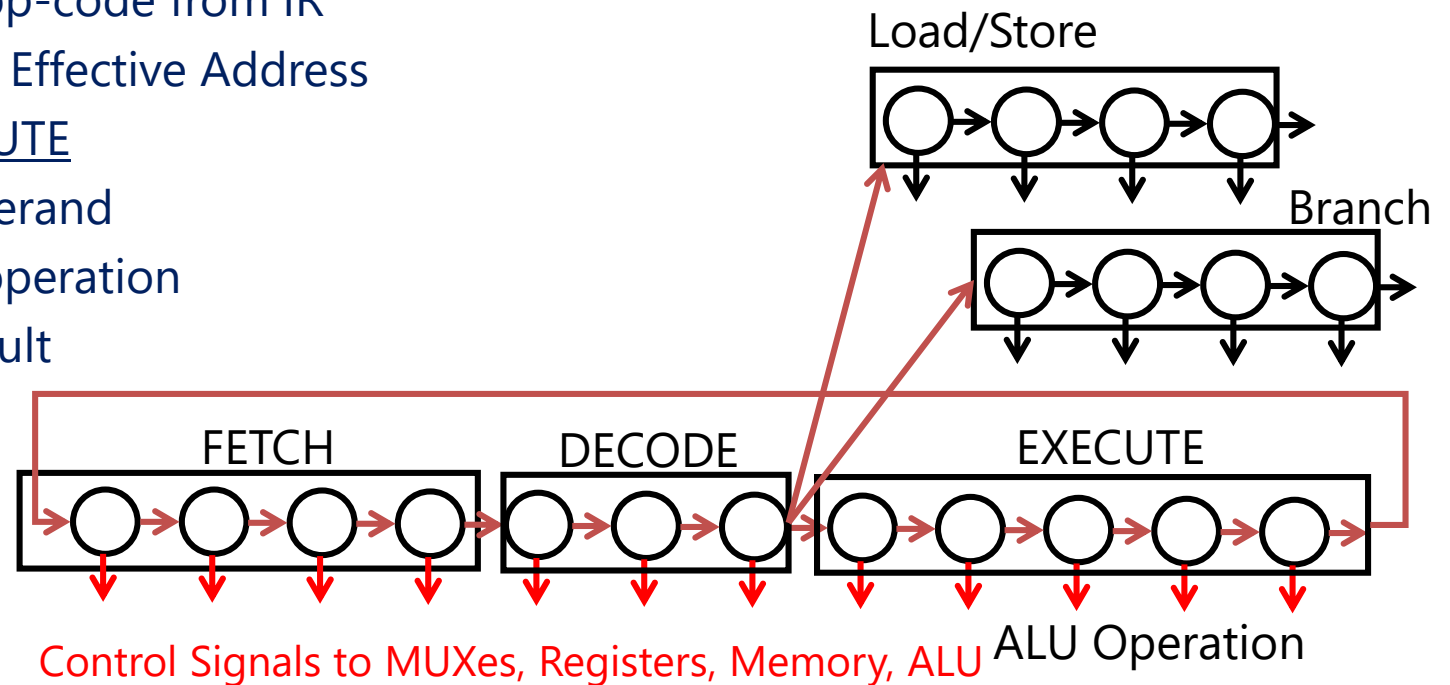
- Phase2: DECODE

- Decode op-code from IR
- Compute Effective Address

- Phase3: EXECUTE

- Fetch Operand
- Execute operation
- Store Result

Note: Cycle counts in diagram not accurate



Understanding the Instruction Cycle

■ Advice:

- Print out ~10 copies of SLC block diagram
- With different colored highlighter, trace out direction each MUX goes for each clock cycle 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 and decide the DECODE & EXECUTE cycles for each instruction

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

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
- $PC \leftarrow (PC + 1)$

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) ...

...
```