ECE 385

Lab 5

Simple Computer SLC-3

Fall 2025

Apoorva Sharma
NetID: as204

Samridhi Verma
NetID: sv49

Introduction

The purpose of this lab is to design a simple microprocessor using SystemVerilog. We designed the SLC-3, which consists of a CPU, memory unit, and an I/O interface that together perform the standard fetch-decode-execute instructions. The SLC-3 is a simplified version of the LC-3. It has a reduced instruction set, which omits some instructions like TRAP, LDI, and STI, which are included in LC-3. Also, there is no Ready ('R') signal from memory in the SLC-3. Instead, it compensated by holding read/write states for multiple cycles. There are no separate I/O instructions, unlike the LC-3. The SLC-3 interfaces with switches and hex displays on the FPGA through a shared memory address (0xFFFF). There is also an additional PAUSE state, which does not exist in the LC-3 instruction set. The PAUSE instructions allow for user interaction on the hardware, allowing manual step-through execution.

Written Description of the SLC-3

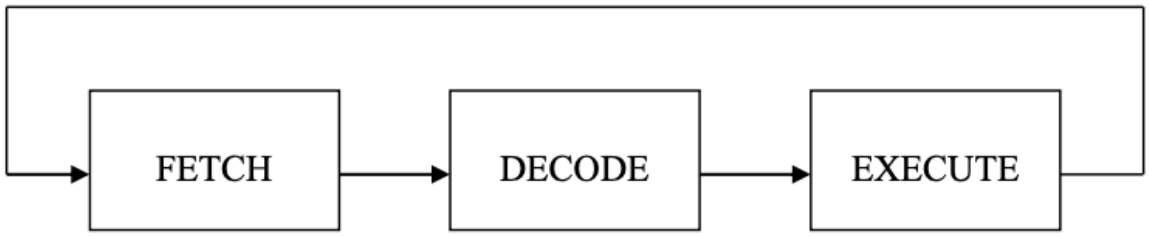


Figure 1. Fetch-Decode-Execute Cycle

The SLC-3 is a RISC processor that performs the fetch-decode-execute sequence to carry out instructions stored in the memory, as shown in *Figure 1*. It consists of a CPU, memory system, and an I/O interface. The CPU consists of a Program Counter (PC), Instruction Register (IR), Memory Address Register (MAR), a Memory Data Register (MDR), and Instruction Sequencer/Decoder, a status register (NZP), an 8x16 general purpose register file, an Arithmetic Logic Unit (ALU), a bus that interconnects all the modules, and some MUXes within the data-path to control the flow of information.

During operation, the processor repeatedly fetches instructions from memory, decodes them to determine the opcode and operands, executes the corresponding operation, and writes the result back to the registers or memory. The input/output is through the switches and hex displays on the FPGA board which are memory-mapped to the address 0xFFFF.

Fetch

The fetch phase involves loading the PC into the MAR, reading the instruction from memory into the MDR, and then transferring it into the IR. After the fetch phase, PC is incremented by 1 so that it points to the next instruction in the memory.

It involves the following 3 states:

- MAR ← PC
- MDR ← M(MAR)
- IR ← MDR

After this, the PC is incremented by 1 to go to the next state.

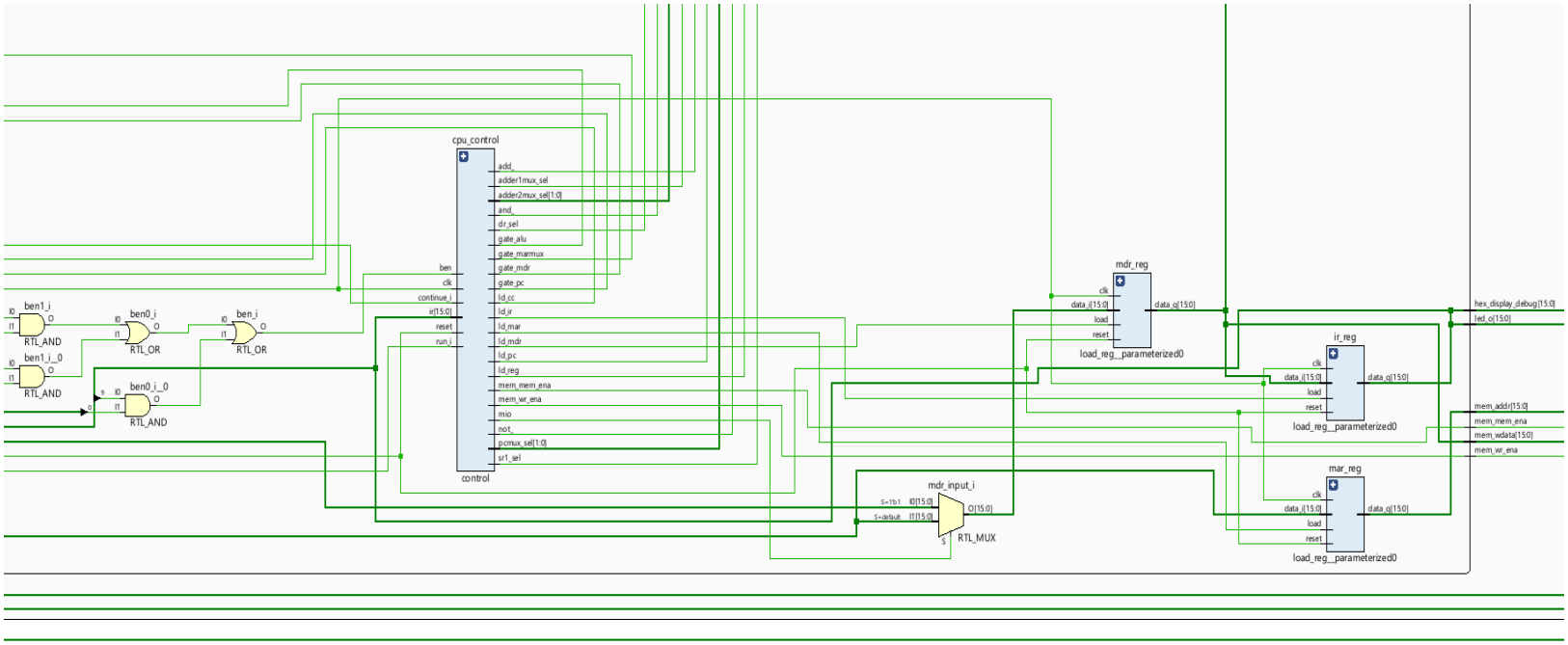
Decode

The instruction sequencer/decoder interprets the 4-bit opcode (IR[15:12]) and extracts the source and destination register addresses, immediate values, and offsets from the instruction. Based on the opcode, the control unit determines the required operation and asserts the appropriate control signals to configure the datapath for execution.

Execute

The control unit directs the datapath to perform the operation specified by the instruction. Results are written to either a destination register, to memory, or to the PC. The NZP flags are updated whenever a value is written to the register file, allowing subsequent conditional branches.

The block diagram of cpu.sv is shown in *Figure 2* (divided into 3 parts for better readability).



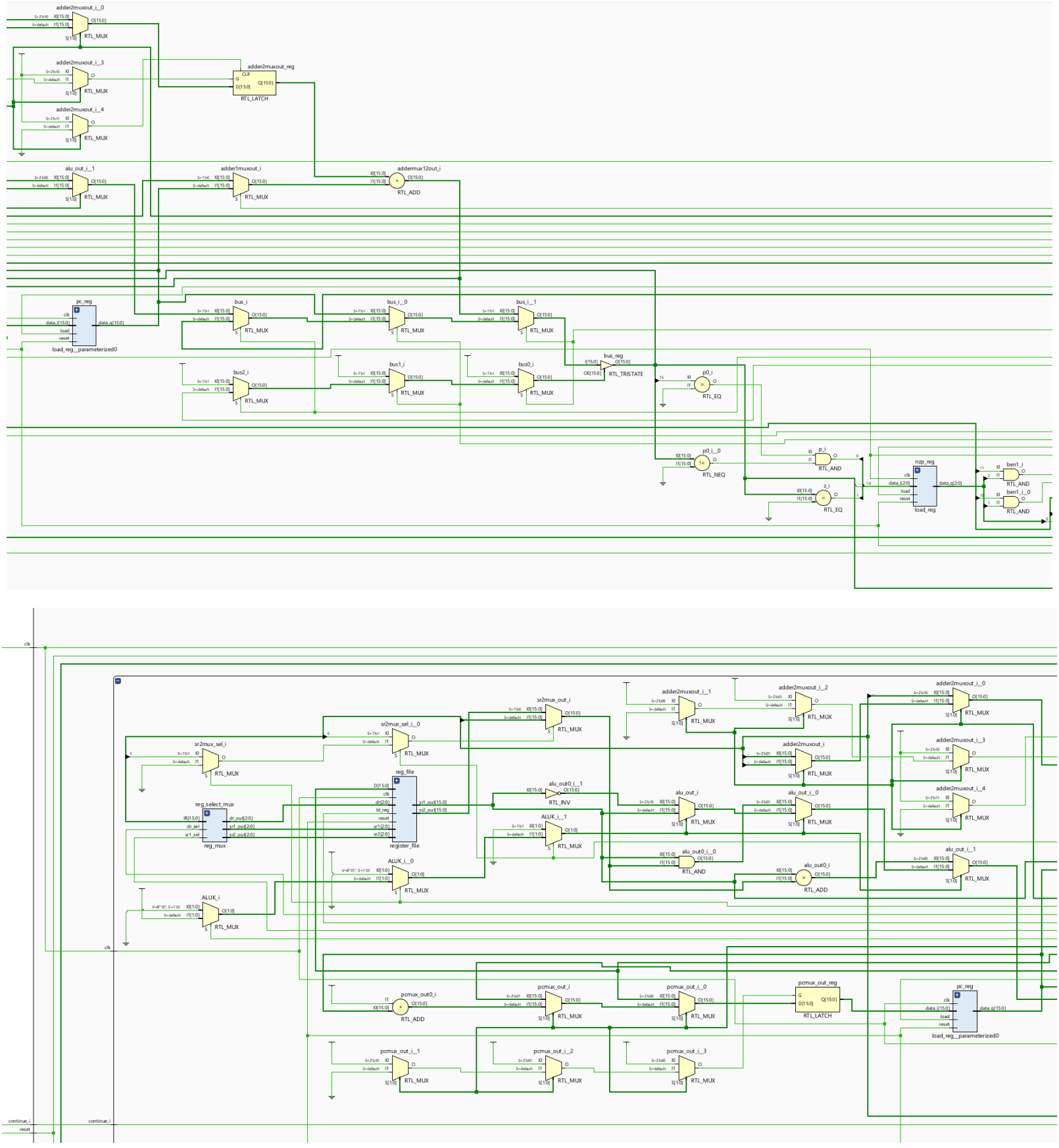


Figure 2. Block Diagram for cpu.sv

The SLC-3 can perform different operations. These include:

`ADD`: Adds two register values or a register and stores the result in a destination register. *Figure 3* shows the datapath for the R(DR) ← R(SR1) + R(SR2) operation.

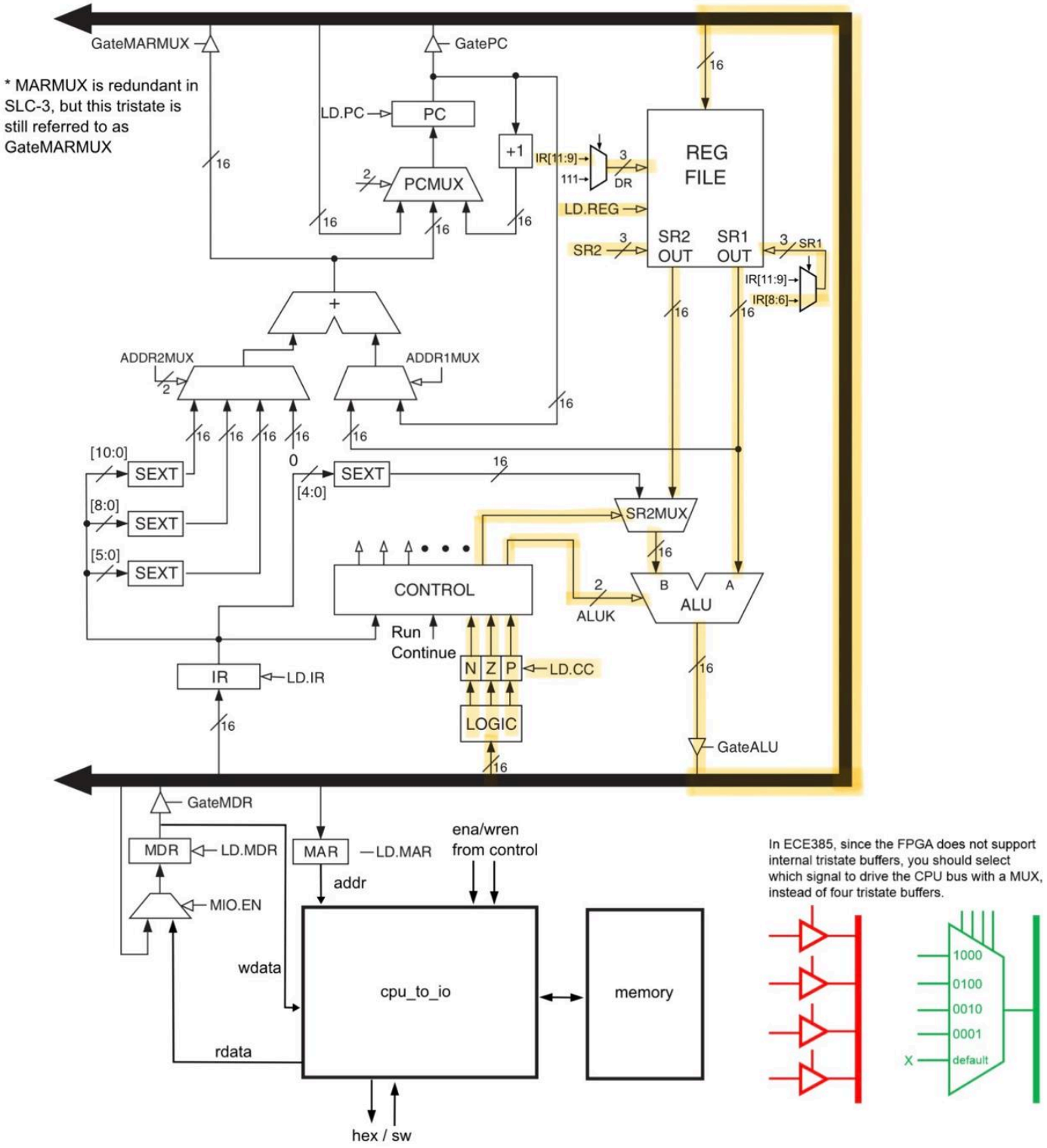


Figure 3. Datapath for the ADD Operation

`ADDi`: Adds a register value and an immediate and stores the result in a destination register. *Figure 4* shows the datapath for the R(DR) ← R(SR) + SEXT(imm5) operation.

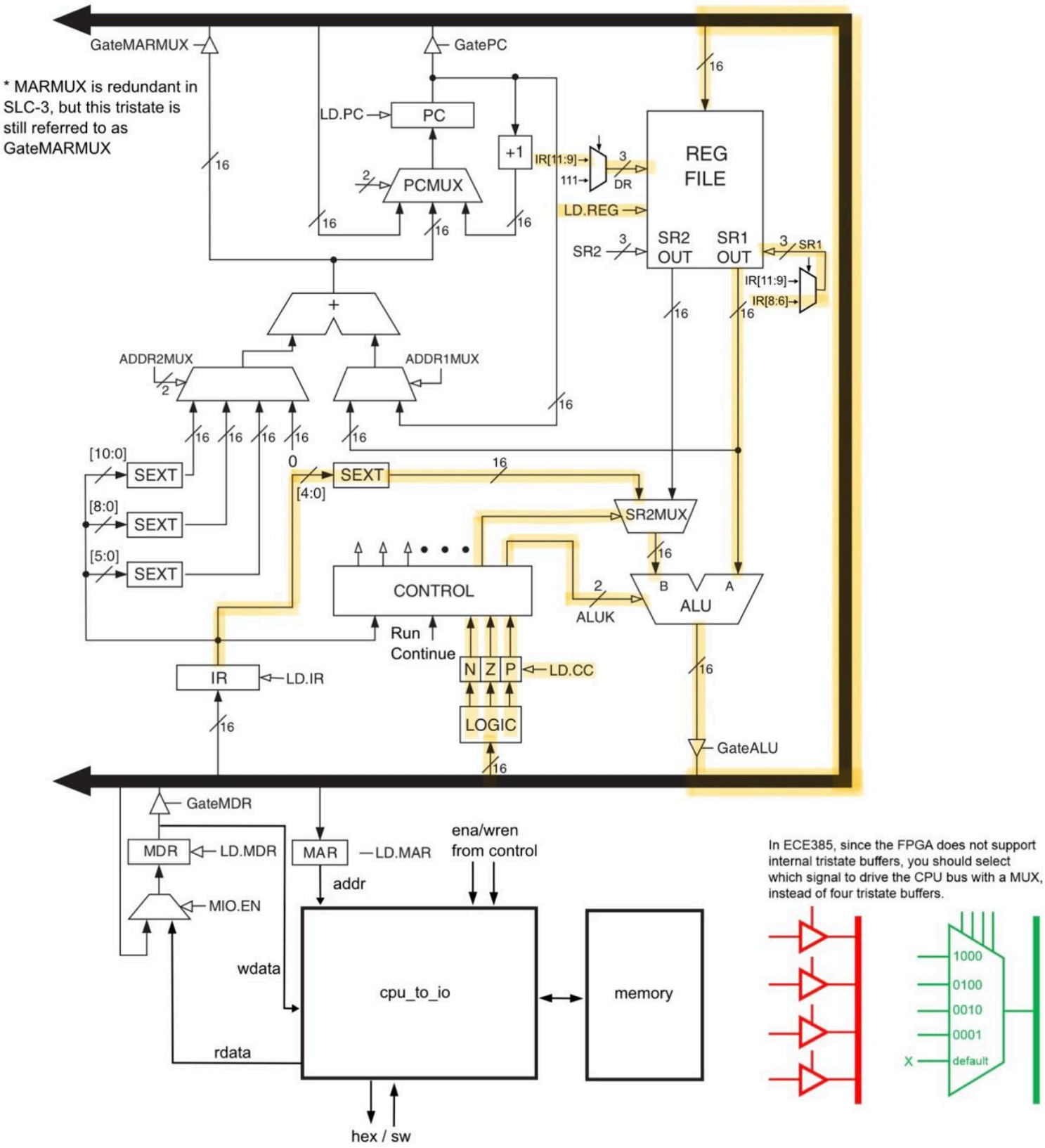


Figure 4. Datapath for the ADDi Operation

AND: Performs bitwise addition between two register values and stores the result in a destination register. *Figure 5* shows the datapath for this operation.

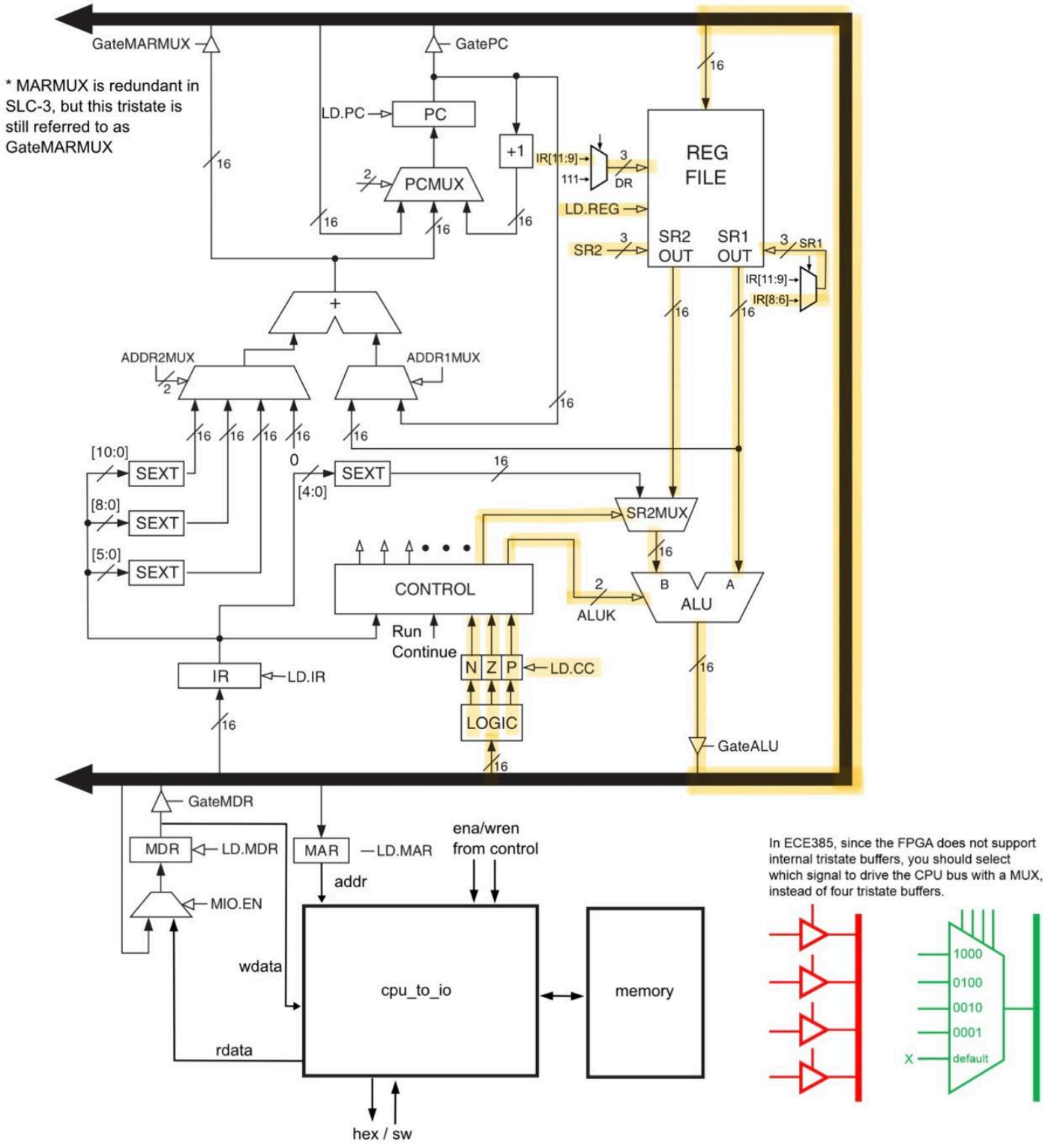


Figure 5. Datapath for the AND Operation

`ANDi`: Performs bitwise addition between a register value and an immediate and stores the result in a destination register. *Figure 6* shows the datapath for this operation.

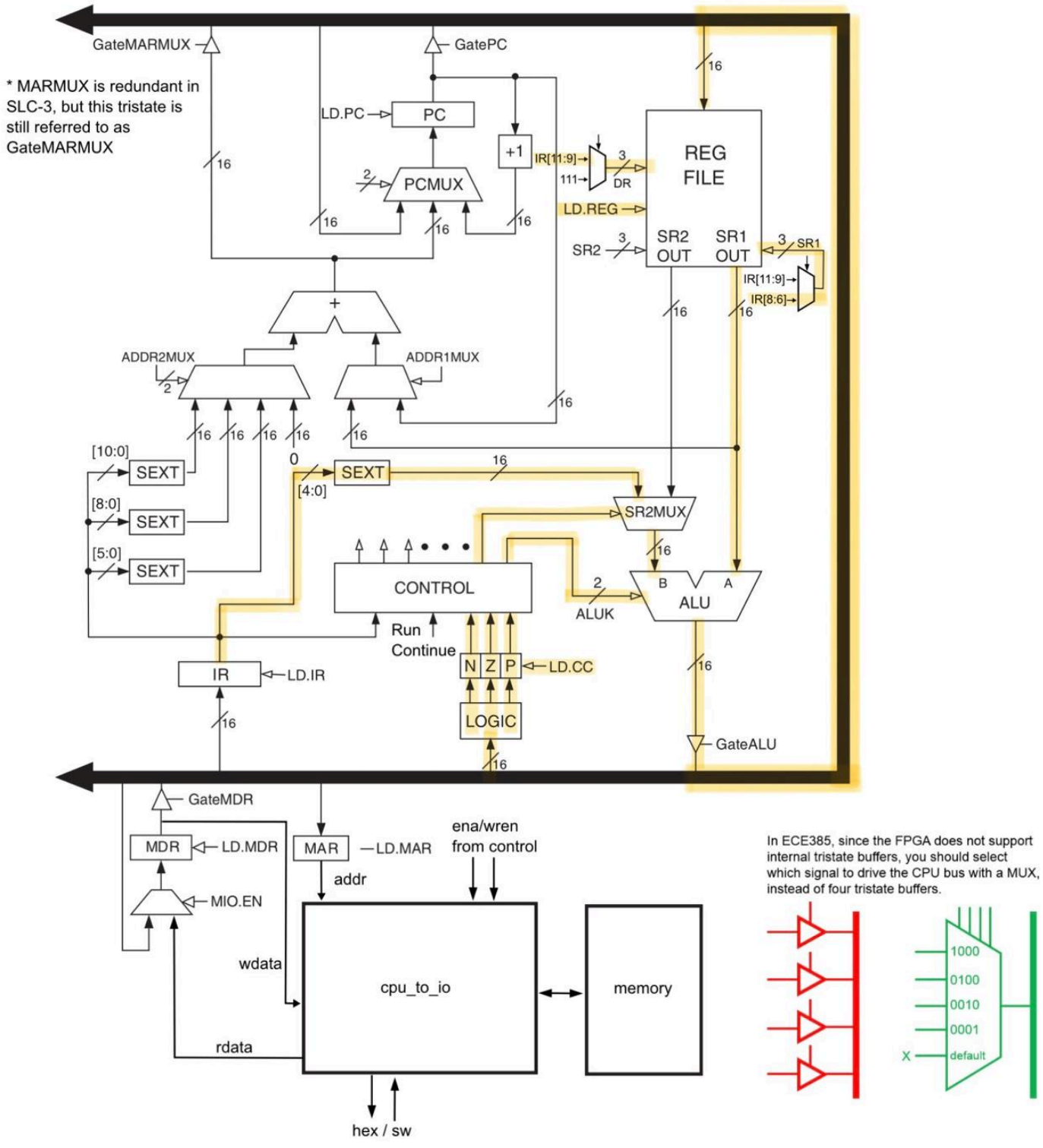


Figure 6. Datapath for the ANDi Operation

NOT: The bitwise complement of the source register is stored in the destination register. *Figure 7* shows the datapath for this operation.

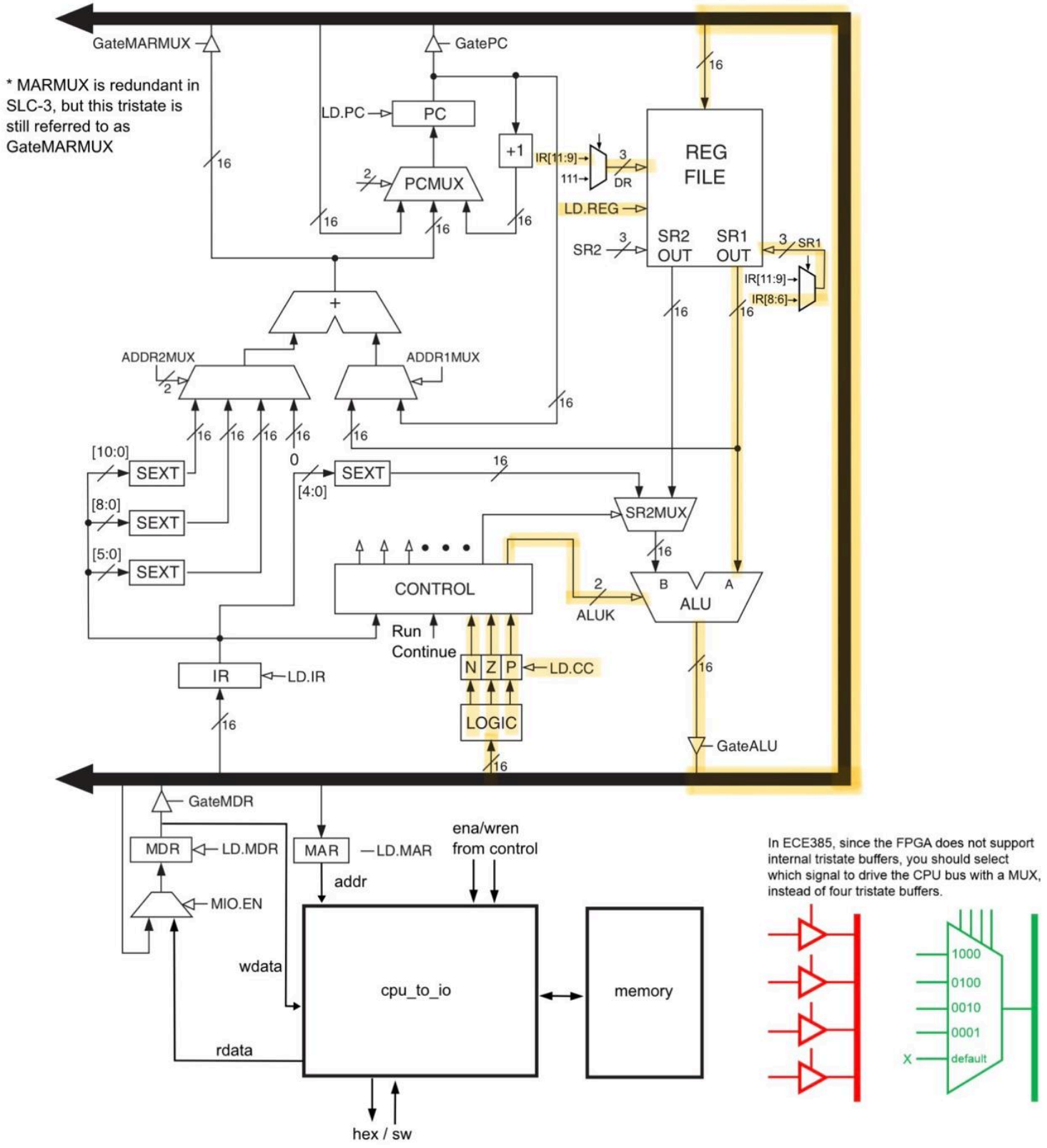


Figure 7. Datapath for the NOT Operation

BR: Branches to PC + SEXT(PCoffset9) if the corresponding NZP conditions are set. *Figure 8* shows the datapath for this operation.

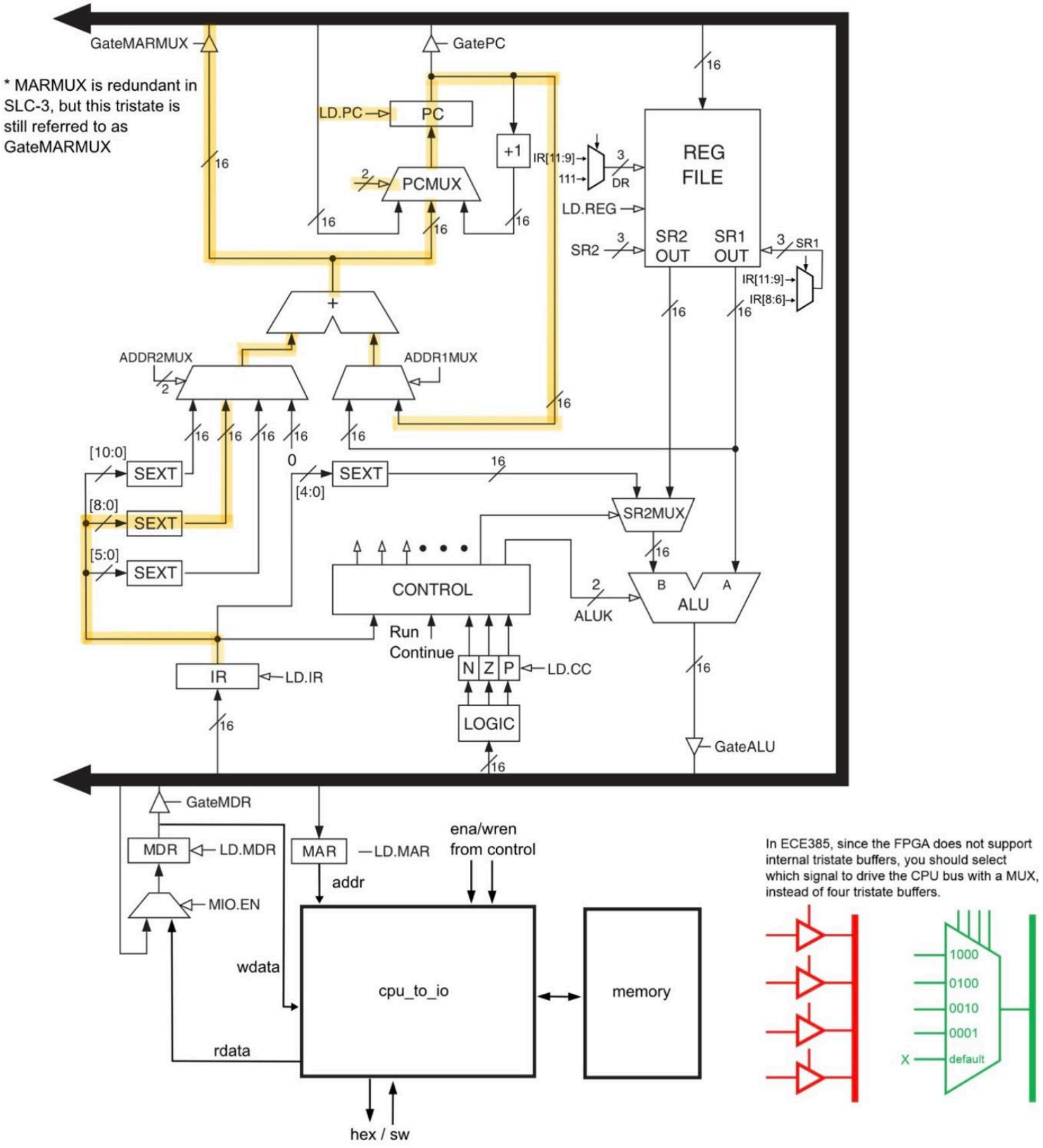


Figure 8. Datapath for the BR Operation

`JMP`: Loads PC with the value in R (BaseR). *Figure 9* shows the datapath for this operation.

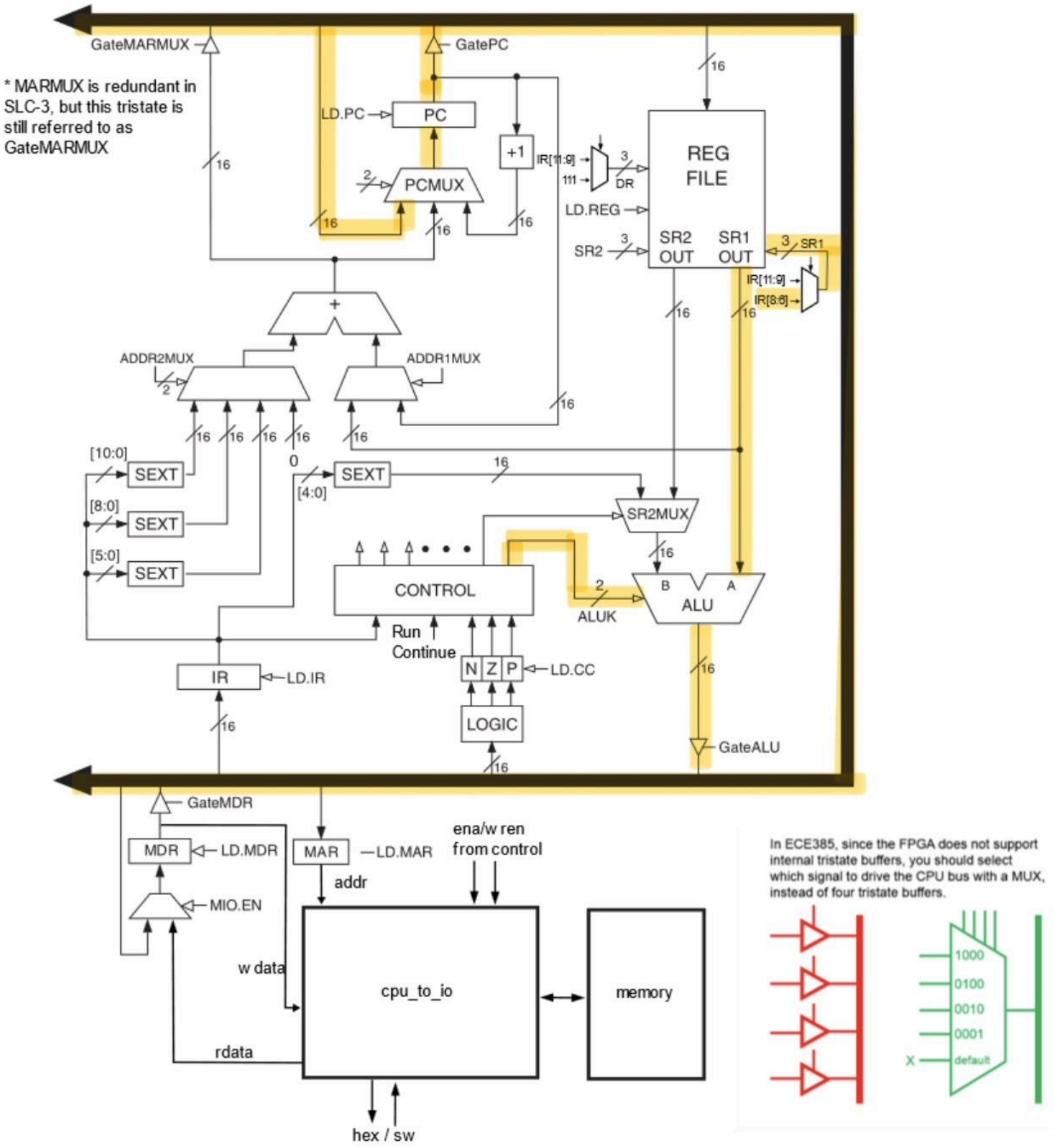


Figure 9. Datapath for the JMP Operation

`JSR`: Stores current PC in R7 and jumps to PC + SEXT(PCoffset9). *Figure 10* shows the datapath for R7 ← PC and *Figure 11* shows the datapath for PC ← PC + SEXT(PCoffset11)

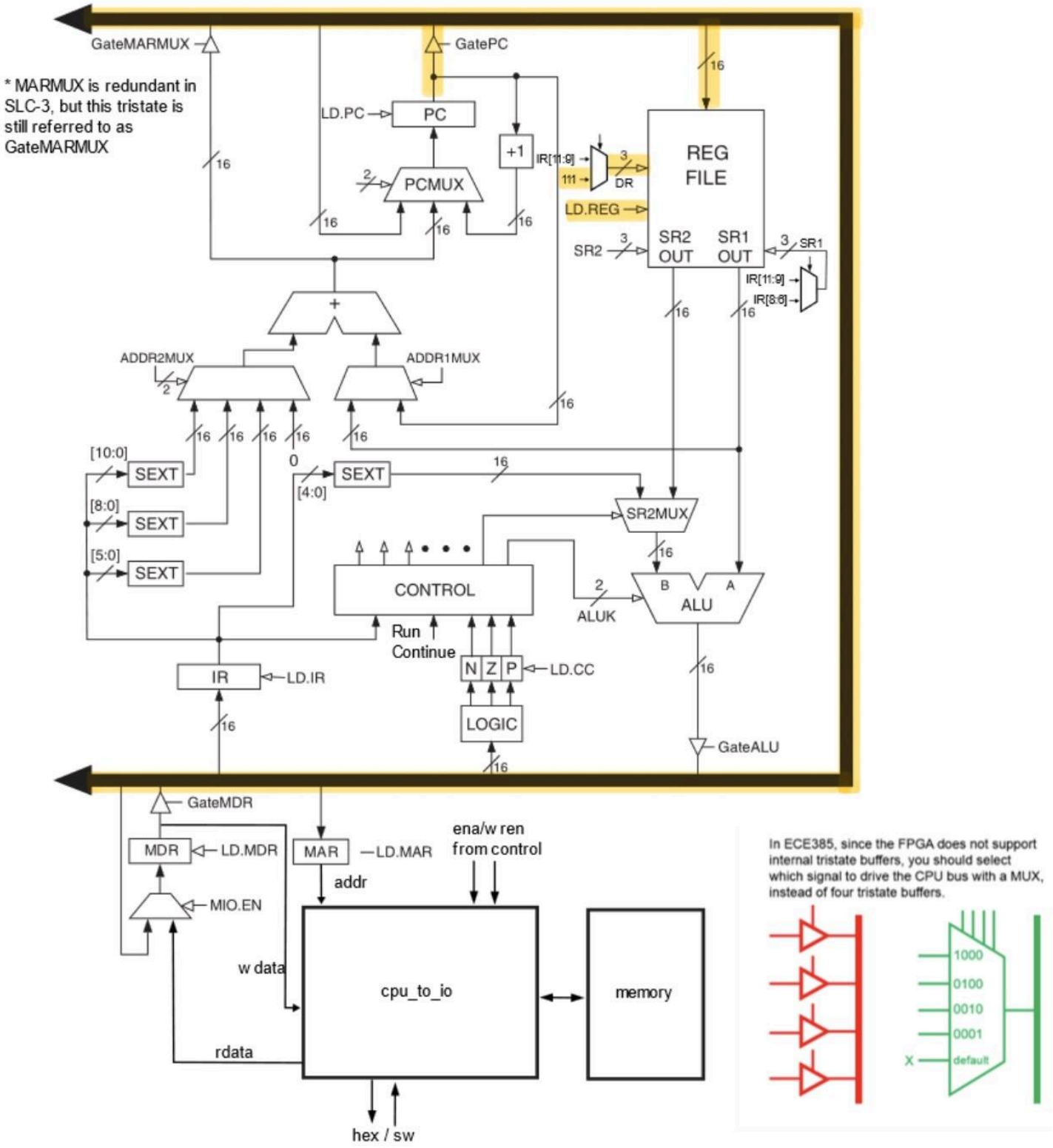


Figure 10. Datapath for Stage 1 of the JSR Operation (R7 ← PC)

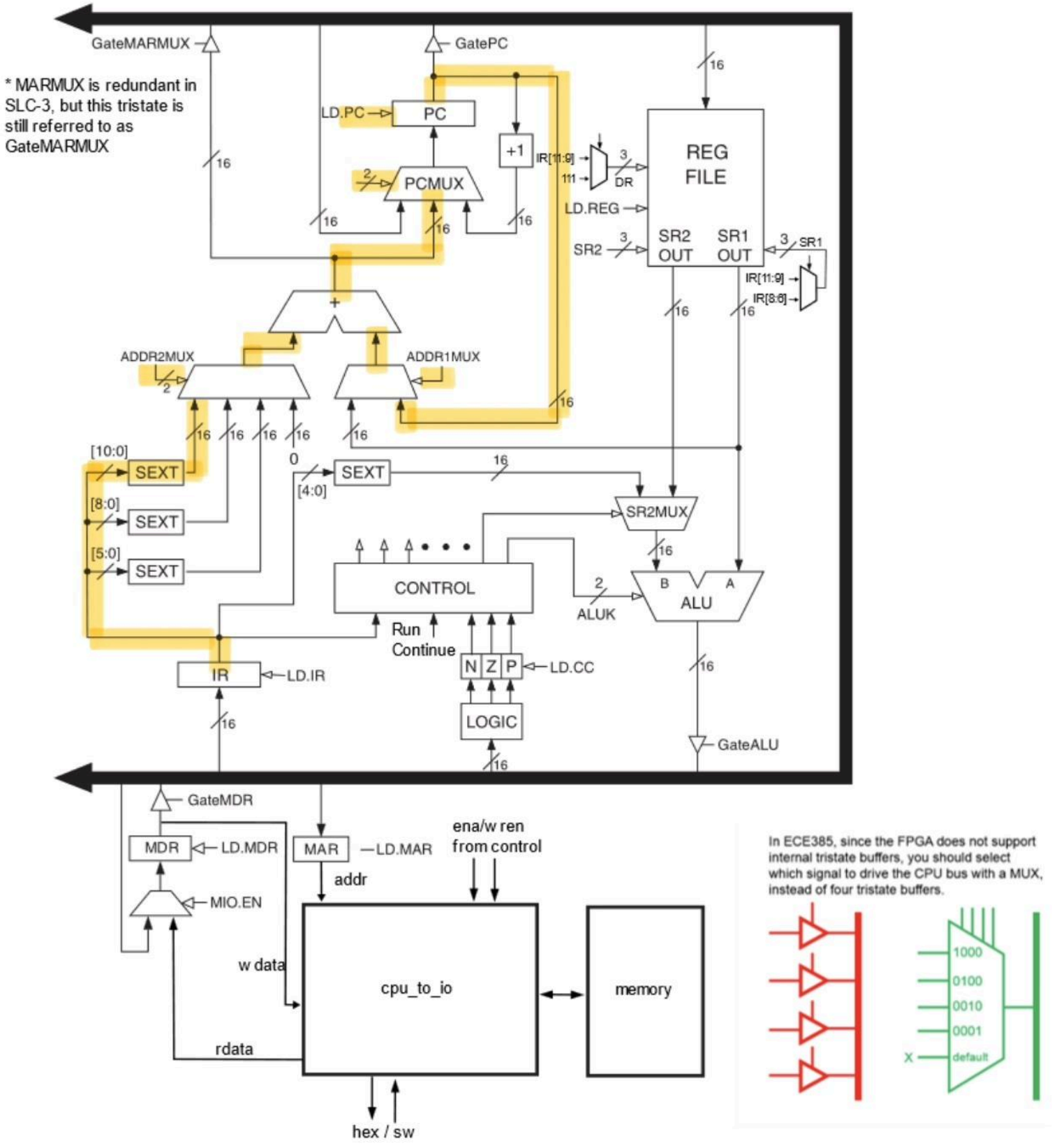


Figure 11. Datapath for Stage 2 of the JSR Operation (PC ← PC + SEXT(PCoffset11))

`LDR`: Loads the destination register with M[R(BaseR) + SEXT(offset6)]. *Figure 12* shows the datapath of the first step MAR ← B + off6. *Figure 13* shows the datapath for the second step, MDR ← M[MAR]. *Figure 14* shows the datapath for the last step, DR ← MDR, and set CC (shown in green).

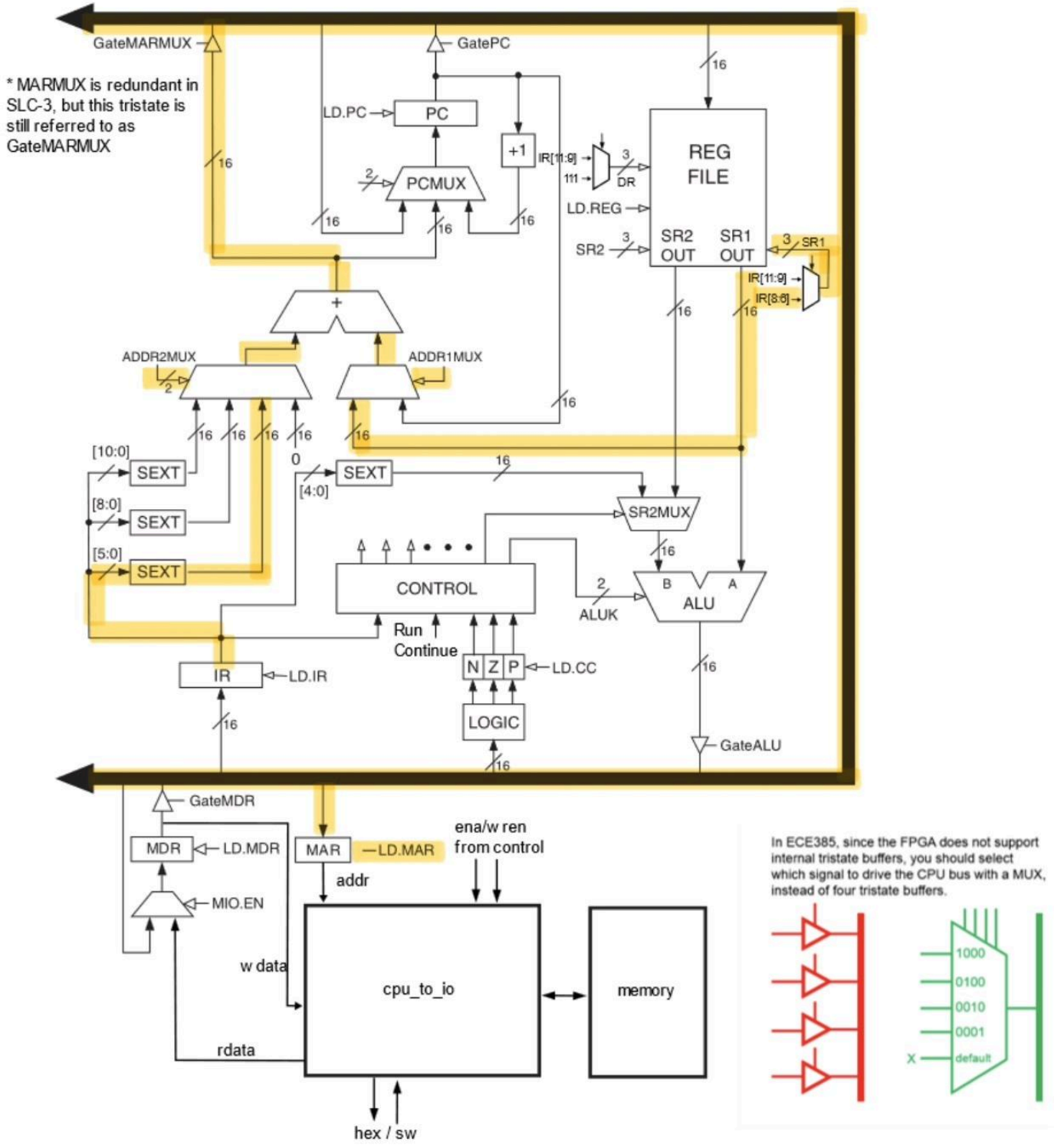


Figure 12. Datapath for Stage 1 of the LDR Operation (MAR ← B + off6)

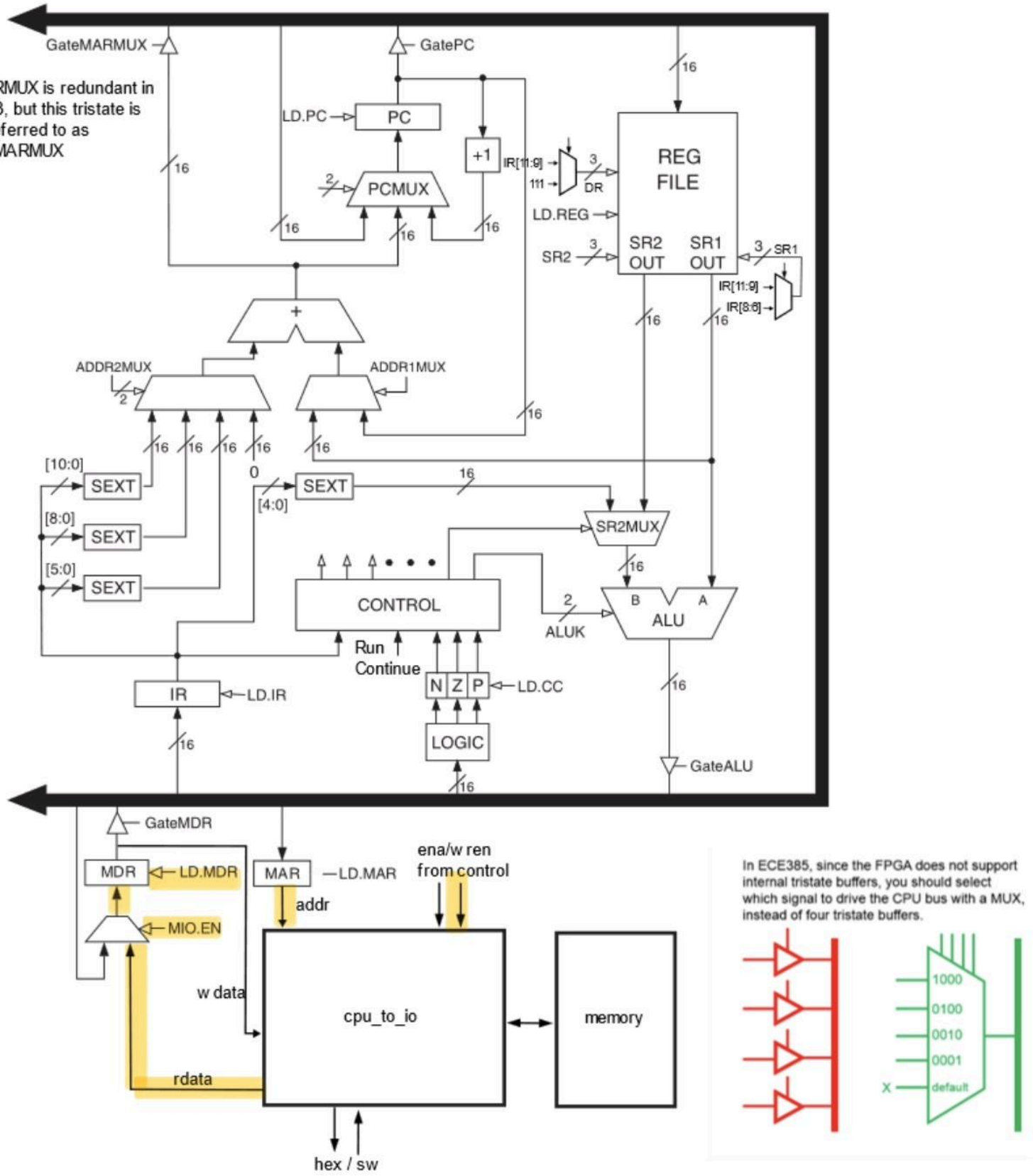


Figure 13. Datapath for Stage 2 of the LDR Operation (MDR ← M[MAR])

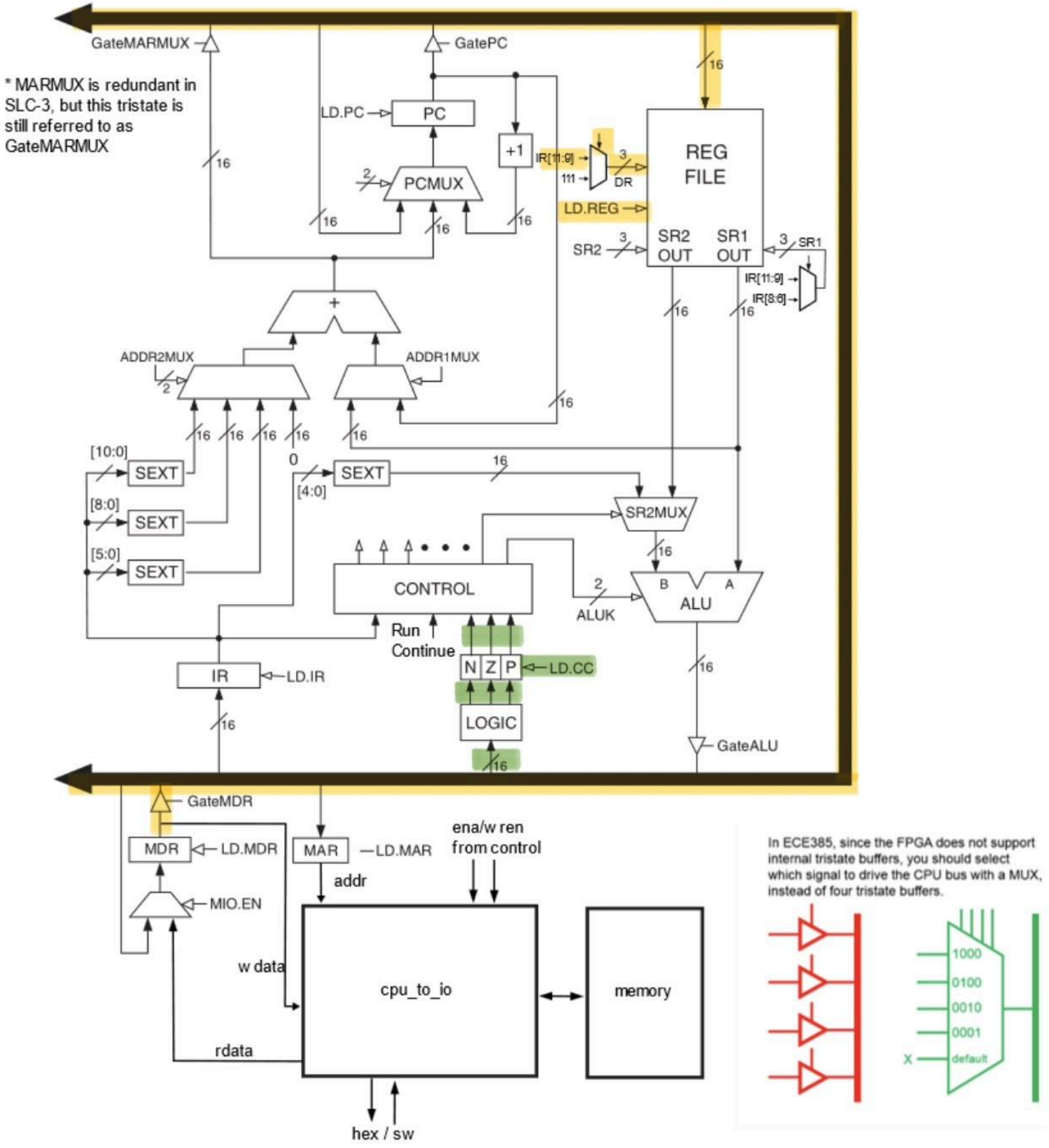


Figure 14. Datapath for Stage 3 of the LDR Operation (MDR ← M[MAR], set CC)

`STR`: Stores R(SR) in M[R(BaseR) + SEXT(offset6)]. It basically stores using the register offset addressing. Stores the contents of the source register at the memory location pointed to by M[R(BaseR) + SEXT(offset6)]. The first step MAR ← B + off6 is the same as that for LDR. This is shown in *Figure 15*. *Figure 16* shows the second stage, MDR ← SR. *Figure 17* shows the last state of the STR operation, M[MAR] ← MDR.

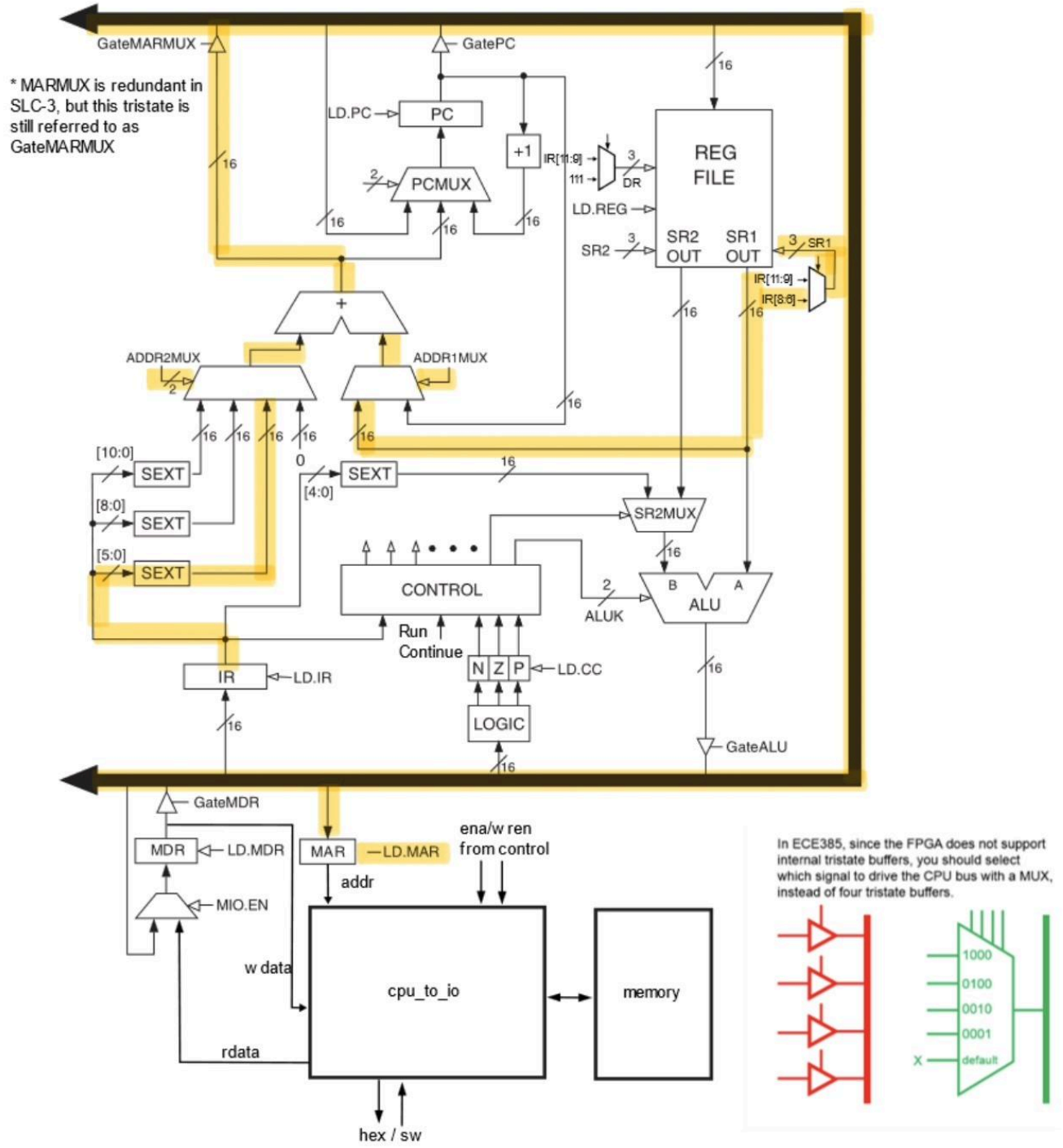


Figure 15. Datapath for Stage 1 of the STR Operation (MAR ← B + off6)

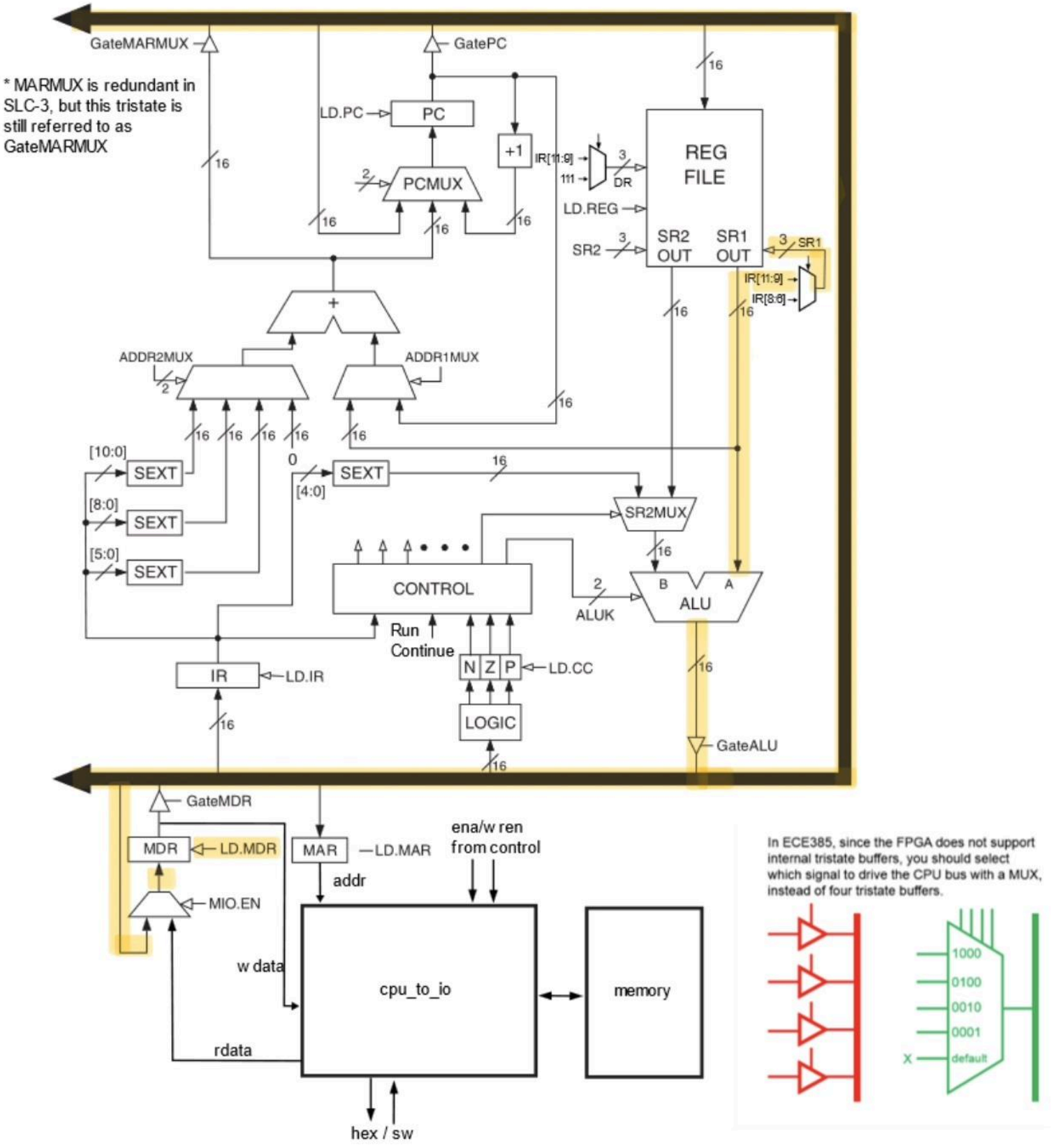


Figure 16. Datapath for Stage 2 of the STR Operation (MDR ← SR)

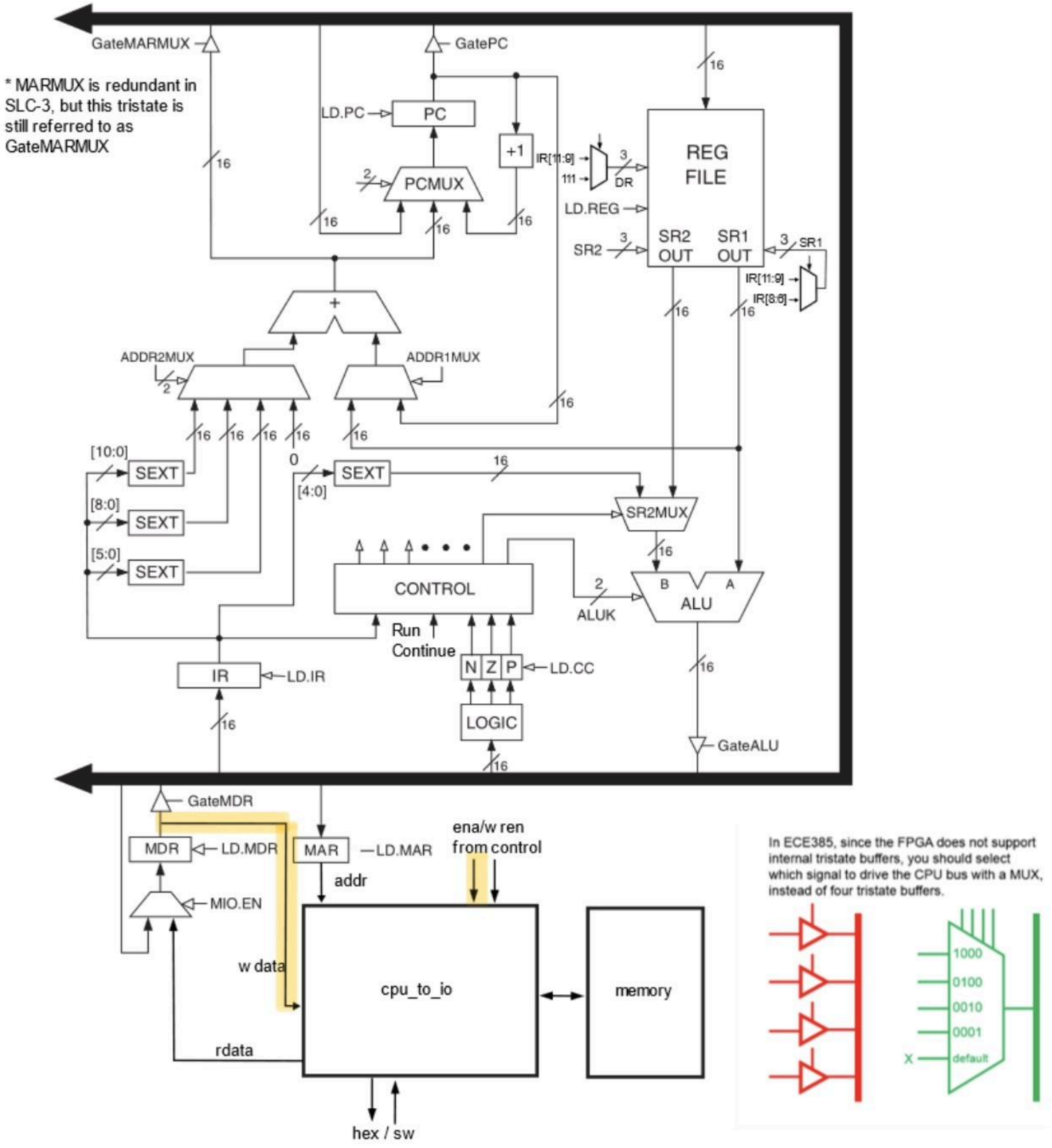


Figure 17. Datapath for Stage 3 the STR Operation (M[MAR] ← MDR)

PAUSE: This state pauses the execution until continue is pressed by the user. ledVect12 is displayed on the LEDs when paused. *Figure 18* shows the datapath for the PAUSE instruction.

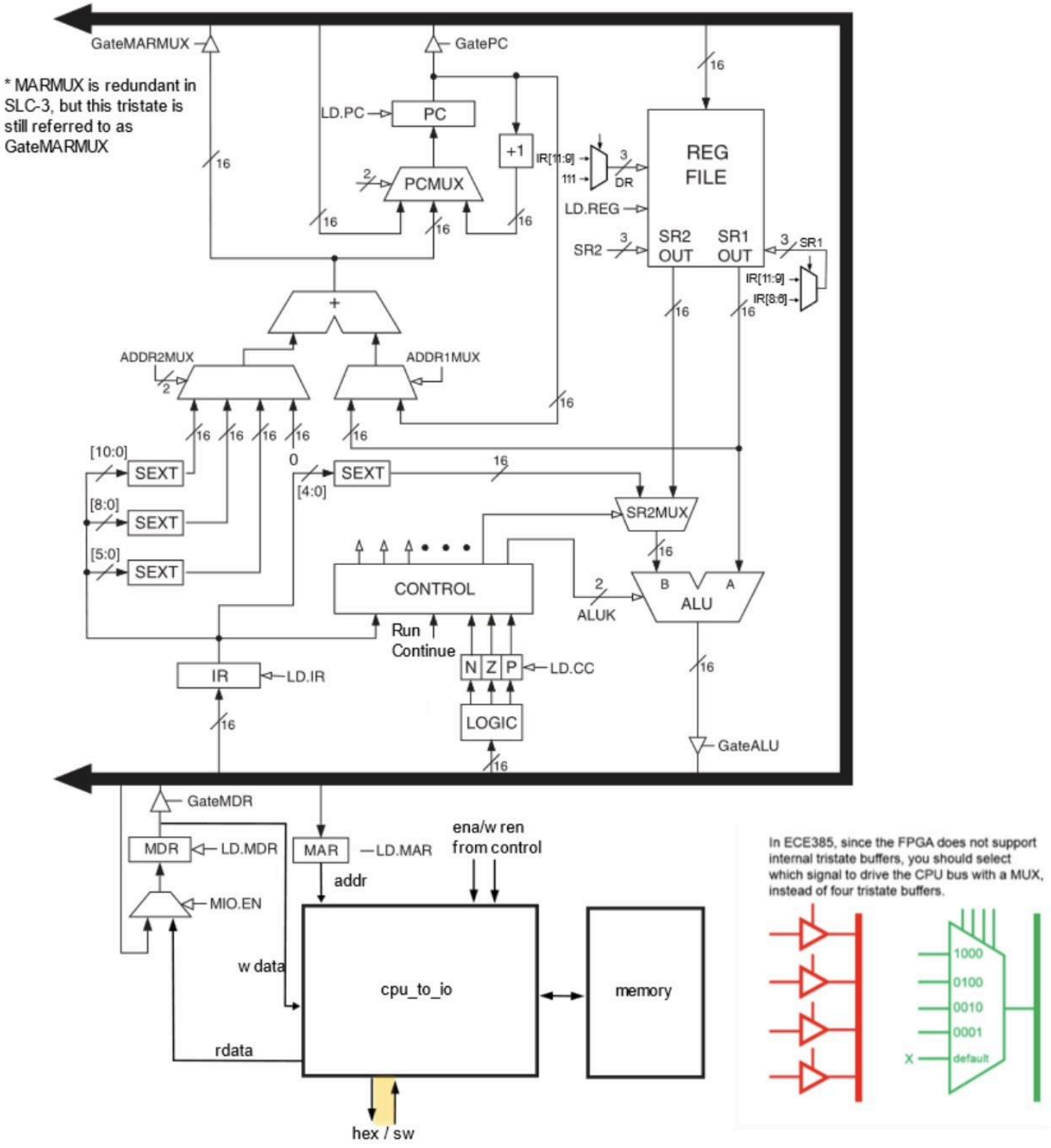


Figure 18. Datapath for the PAUSE Operation

Block Diagram of Top-Level Processor

Figure 19 shows the block diagram for the top level file.

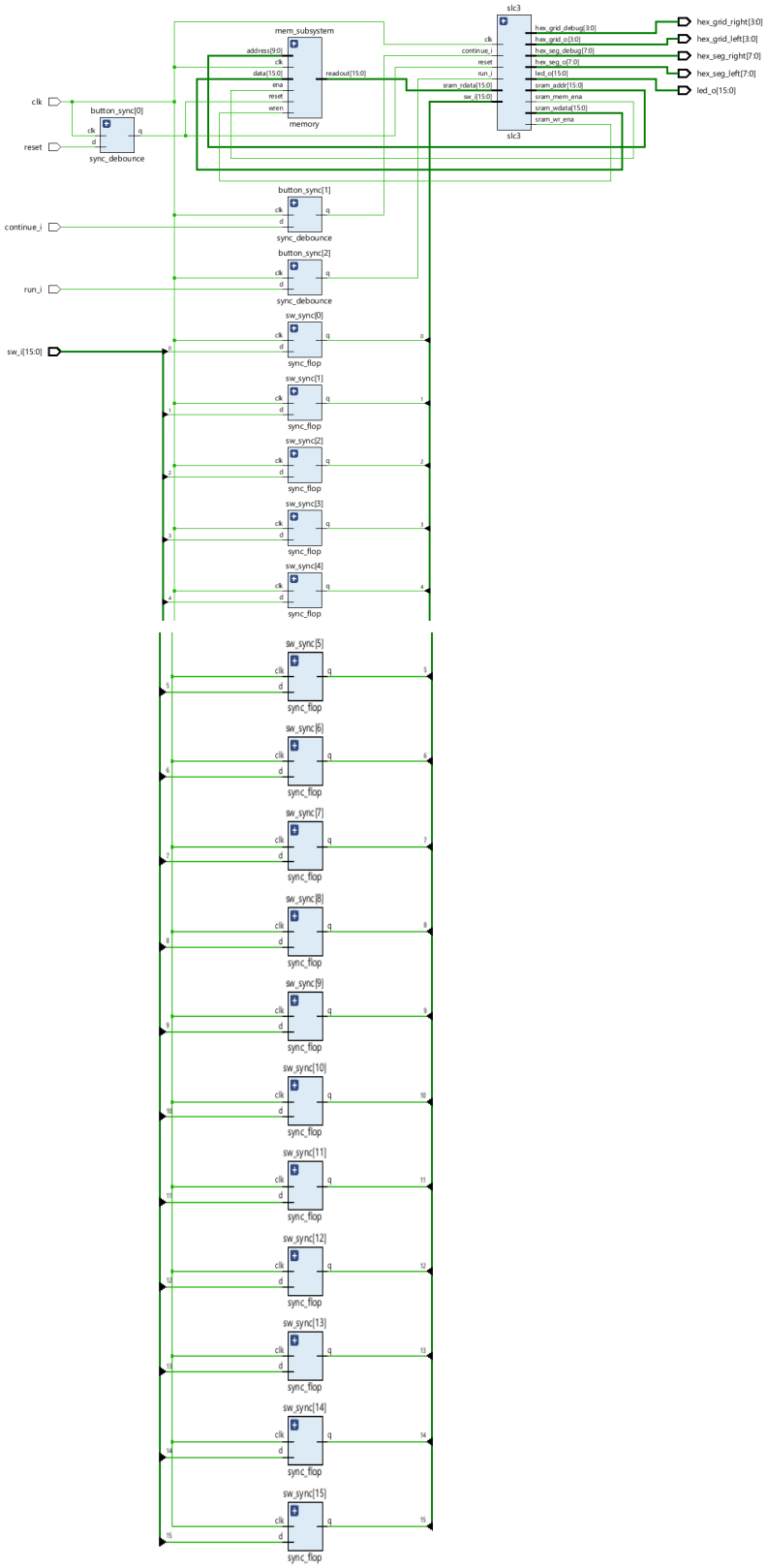


Figure 19. Block Diagram for the Top-Level File

Module Descriptions

Module: **processor_top.sv**

Inputs: `clk, reset, run_i, continue_i, [15:0] sw_i`

Outputs: `[15:0] led_o, [7:0] hex_seg_left, [3:0] hex_grid_left, [7:0] hex_seg_right, [3:0] hex_grid_right`

Description: This is the top-level module for the whole SLC-3 processor system. It connects all the main parts of the system: the processor core (slc3), memory (mem_subsystem), input buttons and switches (with debounce and sync), and output to LEDs and displays. It handles data flow and communication between the processor, memory, and inputs/outputs.

Purpose: This is the top-level module for the multiplier. It ties together all components, handling input, data flow, and display output.

Module: **slc3.sv**

Inputs: `clk, reset, run_i, continue_i, [15:0] sw_i, [15:0] sram_rdata`

Outputs: `[15:0] led_o, [7:0] hex_seg_o, [3:0] hex_grid_o, [7:0] hex_seg_debug, [3:0] hex_grid_debug, [15:0] sram_wdata, [15:0] sram_addr, sram_mem_ena, sram_wr_ena`

Description: This module is the core processor for the SLC-3 system. It connects the CPU, the I/O bridge (for interfacing the CPU with external memory and I/O devices), and a debug display (hex driver) for troubleshooting and showing internal information.

Purpose: The purpose of this module is to handle the CPU operations, manage memory and I/O, and provide debugging outputs. It is a key part of the system and works within the processor_top.sv module to create a complete processor system.

Module: **cpu.sv**

Inputs: `clk, reset, run_i, continue_i, [15:0] mem_rdata`

Outputs: `[15:0] led_o, [15:0] hex_display_debug, [15:0] mem_wdata, [15:0] mem_addr, mem_mem_ena, mem_wr_ena`

Description: This module implements the core logic of the processor in the SLC-3 system. It includes various components of the datapath, such as registers, the ALU, the memory interface, and control logic. The module is responsible for fetching instructions, decoding them, performing arithmetic and logical operations, and interacting with memory. It also controls the flow of data between different parts of the processor and updates the processor's state accordingly.

Purpose: This module serves as the central unit for processing in the SLC-3 system. Its main purpose is to execute instructions by:

1. Fetching instructions from memory.
2. Decoding and performing operations using the ALU.
3. Managing data flow through various registers (e.g., Program Counter, Memory Address Register, Instruction Register).
4. Interfacing with memory for reading and writing data.
5. Controlling the processor's state and handling condition codes (e.g., Negative, Zero, Positive).

This module combines the core computational logic of the processor and allows it to execute tasks based on the instructions it receives.

Module: **`control.sv`**

Inputs: `clk, reset, [15:0] ir, ben, continue_i, run_i,`

Outputs: `ld_mar, ld_mdr, ld_ir, ld_pc, ld_led, ld_cc, gate_pc, gate_mdr, gate_marmux, gate_alu, mem_mem_ena, mem_wr_ena, dr_sel, sr1_sel, ld_reg, [1:0] pcmux_sel, [1:0] adder2mux_sel, adder1mux_sel, mio, add_, and_, not_`

Description: This module generates the control signals that manage the operations of the entire CPU architecture. It controls the flow of data within the system by controlling when specific registers (e.g., mar, mdr, ir, pc, etc.) are loaded, when data is written to memory, and when different components (e.g., ALU, register file) are enabled. This module is driven by an FSM that determines the order of execution of operations.

Purpose: This module manages the sequential execution of operations within the CPU, enabling or disabling various datapath components at the correct times. It controls the timing of memory reads/writes, the execution of ALU operations, and the fetching/decoding of instructions. Overall, this module is the "controller" of the CPU and ensures that each operation is executed in the right order and at the right time.

Module: **`load_reg.sv`**

Inputs: `clk, reset, load, [DATA_WIDTH-1:0] data_i`

Outputs: `[DATA_WIDTH-1:0] data_q`

Description: This module is a parameterized register that stores data based on a load signal. It has a configurable width (DATA_WIDTH) and stores the value of data_i when load is active. On each clock cycle, the stored value is output as data_q. It also supports asynchronous reset.

Purpose: This module stores data and outputs it on the clock edge. It's useful for holding intermediate values, control signals, or buffering data in a digital system, with a configurable data width.

Module: **reg_mux.sv**

Inputs: `[15:0] ir, dr_sel, sr1_sel`

Outputs: `[2:0] dr_out, [2:0] sr1_out, [2:0] sr2_out`

Description: This module selects the values of registers for different operations based on the instruction input ir and the control signals dr_sel and sr1_sel. It produces outputs for the destination register (dr_out), source register 1 (sr1_out), and source register 2 (sr2_out), where source register 2 is always derived from IR[8:6].

Purpose: This module is used for selecting which registers to use as source and destination based on the instruction in the Instruction Register (IR). It helps to route data to the correct registers for execution in the CPU, depending on the control signals.

Module: **hex_driver.sv**

Inputs: `[15:0] ir, dr_sel, sr1_sel`

Outputs: `[2:0] dr_out, [2:0] sr1_out, [2:0] sr2_out`

Description: This module consists of 8 16-bit registers, which can be read and written. The module allows loading data into a register, and on a clock edge, it updates the register or clears all registers on reset. The module outputs the values stored in the source registers sr1 and sr2.

Purpose: This module serves as a register file in a CPU and holds values for computations. It reads from two registers (sr1 and sr2) and writes data to a destination register (dr). The module also supports reset functionality to clear all registers.

Module: **cpu_to_io.sv**

Inputs: `clk, reset, run_i, continue_i, [15:0] sw_i`

Outputs: `[15:0] led_o, [7:0] hex_seg_left, [3:0] hex_grid_left, [7:0] hex_seg_right, [3:0] hex_grid_right`

Description: This module interfaces between the CPU and external I/O devices (e.g., switches, LEDs, and the memory). It bridges the CPU's memory operations to the actual external memory BRAM and provides an interface for I/O devices.

Purpose: This module bridges the CPU external I/O devices and facilitates data transfer between them. It also manages control signals for memory operations, like read and write enable operations. It ensures that information from the CPU can be displayed on hex displays and enables interactions with I/O devices like switches and LEDs.

Module: **hex_driver.sv**

Inputs: `clk, reset, [3:0] in[4]`

Outputs: `[7:0] hex_seg, [3:0] hex_grid`

Description: This module converts the 4 nibbles of hexadecimal values (in[4]) into 7-segment display codes using the nibble_to_hex submodule. It then updates each display one by one using a counter that shows the corresponding value for each nibble. We use the reset button to ensure that the display is cleared when not in use.

Purpose: This module converts the binary values from registers A and B into their corresponding hexadecimal values and displays them on the 7-segment LEDs of the Urbana Board.

Module: **instantiate_ram.sv**

Inputs: `clk, reset`

Outputs: `[9:0] addr, wren, [15:0] data`

Description: This module instantiates and manages the on-chip memory in the SLC-3 system. The memory is initialized during system startup, and its address is incremented during operation. Once initialization is complete, the system enters a normal operational mode, and the memory address is continuously updated.

Purpose: This module manages the initialization and interaction with the on-chip RAM. It handles address generation, memory writes, and data outputs. When the system is reset, the memory is initialized, and once initialization is complete, the memory operates normally, providing data based on the current address.

Module: **sync.sv**

Inputs: `clk, d`

Outputs: `q`

Description: This module uses two flip-flops to get the input signal and a counter to verify if the signal has remained stable for long enough to be valid. During synthesis, the counter waits for a longer period to ensure stability, while during simulation, it only waits for one cycle (since the signal is not coming from a physical switch).

Purpose: All button inputs are sent through this module to ensure they are stable before they reach the circuit. This is especially important for buttons like Execute, Reset, LoadA, and LoadB. Additionally, we use this for the 8-bit data input, 3-bit F input, and 2-bit R input to ensure their stability.

Module: **memory.sv**

Inputs: `clk, reset, [15:9] data, [9:0] address, ena, wren`

Outputs: `[15:0] readout`

Description: This module handles memory operations for the system. It provides a test memory for simulation and uses BRAM for synthesis, allowing memory read and write operations. It also initializes memory for synthesis.

Purpose: This module interfaces the processor with memory by managing operations like reading from and writing to memory. It uses a real external memory (BRAM) in synthesis and a simulated memory model for testing to ensure that the CPU can store and retrieve the required data.

Module: **test_memory.sv**

Inputs: `clk, reset, [15:0] data, [9:0]  address, ena, wren`

Outputs: `[15:0] readout`

Description: This module simulates the on-chip memory for the SLC-3 processor and provides read and write functionality for testing purposes.

Purpose: This module is designed to be used in testbenches for memory simulation. It is used for debugging and verification of memory-related operations.

Module: **types.sv**

Description: This file contains functions that create machine instructions for the SLC-3 processor. These instructions include things like adding numbers, jumping to different parts of the program, or calling functions.

Purpose: This file is used to test if the SLC-3 processor is working correctly or not by simulating real-world operations.

Control Unit

Figure 20 shows the state diagram for the finite state machine of the control unit.

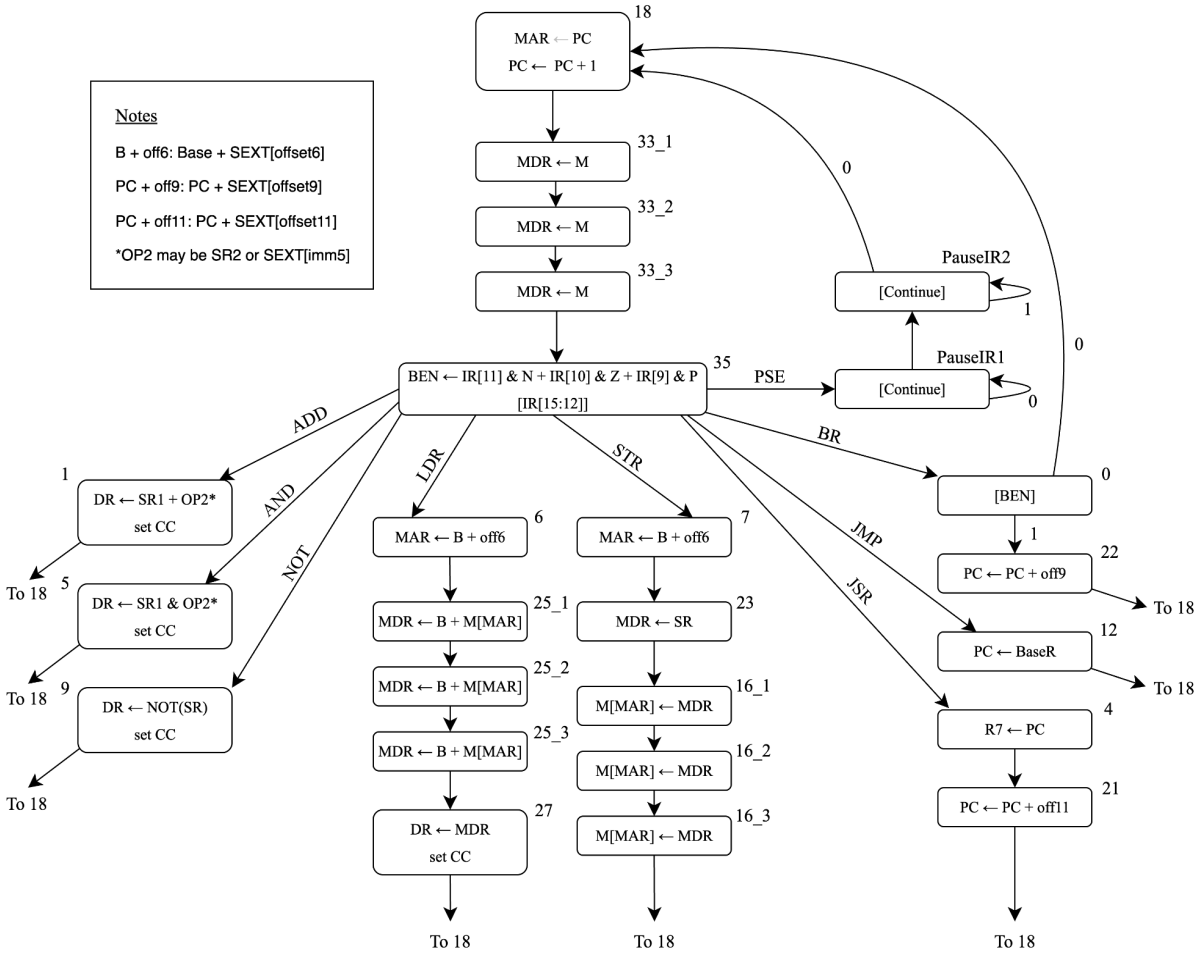


Figure 20. State Diagram of the Control Unit

The control unit is like the "brain" of the processor. It decides what happens at each clock cycle. It's built as a finite state machine (FSM). When the processor is running, the control unit goes through three main phases:

1. **Fetch phase** - This is where the processor gets the next instruction from memory. The control unit enables signals such that the Program Counter (PC) sends the address to memory. Then, the instruction at that address is loaded into the Instruction Register (IR). The PC is then incremented to point to the next instruction.

2. **Decode phase** - Once the instruction is fetched, the control unit looks at the opcode (the upper bits of IR, like IR[15:12]) to figure out what type of instruction it is. It checks whether the instruction is an ADD, AND, NOT, LDR, STR, BR, JSR, JMP, or PSE. Based on this, it decides which path or sequence of states to follow next.

3. **Execute phase** - This is where the control unit activates the correct control signals to actually carry out the instruction.
    a. For ADD, AND, or NOT instructions, it enables the ALU and loads the result into a register.
    b. For a load (LDR) or store (STR), it enables the memory read or write signals.
    c. For a branch (BR), it checks condition flags (like the "ben" signal) to decide if the PC should change.
    d. For jump (JMP) or subroutine (JSR), it updates the PC accordingly.

Simulations

Basic I/O Test 1

Figure 21 shows the simulation for Basic I/O Test 1.

From the Test Programs Documentation, "This program uses ANDi to clear R0 (i.e., ANDi R0, R0, 0), which is used as a base register for memory operations (using negative values for the offset). All test programs do this to set up for I/O operations. The program then reads in the data on the switches, writes this value to the hex display, and loops back (using BRnzp as an unconditional jump) to repeat the process indefinitely. When working correctly, the hex displays should appear to always display the value of the switches."

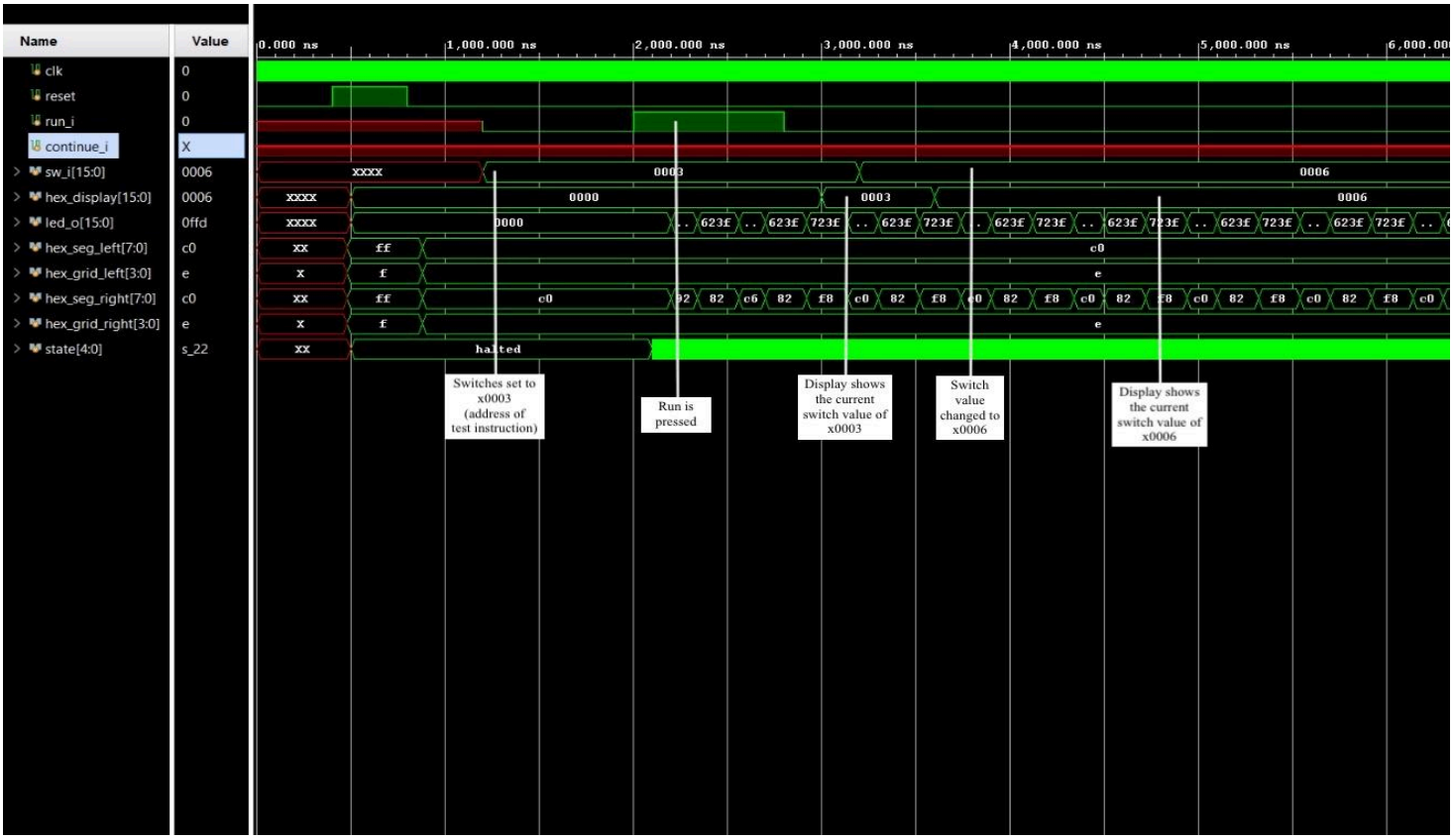


Figure 21. Simulation for Basic I/O Test 1

Basic I/O Test 2

Figure 22 shows the simulation for Basic I/O Test 2.

From the Test Programs Documentation, "The code for this program is identical to the previous test, except that it uses pause instructions to ask for input and report output. The first pause instruction will ask for input on the switches. The second and subsequent pauses will display x02 and both 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."

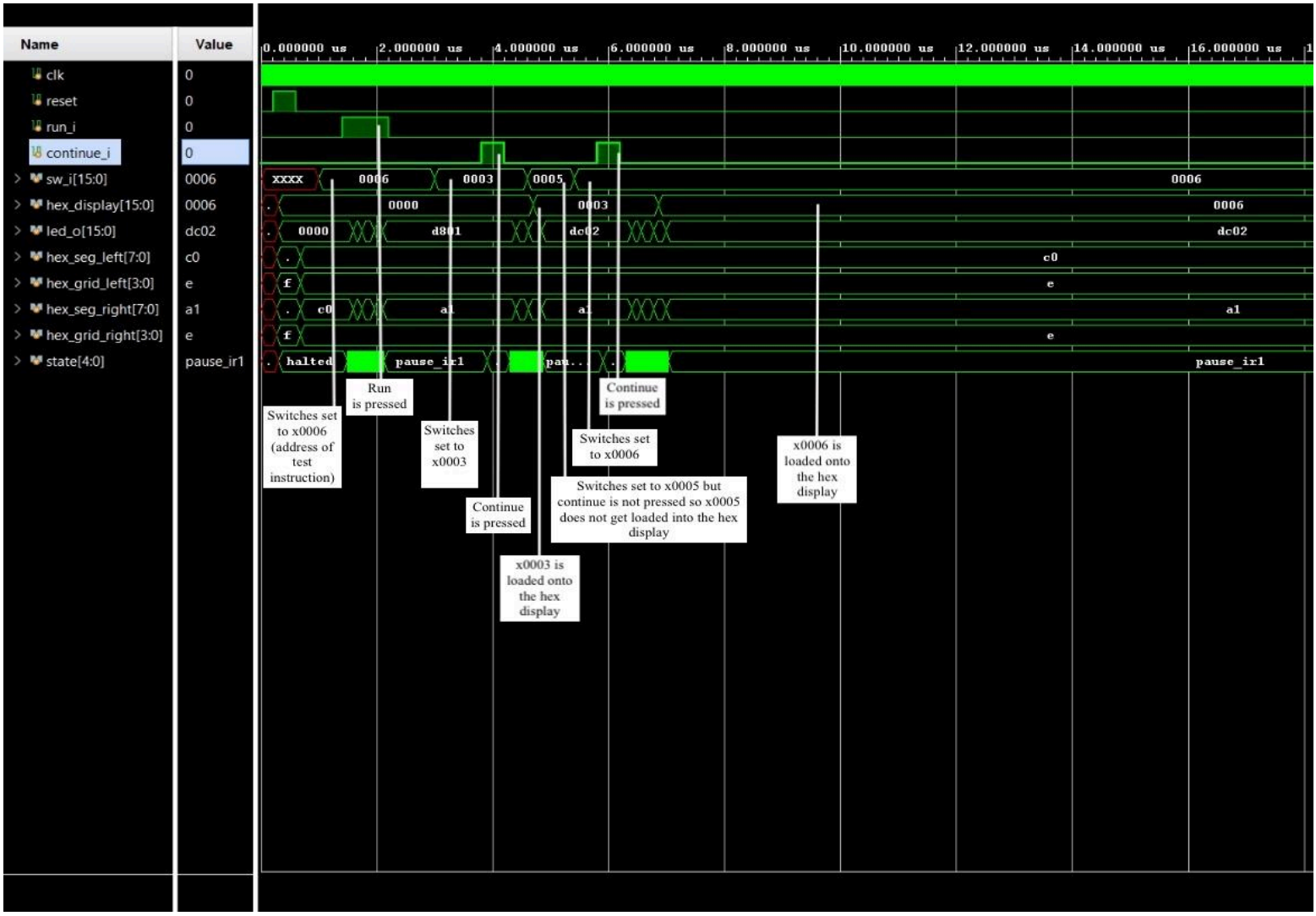


Figure 22. Simulation for Basic I/O Test 2

Self-Modifying Code Test

Figure 23 shows the simulation for the Self-Modifying Code Test.

From the Test Programs Documentation, “This program is based upon the same loop as the previous test, but inserts some additional operations. Before the loop begins, JSR 0 is executed. This serves to put the PC address in R7 without actually changing the value of the PC. This PC value is used to load the data for the second pause instruction into a register. The loop then operates normally, except that in each iteration, the data for the pause instruction is incremented and stored back to the proper memory location. The result is that with each iteration of the loop, the pause instruction will display a value on the LEDs one greater than the value in the last iteration.”

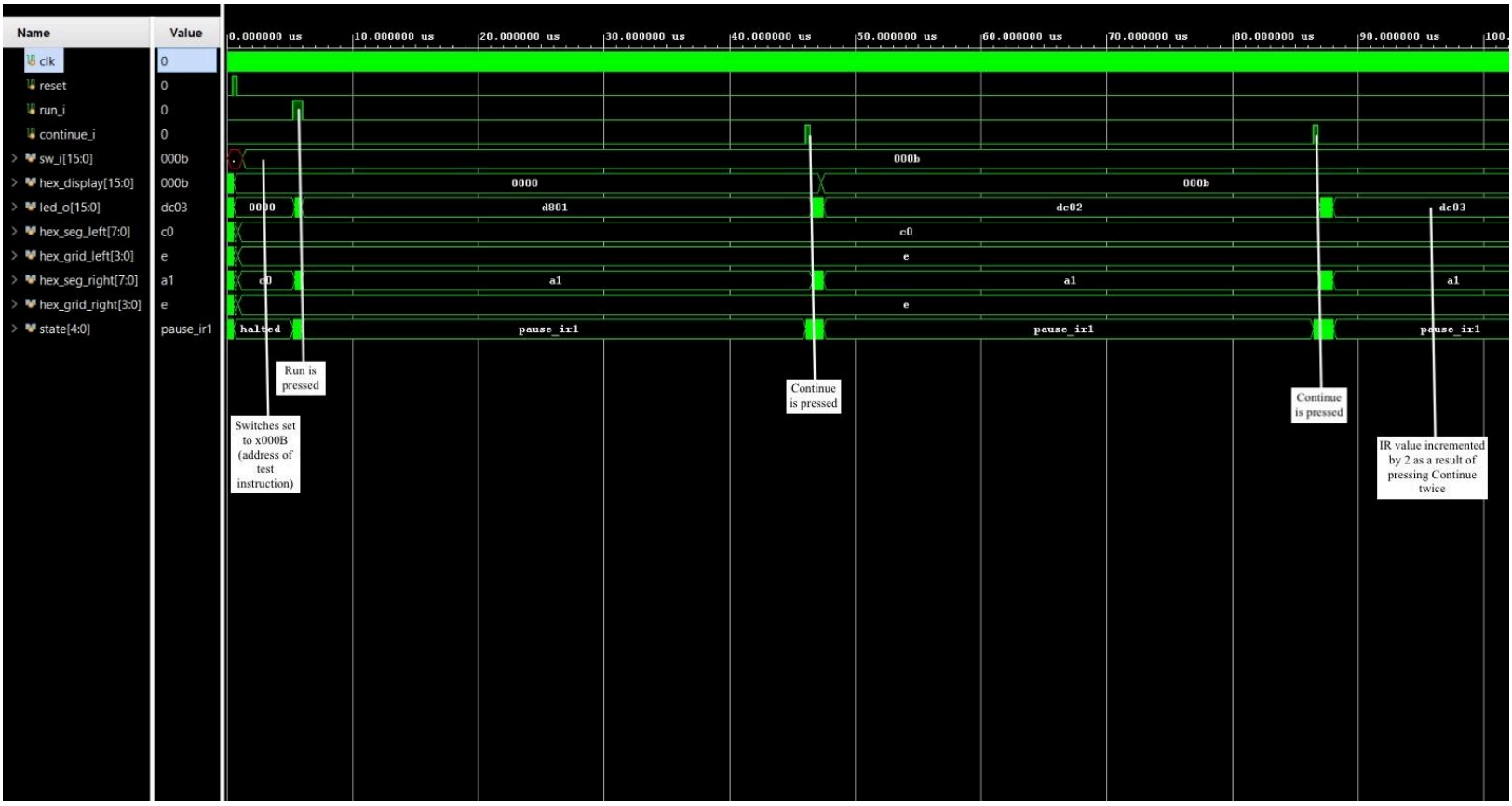


Figure 23. Simulation for Self-Modifying Code Test

Auto Counting Test

Figure 24 shows the simulation for the Auto Counting Test.

From the Test Programs Documentation, “This program will display an automatically incrementing counter to the hex displays without the need of inputs after jumping to the program. It loads preset constants into R1 and R2 using LDR which serve as counting variables in a double for loop to create delay between the counter increments. After each loop iteration, the respective counter variable is decremented using the ADDi instruction (ADD R ← R - 1). To traverse between the for loops, the BR instruction is used.”

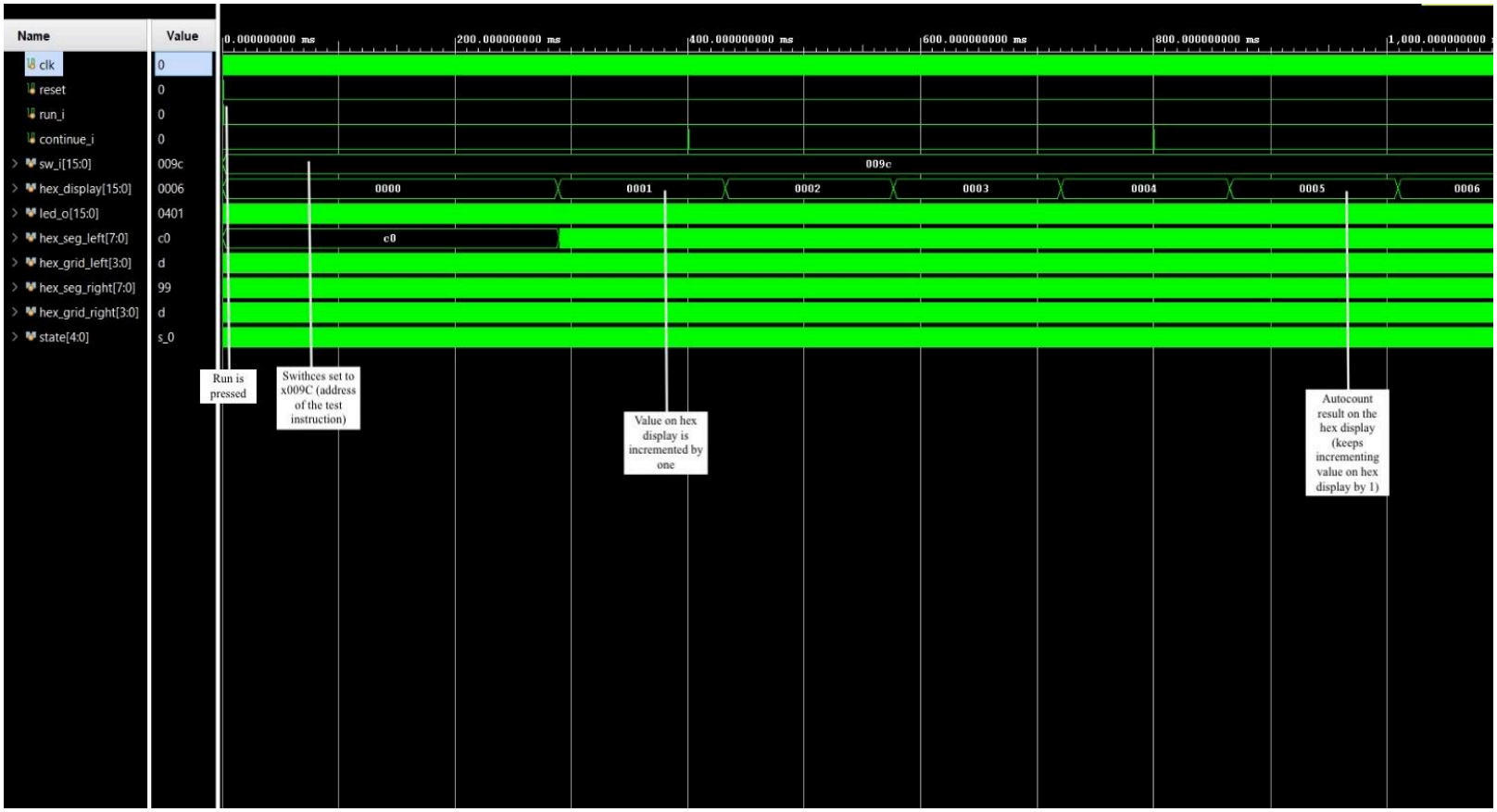


Figure 24. Simulation for Auto Counting Test

XOR Test

Figure 25 shows the simulation for the XOR Test.

From the Test Programs Documentation, “This program performs XOR operation. In sLC-3, there is no dedicated XOR instruction, so the XOR is performed by multiple simple instructions (AND and NOT). The program will ask for input values, XOR them, and display the result on the hex display.”

For our simulation, we loaded x0003 followed by x0004 to perform the XOR operation. The final result of x0007 was displayed on the hex displays.

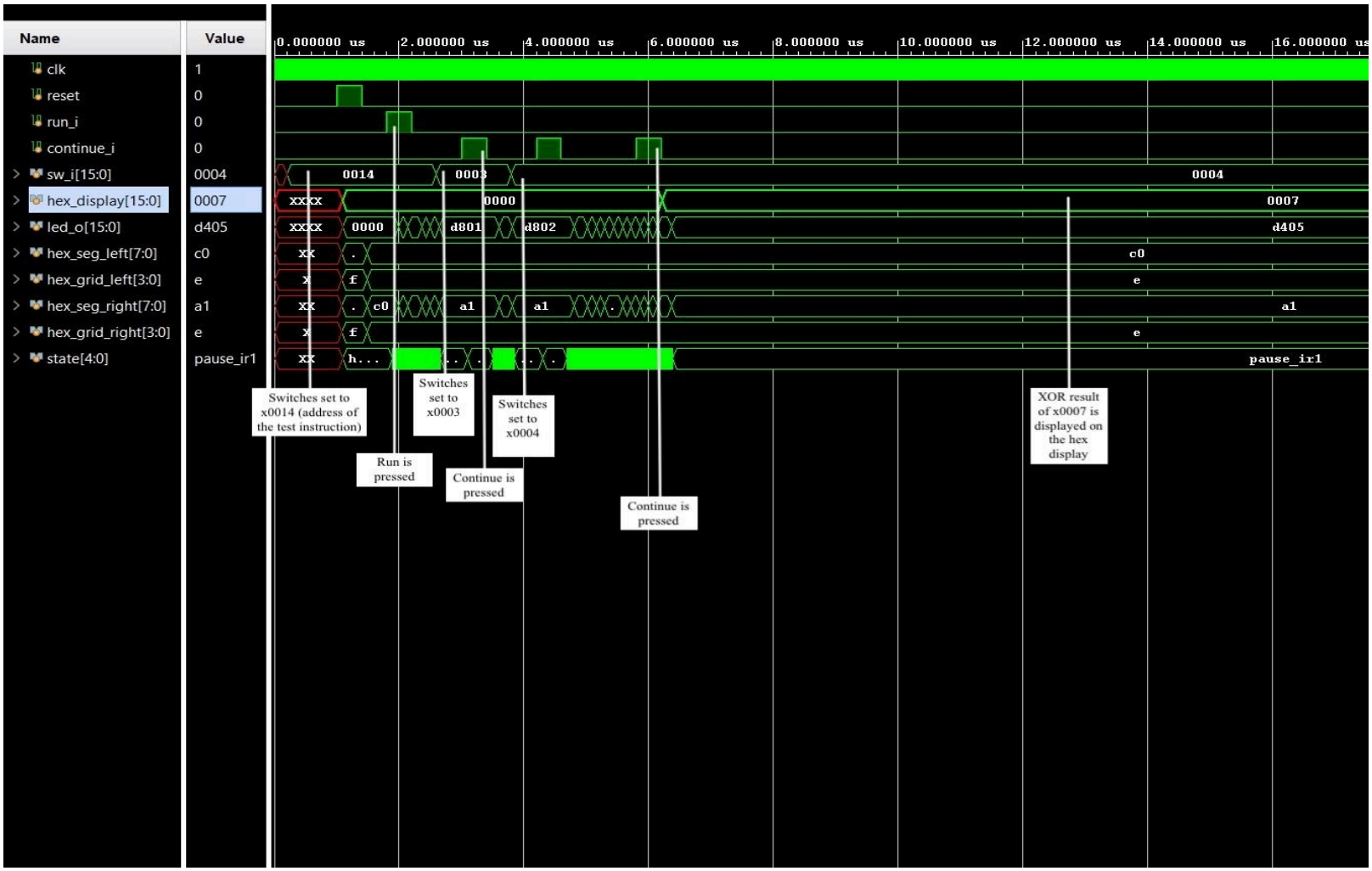


Figure 25. Simulation for XOR Test (x0003 XOR x0004 = x0007)

Multiplication Test

Figure 26 shows the simulation for the Multiplication Test.

From the Test Programs Documentation, “This program performs multiplication, using a variation on the shift-and-add algorithm (using ADD Rx, Rx, Rx as a left-shift operation). The program will ask for input values, multiply them, and display the result on the hex display. It will then loop back to the top and repeat the process. Note multiplication is performed on unsigned values, so negative multiplications are not supported.”

For our simulation, we loaded x0003 followed by x0006 to perform the multiplication. The final result of x0012 was displayed on the hex displays.

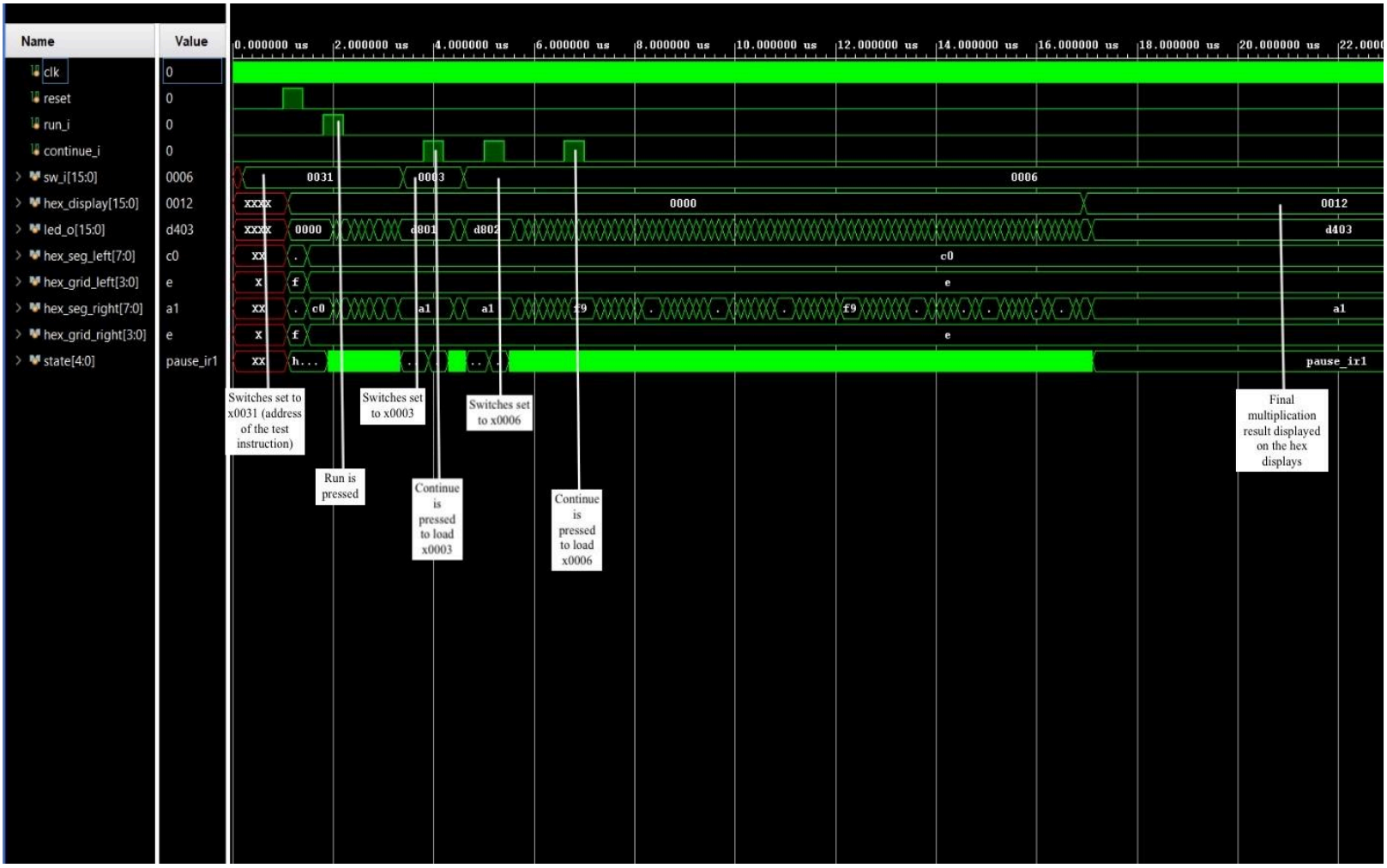


*Figure 26. Simulation for Multiplication Test (x0003 * x0006 = x0012)*

Sort Test

Figure 27 shows the simulation for the Sort Test.

From the Test Programs Documentation, “This program is organized into four parts. The first part is a menu, containing function calls to the other three parts that are executed based on input. The menu contains a single pause instruction, checkpoint 1 (displays xFF on the LEDs). Entering x0001 will call the “data entry” function, entering x0002 will call the “sort” function, and entering x0003 will call the “display” function. Any other value will simply cause the menu to loop back to the start without doing anything. The sort function will sort the values in the list using the Bubble Sort algorithm. No feedback is given to the user about the completion of the algorithm; it will return (seemingly) immediately to the menu.”

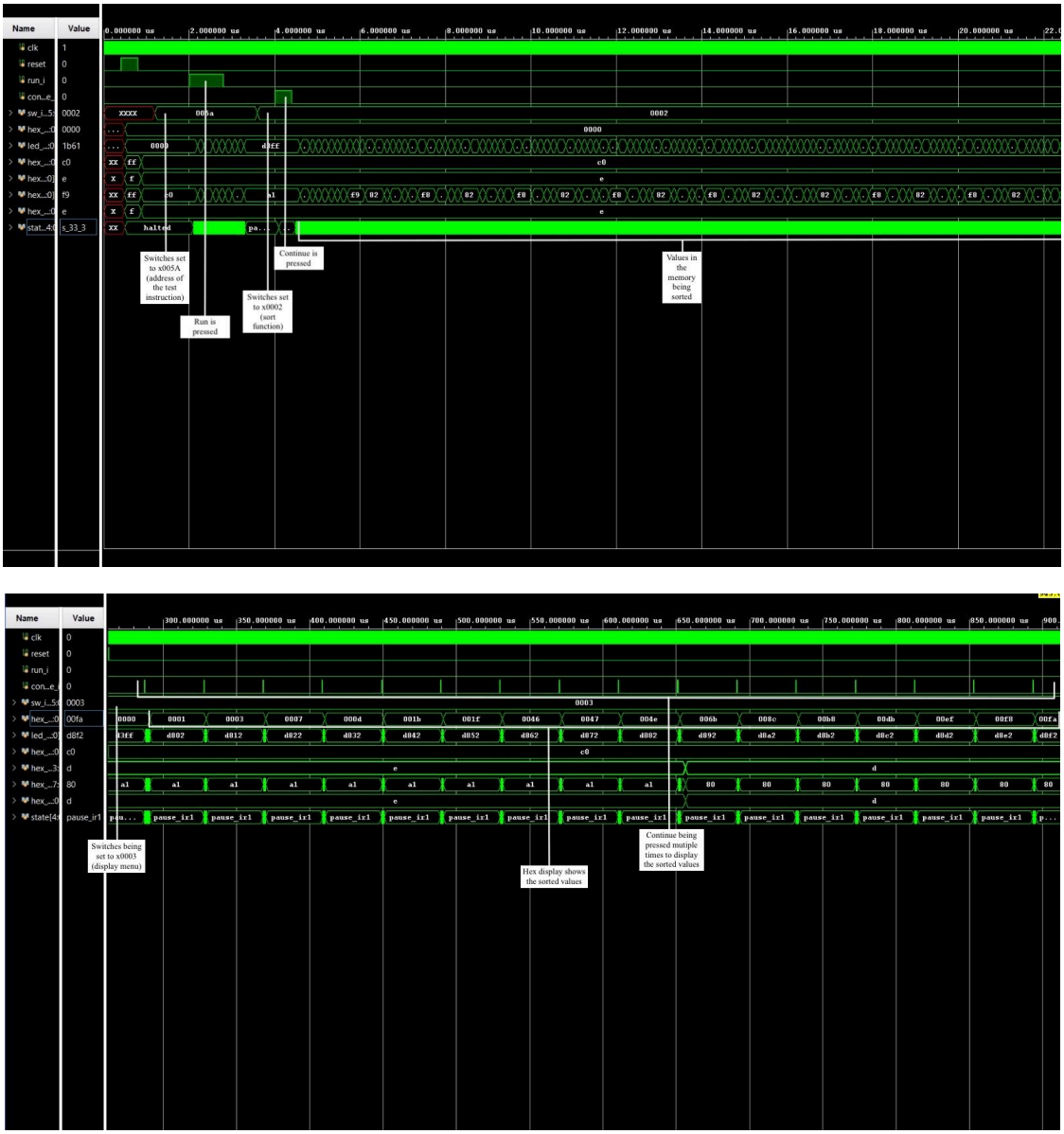


Figure 27. Simulation for Sort Test

“Act Once” Test

Figure 28 shows the simulation for the “Act Once” Test.

From the Test Programs Documentation, “This program is a simple loop that counts up from zero, once per iteration, and displays the count on the hex display. When operating correctly, the counter will increase by one (no more, no less) with each press of Continue (i.e., the CPU will act once per press.) This test need only be run if the “act once” functionality of the run or continue buttons is called into question in the course of running other tests.”

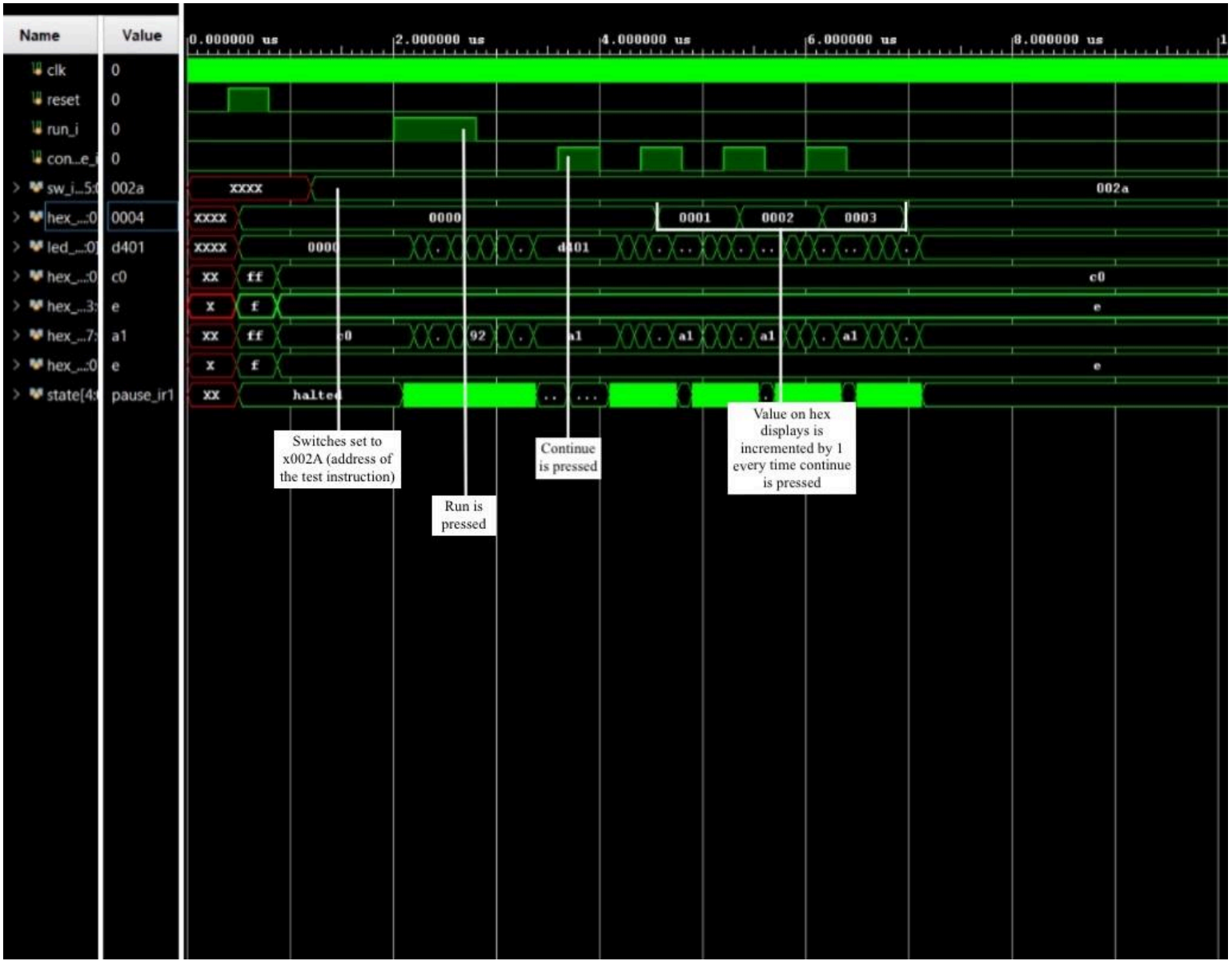


Figure 28. Simulation for “Act Once” Test

Post Lab

1. *Table 1* shows the design resources and statistics for our implementation of the SLC-3 processor.

Table 1. Design Resources and Statistics

LUT	474
DSP	0
Memory (BRAM)	0.5
Flip-Flop	351
Latches	0
Frequency	100.61 MHz
Static Power	0.074 W
Dynamic Power	0.013 W
Total Power	0.087 W

2. *What is CPU_TO_IO used for, ie, what is its main function?*

The cpu_to_io module acts as a bridge between the CPU, on-chip memory (BRAM), and the I/O devices. It connects the CPU to the BRAM so that the CPU can read and write data from and into it. It also handles the special I/O operations when the CPU accesses the memory address 0xFFFF. If the CPU reads from address 0xFFFF, it gets the switch values (`sw_i`). If the CPU writes to address 0xFFFF, it updates the hex display (`hex_seg_o`, `hex_grid_o`). For all other addresses, the module simply passes the signals to BRAM for normal memory access.

What is the difference between BR and JMP instructions?

Both the BR (Branch) and JMP (Jump) instructions are used to change the value of the program counter (PC), but they do so in different ways. The BR instruction performs a conditional branch based on the condition codes, negative (N), zero (Z), or positive (P). During execution, the control unit uses the BEN (Branch Enable) signal to decide whether the branch should be taken. If BEN = 1, then the PC is updated by adding the sign-extended 9-bit offset (IR[8:0]) to its current value. If BEN = 0, the PC remains unchanged, and the operation moves back to the fetch phase. On the other hand, the JMP instruction performs an unconditional jump, meaning that it always updates the PC, regardless of the condition codes. The new PC value is taken directly from the base register specified by bits IR[8:6].

What is the purpose of the R signal in Patt and Patel? How do we compensate for the lack of the signal in our design? What impact does that have on performance?

In LC-3, the R (Ready) signal is used by the memory to indicate that a read or write operation has completed and that valid data is available. The processor waits for this signal before proceeding, allowing it to synchronize with memory devices that may have variable access times. It basically indicates that a read/write operation is ready.

In the SLC-3 design, the FPGA's on-chip memory (BRAM) works with fixed timing and does not provide an R signal. To compensate, the processor's control unit is designed to remain in the memory read or write states for several clock cycles, ensuring that memory operations complete before moving on.

This approach simplifies the control logic and guarantees correct memory synchronization. But at the same time, it reduces the performance slightly since the CPU always waits a fixed number of cycles, even when the memory could respond more quickly.

Conclusion

In this lab, we successfully designed and implemented the SLC-3, a simplified RISC-based microprocessor inspired by the LC-3 architecture. It follows the fetch-decode-execute cycle and uses memory-mapped I/O to interact with switches and hex displays. We created several modules like the CPU, memory, control unit, and I/O bridge, and tested them using different simulation programs such as input/output, multiplication, and sorting tests.

While working on this lab, we faced a few problems. One issue was that we forgot to update the sr1_sel signal during the LDR instruction, which caused wrong results. Another problem was with our testbench for the Basic I/O Test 2, we didn't set enough clock cycles after pressing the continue button, so we thought the code was wrong when it wasn't.

Overall, the lab helped us understand how a simple processor works and taught us how important debugging and careful testing are during hardware design.