ECE385 Experiment #6

Eric Meyers, Ryan Helsdingen

Section ABG; TAs: Ben Delay, Shuo Liu

March 9th, 2016

emeyer7, helsdin2

I. Introduction

The purpose of this lab is to create a very primitive processing unit designed around the Little Computer 3 (LC3) that was explored during previous ECE curriculum. This will be referred to the SLC3 Processing Unit throughout this lab. The SLC3 is a condensed version of the LC3 that allows user interfacing through memory-mapped I/O on board the Altera Cyclone IV SRAM Module, along with switches and LED indicators to show the user the status of the data within registers of the SLC3.

II. Description of Circuit

The circuit consists of several modules specifically the high level SLC3 module, the register file, the datapath, the Instruction Decoder/Sequencer Unit (IDSU), the Arithmetic and Logic Unit (ALU), many 16-bit registers (MAR, MDR, IR, and PC), several multiplexers, and some tristate buffers. The datapath and ISDU (Control) are shown in Figure 2. The memory interface is shown directly below in the same section in Figure 3.

All of these modules work together with one another to form the top level SLC3. The SLC3 will have the ability to perform up to 11 instructions with operations on 8 16-bit registers (R0-R7).

III. Purpose of Modules

As stated in the previous section there are many modules that work together in this system to form the top level SLC3. The following modules were created:

16-bit Shift Register

These modules have 16 inputs and 16 outputs with a load_enable line that determines if the data-in is sent to the data-out. It was used for the following units:

- Instruction Register (IR)
- Memory Address Register (MAR)
- Memory Data Register (MAR)
- Program Counter (PC)
- Register File (Reg_File)

Multiplexers

A 16-bit 2-to-1 MUX will be used for the ADDR1MUX and the SR2MUX. A 3-bit 2-to-1 MUX will be used for the DRMUX and the SR1MUX.

A 16-bit 4-to-1 MUX will be used for the ADDR2MUX only.

A 16-bit 3-to-1 MUX will be used for the PCMUX. This will take inputs of the databus, the 16-bit adder output, and the PC+1.

Instruction Sequencer and Decoder

Please refer to "Section V: Instruction Sequencer and Decoder" in this document to get more details regarding the use of this module.

Datapath

TODO

Arthmetic and Logic Unit

ALUK	OUTPUT FUNCTION
00	A ADD B
01	A AND B
10	NOT A
11	PASS A

The ALU module does the brunt arithmetic work of the SLC3 machine. The ALU takes in two 16 values

(A and B) and does one of four things with the data: bitwise ADD, bitwise AND, bitwise NOT, or passes the A input through to the output terminal.

NZP

The NZP Module is meant for branching and conditional statements in the SLC3 ISA. The module takes inputs as the 3 NZP bits from the IR (bits 11-9) along with the 16-bit databus. The module outputs a single BEN bit that is then fed into the ISDU to determine if the "BR" instruction must be executed.

This module is only used in the "BR" instruction (state 0), and is updated in the "AND" (state 1), "ADD" (state 5), "NOT" (state 9), and "LDR" (state 27).

SEXT/ZEXT

The SEXT/ZEXT modules take in data that is M bits long and extend it to N bits long where M¡N by adding bits to the front of the input. ZEXT extends the input with purely zeros added to the front of it, whereas SEXT depends on the sign of the value. If the MSB=1, the input is negative and gets extended with the proper number of 1s. On the contrary, if MSB=0, the input is positive and gets extended with the proper number of zeros.

Register File

TODO

16-bit Adder

TODO

Tristate Buffer

The tristate buffer module is used to protect the bus from multiple signals entering the bus. The tristate buffer takes in gate signals determined by the state of the machine and sends the respective data to the proper databus input.

Tristate

TODO

Mem2IO

TODO

IV. State Diagram

The State Diagram can be found on Figure 1 in "Section XI: Figures".

V. Instruction Sequencer / Decoder

This module will contain the state machine for the entire SLC3. It will have the ability to implement 11 states and cycle through them to perform the desired action. Each instruction has a unique "opcode" associated with it that determines which states it branches to once the instruction is loaded into the IR. Every instruction must perform a "FETCH" to store the current instruction , and only some instructions must perform a "LOAD" or "STORE" sequence.

Essentially the "FETCH" sequence of instructions must take the PC and load its associated content into the IR. This consists of moving the PC register contents into the MAR, then retrieving the associated content from the SRAM and moving it into the MDR. Once the MDR is loaded, then this data must be moved from the databus into the IR.

The "LOAD" and "STORE" sequences are meant for the "LDR" and "STR" instruction and memory on board the SRAM must be accessed.

Essentially, in each state, proper variables must be updated to ensure the correct operation of the state diagram. For example in the FETCH sequence of instructions, in order to get the PC to move from the PC Register to the MAR, GatePC must be set to 1, LD_MAR must be set to 1, PCMUX must be set to 00, LD_PC must be set to 1. Please refer to the Appendix of this document to view the code to update the proper variables in the ISDU.

VI. Schematic/Block Diagram

RYAN SECTION

The Schematic / Block Diagrams can be found on Figure **??** in "Section XI: Figures".

VII. Pre-Lab Simulation Waveforms

The Pre-Lab Simulation Waveforms can be found on Figure **??** in "Section XI: Figures".

VIII. Design Statistics

ERIC SECTION

IX. Post Lab

1.) What is MEM2IO used for, i.e. what is its main function?

MEM2IO acts as the middle-man between the data going to/from the SRAM and CPU and the I/O of the

system. For this lab, the I/O includes the switches as input and the Hex displays as output.

2.) What is the difference between BR and JMP instructions?

While the BR branch instruction and the JMP jump instruction pushes the PC to a non-sequential location as a result, they are completely different functions in design. The JMP instruction is simple in design and function. It takes the value in BaseR and shoves it into the PC regardless of any other conditions. The BR instruction is conditional. It relies on the current values of n, z, and p. If any of the values match their equivalents in the status register, the branch is enabled and the value of PC is updated with the new address. If none of the values match, the branch instruction is ignored and the next instruction is called.

X. Conclusion

Week 1 was successful for the team as we finished the Fetch command with relative ease. The team had minor problems with understanding the datapath vs. databus, minimizing delay in data movement, and properly implementing test_memory.sv into the design for simulation and testing. Initially too many modules were created building a high enough delay to the point that signals were missing their proper clock cycles.

Week 2 gave the team much more trouble. Each function required some sort of debugging after the first failed attempt. The largest source of errors was improper connections between modules. Unfortunately, not all of the functions were able to be completely debugged in time for the lab section.

Beyond the problems experienced in lab, understanding the creation of the simple SLC3 processor in System Verilog is a valuable tool for a computer engineer moving foward.

XI. Figures

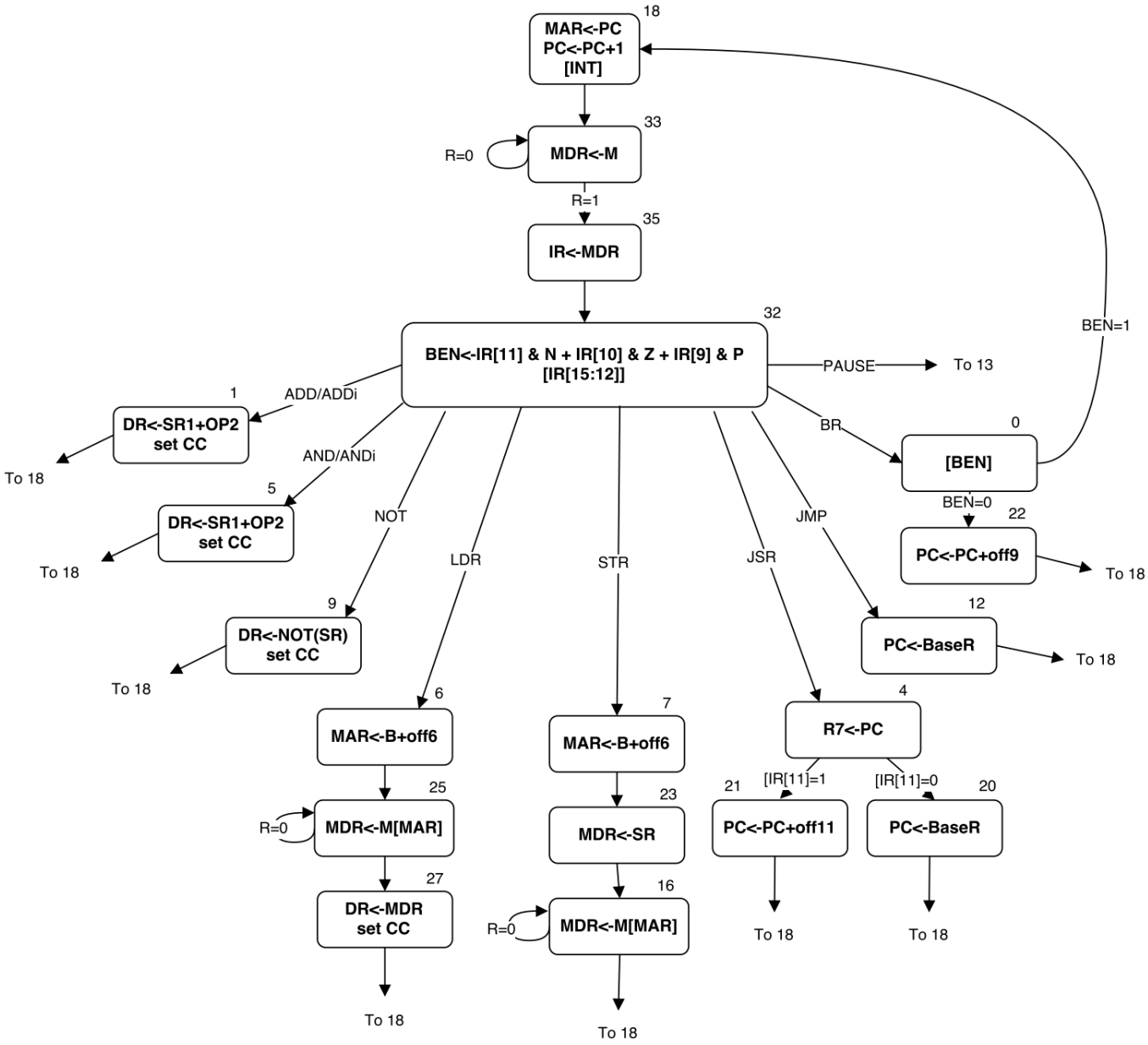

Fig. 1: SLC3 Machine State Diagram

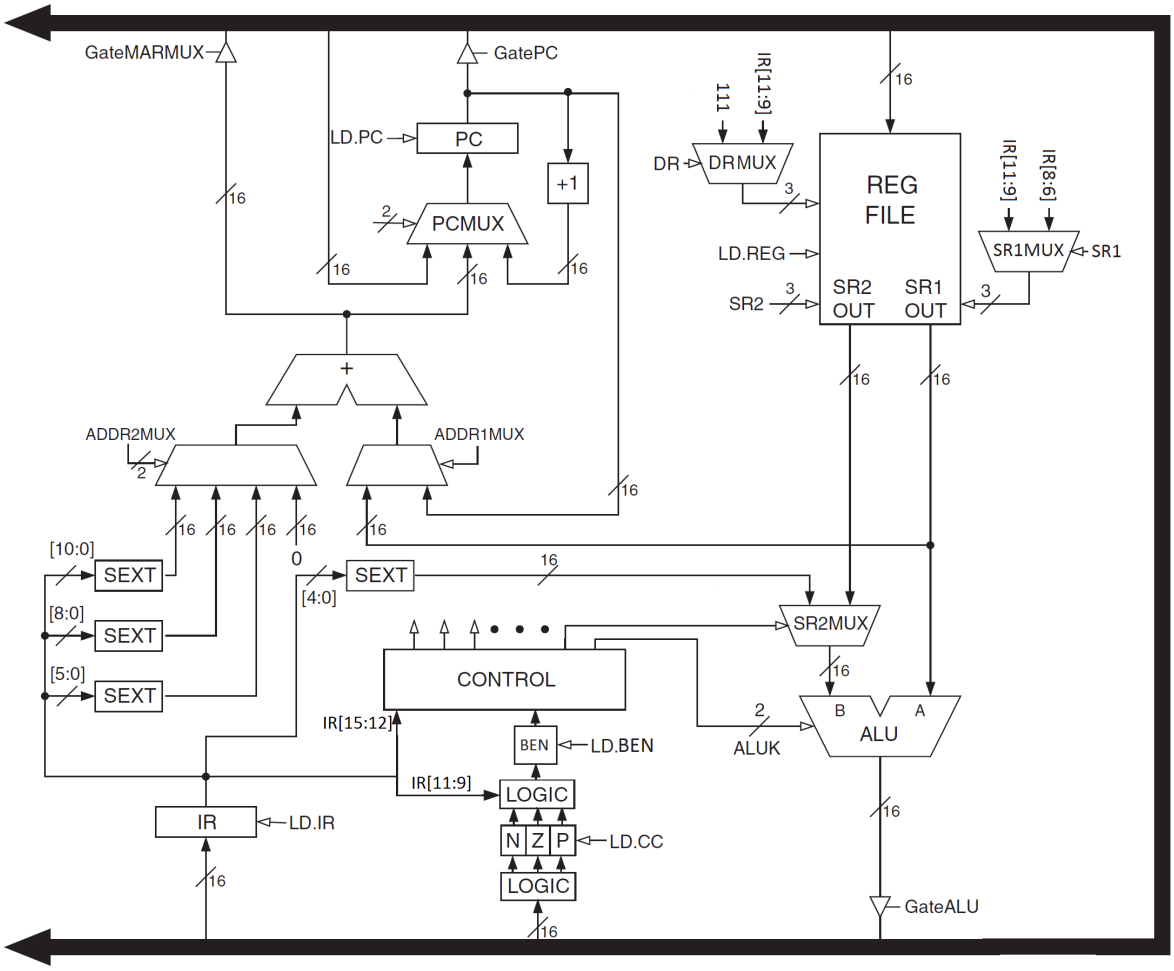

Fig. 2: SLC3 CPU

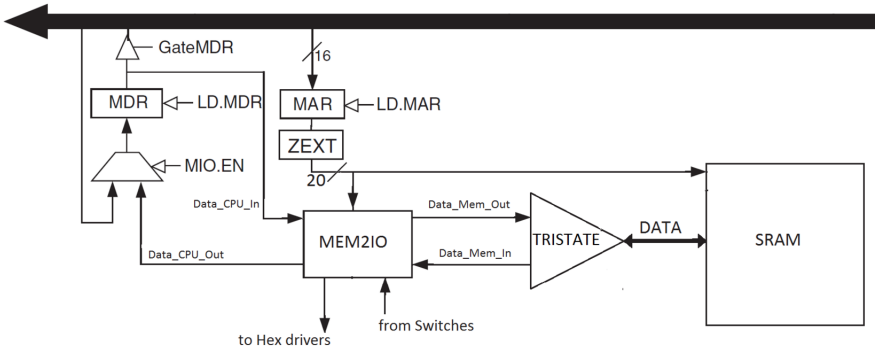

Fig. 3: Memory, MAR, MDR, Mem2IO Configuration

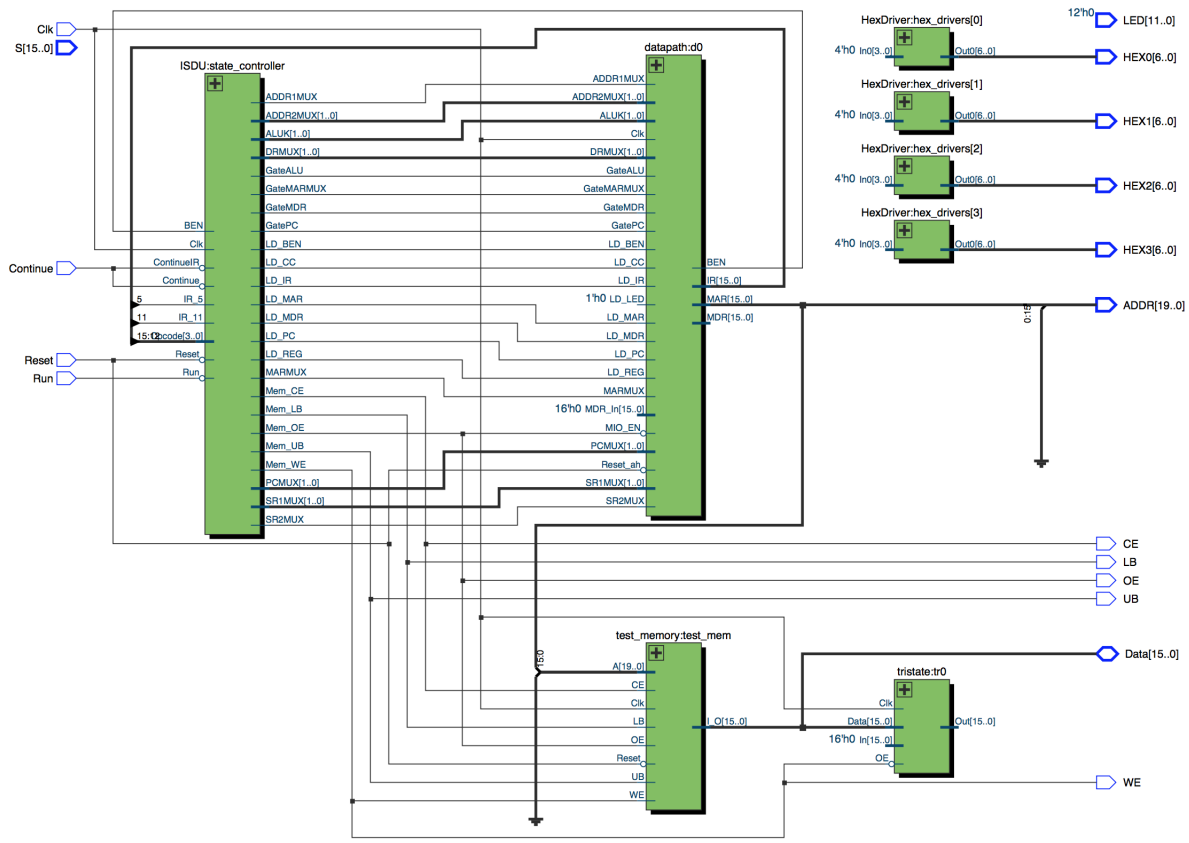

Fig. 4: Memory, MAR, MDR, Mem2IO Configuration

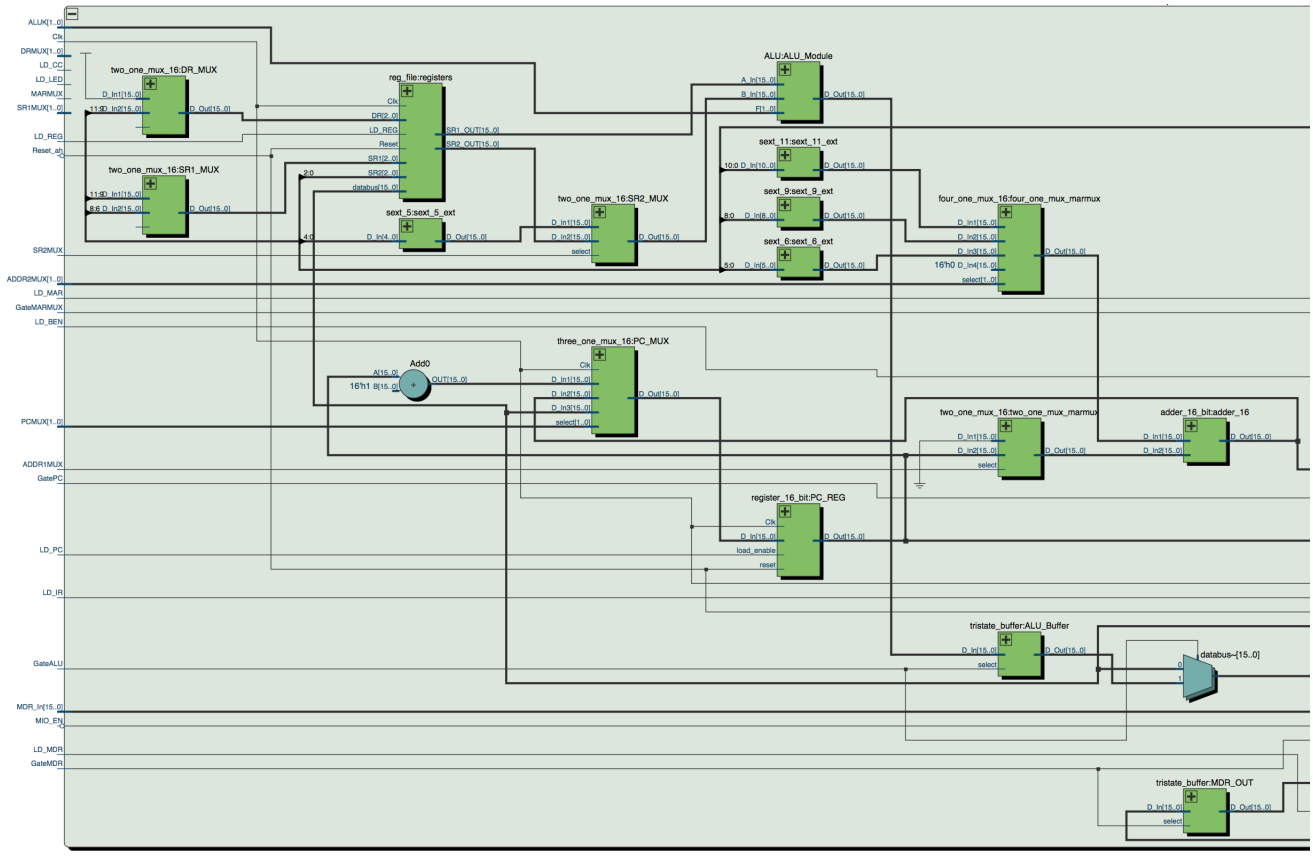

Fig. 5: Memory, MAR, MDR, Mem2IO Configuration

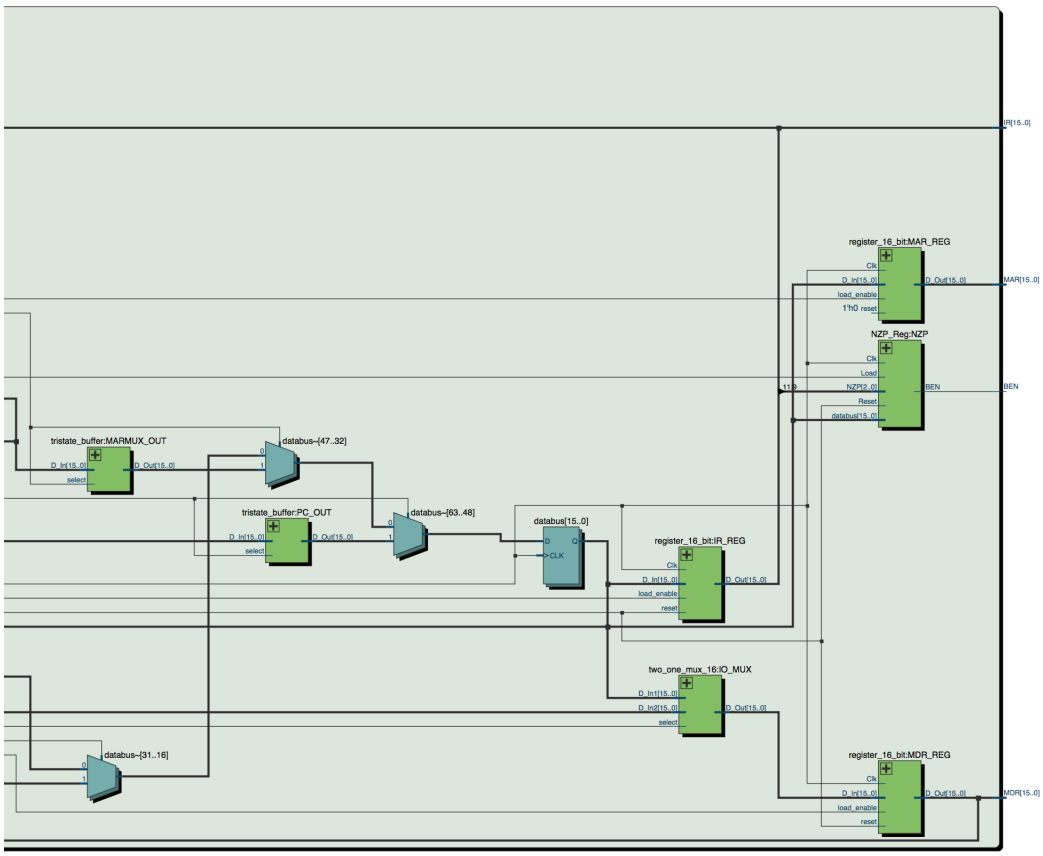

Fig. 6: Memory, MAR, MDR, Mem2IO Configuration

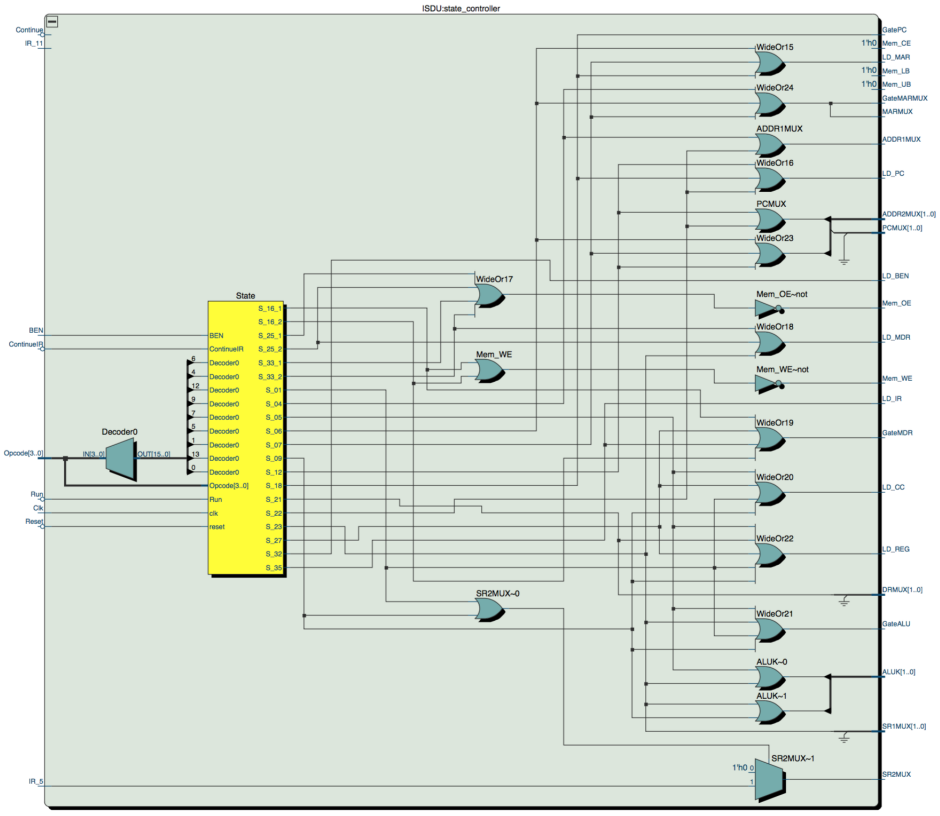

Fig. 7: Memory, MAR, MDR, Mem2IO Configuration

XII. Appendix

```
always_comb
begin
  case (State)
  Halted: ;
  S_18 : //FETCH - step 1
    begin
      GatePC = 1'b1;
      LD_MAR = 1'b1;
      PCMUX = 2'b00;
      LD_PC = 1'b1;
    end

  S_33_1 : //FETCH - step 2
    begin
      Mem_OE = 1'b0;
    end

  S_33_2 : //FETCH - step 3
    begin
      Mem_OE = 1'b0;
      LD_MDR = 1'b1;
    end

  S_35 : //FETCH - step 3
    begin
      GateMDR = 1'b1;
      LD_IR = 1'b1;
    end

  PauseIR1: ;

  PauseIR2: ;

  S_32 :
      LD_BEN = 1'b1;

  S_01 : //ADD
    begin
      SR2MUX = IR_5;
      LD_CC = 1'b1;
      ALUK = 2'b00;
      GateALU = 1'b1;
      LD_REG = 1'b1;
    end

  S_05: //AND
    begin
      LD_REG = 1'b1;
      LD_CC = 1'b1;
      ALUK = 2'b10;
      GateALU = 1'b1;
    end

  S_09: //NOT
    begin
```

```
      LD_REG = 1'b1;
      LD_CC = 1'b1;
      ALUK = 2'b01;
      GateALU = 1'b1;
      SR2MUX = IR_5;
    end

S_22: //BR
    begin
      LD_PC = 1'b1;
      PCMUX = 2'b01;
      ADDR1MUX = 1'b1;
      ADDR2MUX = 2'b01;
    end

S_12: //JMP
    begin
      LD_PC = 1'b1;
      PCMUX = 2'b01;
      ADDR1MUX = 1'b0;
      ADDR2MUX = 2'b11;
    end

S_04: //JSR - step 1
    begin
      MARMUX = 1'b1;
      GateMARMUX = 1'b1;
      ADDR2MUX = 2'b00;
      ADDR1MUX = 1'b1;
    end

S_21: //JSR - step 2
    begin
      DRMUX = 1'b1;
      LD_REG = 1'b1;
    end

S_06: //LDR - step 1
    begin
      MARMUX = 1'b1;
      GateMARMUX = 1'b1;
      ADDR1MUX = 1'b0;
      ADDR2MUX = 2'b10;
      LD_MAR = 1'b1;
    end

S_25_1: //LDR - step 2
    begin
      Mem_OE = 1'b0;
    end

S_25_2: //LDR - step 3
    begin
      Mem_OE = 1'b0;
      LD_MDR = 1'b1;
    end
```

```
  S_27: //LDR - step 4
    begin
      LD_REG = 1'b1;
      LD_CC = 1'b1;
      GateMDR = 1'b1;
    end

  S_07: //STR - step 1
    begin
      LD_MAR = 1'b1;
      ADDR1MUX = 1'b0;
      ADDR2MUX = 2'b10;
      MARMUX = 1'b1;
      GateMARMUX = 1'b1;
    end

  S_23: //STR - step 2
    begin
      SR1MUX = 1'b1;
      ALUK = 2'b11;
      GateALU = 1'b1;
      LD_MDR = 1'b1;
    end

  S_16_1: //STR - step 3
    begin
      Mem_WE = 1'b0;
      GateMDR = 1'b1;
    end

  S_16_2: //STR - step 4
    begin
      Mem_WE = 1'b0;
      GateMDR = 1'b1;
    end

  default : ;

  endcase
end
```