# ECE 385

Spring 2024

Experiment # 6

# Lab6 : Simple Computer SLC-3.2

# in System Verilog

Name:Jie Wang, Shitian Yang

Student ID: 3200112404, 3200112415

Prof. Chushan Li, Prof. Zuofu Cheng

ZJU-UIUC Institute

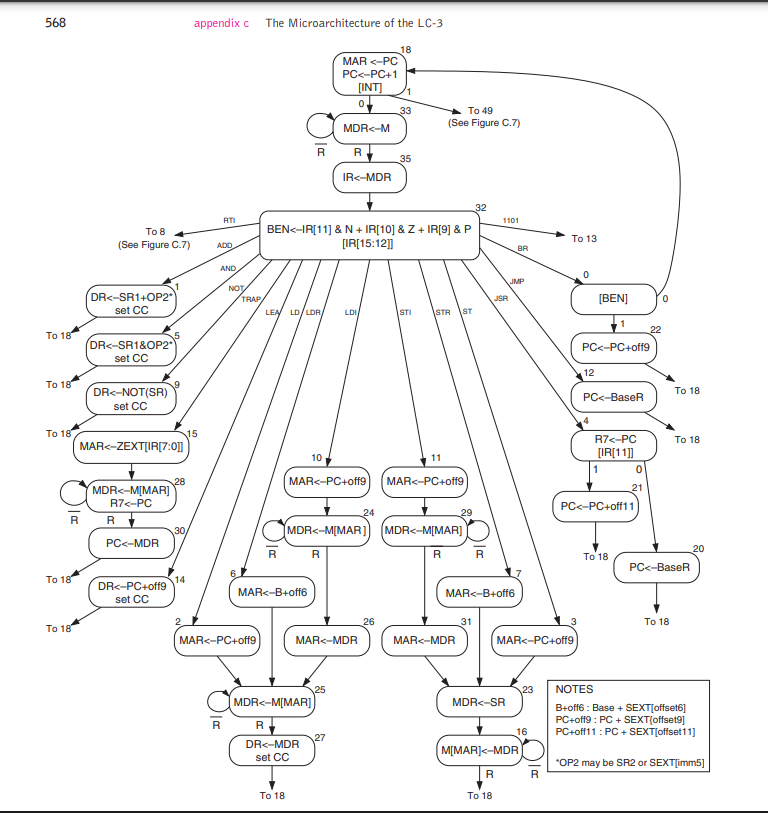
March 22, 2024, Friday D-225

TA: Jiebang Xia

**Demo Point: 10/10**

## 1.Introduction

Let time goes back to the golden time of ECE120/220, we meet the ***Little Computer 3 Assembly(LC-3)*** once more in the ECE385. This lab aimed to design a simple microprocessor using SystemVerilog, implementing a subset of the LC-3 Instruction Set Architecture (ISA). The processor is 16-bit, with a 16-bit Program Counter (PC), 16-bit instructions, and 16-bit registers. The design process involved understanding the central processing unit (CPU), memory storage, and input/output interface.



**Fig-1:**Holy State Diagram of the Whole LC-3, displayed here for fun.

Luckily, we just need to write the simplified version of the LC-3. Operating on a Fetch-Decode-Execute cycle, it includes components such as the Program Counter(**PC**), Instruction Register (**IR**), Memory Address Register (**MAR**), Memory Data Register (**MDR**), Instruction Sequencer/Decoder, status register (**nzp**), a general-purpose register file, and an Arithmetic Logic Unit (**ALU**).

## 2. SLC-3 System Design

The SLC-3 processor's design includes several key modules:

* **CPU Core**: Contains the PC, IR, MAR, MDR, ALU, and register file. This is realized by file: *‘slc3.sv ’, ‘MUX.sv’, ‘datapath.sv’, ‘register.sv’,*
* **Instruction Sequencer/Decoder (ISDU)**: Generates control signals for the processor's operations. This is realized by *‘ISDU.sv’,*
* **Memory Interface**: Facilitates communication between the CPU and memory. This is realized by*, ‘Mem2IO.sv’,* ‘*tristate.sv’*
* **I/O Interface (Mem2IO)**: Manages I/O with physical devices like switches and displays. This is realized by *‘HexDriver.sv’,*
* **Supporting Modules:** Contains constants and helper functions for instruction encoding, used in testbenches and initializations. This is realized by*: ‘SLC3\_2.sv’* ,*‘sys\_clock.sdc’, ‘Synchronizers.sv’.* They do not belong to a specific hardware module but serves as a shared resource for defining and interpreting the instruction set of SLC-3.
* **Test program**:
* *‘memory\_contents.sv’,* the LC-3 Code before compilation, we just need to read it.
* *‘Test\_Programs\_image.ram’,* the compiled LC-3 image, need to be loaded into RAM.
* *testbench\_week1.sv & testbench\_week2.sv*: simulation wave test program for two weeks checkpoints separately.

图示, 示意图

描述已自动生成

**Fig-2:** The circuit diagram of our SLC-3

T**he written description of all \*.sv modules are recorded in Appendix section A.**

### Summary of Operation

The SLC-3 operates on a basic Fetch-Decode-Execute cycle. This cycle includes fetching an instruction from memory( test memory in simulation and hardware memory after IO Mapping), decoding it to understand what actions to perform, and then executing those actions, which involve arithmetic operations, memory access, and control operations.

### SLC-3 Functionality Mechanism

**Fetch:** The Memory Address Register (MAR) is loaded with the value of the Program Counter (PC), indicating the memory address to read the next instruction from. The Memory Data Register (MDR) is then loaded with the instruction fetched from memory at the address specified in the MAR. The Instruction Register (IR) receives the instruction from the MDR, and the PC is incremented to point to the next instruction.

**Decode:** The fetched instruction in the IR is decoded by the Instruction Sequencer/Decoder, which interprets the opcode and specifies the operation to perform. This may involve identifying the type of instruction (e.g., arithmetic, memory access, control) and the operands involved.

**Execute:** Based on the decoded instruction, the SLC-3 performs the specified operation. This could involve arithmetic computations in the ALU, loading values from or storing values to memory, or altering the flow of execution based on branch instructions.

### Block Diagram of slc3.sv

Here is the rendered RTL view of our SLC-3 CPU core. It is highly complex but works well, passing all the test cases with 10/10 grade in the demo.

图示, 示意图

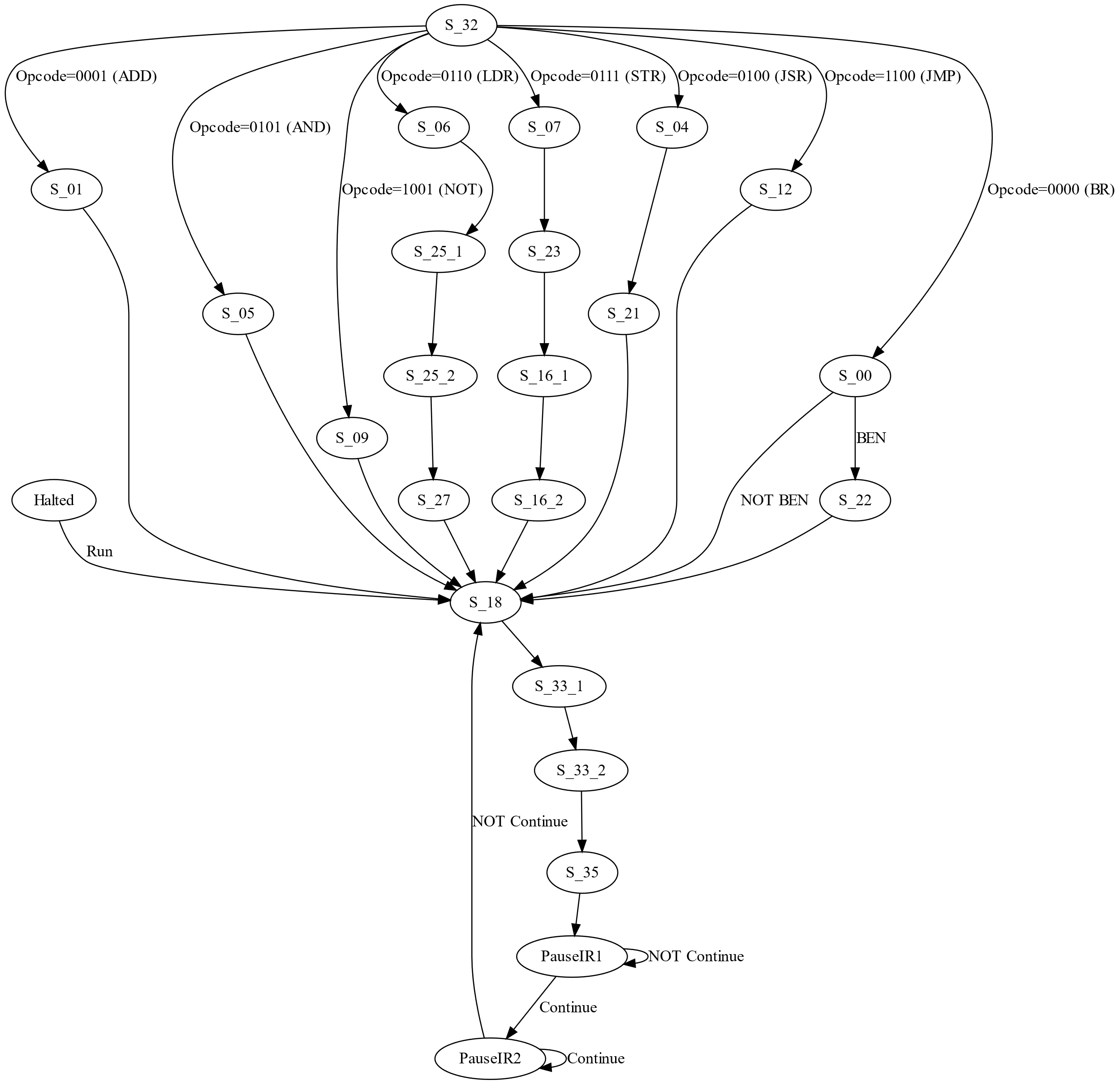
描述已自动生成

**Fig-3: Our SLC-3 Block Diagram**

### Operation of the ISDU

**You can view this section in Appendix Section B.**

### State Diagram of ISDU



**Fig-4:** FSM of our ISDV modulo, drawn using Graphviz Tool

Above FSM demonstrates the mechanism we took through the Instruction flow.

## 3.Simulations of SLC-3 Instructions

#### A. Simulate the completion of 3 instructions from the following groups:

ADD/ADDi/AND/ANDi/NOT; BR/JMP/JSR; LDR/STR.

电视游戏的萤幕截图

描述已自动生成

#### Simulation using TestBench\_Week2.sv with Annotation

We comment other unit test cases, leaving only the "Act Once" Test:

电脑萤幕截图

描述已自动生成

**Fig-4: Passed all the testcase in testbench\_week2.sv**

## 4.Post-lab Question

#### a. Design Resources and Statistics table

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Feature** | **LUT** | **DSP** | **Memory**  **(BRAM)** | **Flip-Flop** | **Frequency** | **Static power** | **Dynamic Power** | **Total Power** |
| **Data** | 737 | 0 | 0 | 286 | 67.01MHz | 96.33mW | 6.9mW | 158.23mW |

The data of our system is computed and recorded above.

#### b. Answer Question

##### 1. What is the function of the MEM2IO.sv module?

The MEM2IO module serves as a bridge program between the SRAM module and the SLC3 processor within the FPGA DE2 Board.

This module primarily enables data transactions from input devices, such as switches, to the processor memory and to allow the processor to output data to display units like the HEX display.

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

The JMP (Jump) instruction directs the program control to a specified location using the address stored in the Base Register, without evaluating any condition codes. In contrast, the BR (Branch) instruction conditions the jump on the match between the condition codes and the status register's contents. If a match is found, it branches to a location computed by adding a sign-extended offset (PCoffset9) to the current PC; if not, it proceeds with the sequential execution.

***Key Differences:***

* JMP performs an unconditional jump, while BR requires a conditional evaluation.
* JMP sources the jump address from a register, whereas BR derives the jump address from the immediate value (offset) in IR.

##### 3. Purpose of the R Signal in Textbook, and the Design Compensation:

In Patt and Patel's architecture, the R signal indicates the readiness of memory for data transactions. In our design, where the R signal is absent, we introduce an intermediary state, thereby expanding state 33 into two sub-states: 33-1 and 33-2. State 33-1 introduces a delay cycle to emulate the memory's readiness time, and state 33-2 proceeds with the signal operations as originally designed in state 33. This modification eliminates the need for the R signal by ensuring that memory operations are delayed by one clock cycle, thus maintaining synchronization within the design without an additional synchronizer.

## 5. Conclusion

Our SLC-3.2 implementation in SystemVerilog is functionally successful. It refreshes our memory of CPU operations as taught in ECE 120/220. Key modules such as the CPU core, ISDU, memory interface, and I/O interface performed well, passing all test cases.

Challenges like the MDR-IR loading bug were instructive, emphasizing the iterative nature of digital design. The lab manual is clear, though more examples for complex sections could help understanding. The structured approach and inclusion of testbenches are practices that should be maintained.

Overall, the lab solidified our understanding of microprocessor architecture and prepared us for more complex design challenges.

## 6. References

[1] KTTECH. (2017, February 17). ECE 385 Lab 6: sLC3. Retrieved from <https://kttechnology.wordpress.com/2017/02/17/ece-385lab-6-slc3/> , Teaching Assistant Blog

[2] ECE385 Faculty. (n.d.). [Lab 6 description](https://learn.intl.zju.edu.cn/bbcswebdav/pid-101276-dt-content-rid-1361038_1/xid-1361038_1)

[3] ECE385 Faculty. (n.d.). [Introduction to SystemVerilog (pdf)](https://learn.intl.zju.edu.cn/bbcswebdav/pid-101280-dt-content-rid-1361044_1/xid-1361044_1)

[4] ECE385 Faculty. (n.d.). [Introduction to Quartus Prime in the lab manual.](https://learn.intl.zju.edu.cn/bbcswebdav/pid-101280-dt-content-rid-1361046_1/xid-1361046_1)

## 7. Appendix

### Written Description of all .sv modules

**Module Name:**

datapath

**Inputs:**

logic ADDR1MUX, SR2MUX, MIO\_EN, SR1MUX, DRMUX

logic GateMARMUX, GateMDR, GateALU, GatePC, Clk, Reset

logic LD\_REG, LD\_BEN, LD\_CC, LD\_IR, LD\_MAR, LD\_MDR, LD\_PC, LD\_LED

logic [1:0] PCMUX, ADDR2MUX, ALUK

logic [15:0] MDR\_in

**Outputs:**

logic BEN

logic [15:0] MDR\_OUT, MAR\_OUT, IR\_OUT, PC\_OUT

logic [11:0] LED

**Description:**

The datapath module represents a crucial component of a digital system, typically a microprocessor or microcontroller, that orchestrates the flow of data between various registers, arithmetic logic units (ALU), and memory components.

**Operation:**

Control Signals (ADDR1MUX, SR2MUX, etc.): Determine the behavior of the datapath by controlling multiplexers, enabling specific gates, and loading values into registers or memory.

PCMUX, ADDR2MUX, ALUK: Select the source for the Program Counter (PC) updates, address calculations, and the type of arithmetic or logical operation to perform in the ALU, respectively.

MDR\_in: Serves as the input data for memory data operations, potentially coming from an external source or memory unit.

Data Flow Control: The GateXXX inputs control the flow of data within the datapath, enabling the output from the MAR (Memory Address Register), MDR (Memory Data Register), ALU, and PC to be routed correctly for either further processing or external output.

Load Controls (LD\_REG, LD\_BEN, etc.): Specify when to update the internal state of the datapath, including registers and condition flags, based on the execution of the current instruction.

Outputs (BEN, MDR\_OUT, MAR\_OUT, IR\_OUT, PC\_OUT, LED): Provide the status of branch conditions (BEN), the current state of key registers and memory addresses, and allow for external visibility of internal states or conditions through LEDs.

**Module Name:**

ALU

**Inputs:**

logic [15:0] A, B: Two 16-bit input operands for arithmetic and logical operations.

logic [1:0] ALUK: A 2-bit control signal that determines the operation to be performed on the inputs.

**Output:**

logic [15:0] ALU\_OUT: The 16-bit result of the operation performed by the ALU.

**Description:**

The ALU (Arithmetic Logic Unit) module performs a variety of arithmetic and logical operations based on the control signal ALUK. It takes two 16-bit inputs A and B and outputs a 16-bit result ALU\_OUT. The specific operation performed (addition, bitwise AND, bitwise NOT on A, or passing A through) depends on the 2-bit ALUK signal.

**Operation:**

2'b00: Addition - The ALU\_OUT is the sum of A and B.

2'b01: Bitwise AND - The ALU\_OUT is the result of a bitwise AND operation between A and B.

2'b10: Bitwise NOT - The ALU\_OUT is the result of a bitwise NOT operation on A (ignoring B).

2'b11: Pass-through - The ALU\_OUT is simply the value of A.

**Module Name:**

BEN

**Inputs:**

logic [15:0] data\_in: 16-bit input data for condition code flag evaluation.

logic LD\_CC, LD\_BEN, clk, Reset: Control signals for loading condition codes (LD\_CC), updating the branch enable (LD\_BEN), the clock signal (clk), and reset (Reset).

logic [2:0] IR\_in: Instruction register input, used to determine branch conditions based on the current instruction.

**Output:**

logic BEN\_out: The branch enable output signal, indicating whether a branch condition has been met.

**Description:**

The BEN module evaluates branch conditions based on the condition codes (negative, zero, positive) and the instruction register's relevant bits (IR\_in). It updates its output, BEN\_out, indicating whether a branch should be taken based on the alignment between the condition codes and the instruction's requirements for branching. The condition codes are updated based on the data\_in input whenever LD\_CC is asserted, and the branch enable signal is updated when LD\_BEN is asserted.

**Operation:**

Condition Code Calculation: On every combinational path evaluation, the module calculates the negative (n), zero (z), and positive (p) flags based on data\_in. If data\_in is negative (data\_in[15] is 1), n is set. If data\_in is zero, z is set. If data\_in is positive (not zero and data\_in[15] is 0), p is set.

**Module Name:**

gates\_control

**Inputs:**

logic [3:0] select: A 4-bit input used to select one of the input data lines (d0, d1, d2, d3) for output.

logic [15:0] d0, d1, d2, d3: Four 16-bit data inputs that can be selected for output based on the select signal.

**Output:**

logic[15:0] out: The selected 16-bit data output based on the select input.

**Description:**

The gates\_control module acts as a 4-to-1 multiplexer, selecting one of its four 16-bit data inputs (d0, d1, d2, d3) to pass to the output, based on the 4-bit select signal. Each bit in the select signal corresponds to one of the data inputs, and the module outputs the data input corresponding to the bit that is set to 1. If multiple bits in select are set or if select is 0, the default output is 0.

**Operation:**

When select is 4'b0001, the output (out) is d0.

When select is 4'b0010, the output is d1.

When select is 4'b0100, the output is d2.

When select is 4'b1000, the output is d3.

**Module Name:**

mux\_2\_to\_1\_16bits

**Inputs:**

logic select: A single-bit selection signal.

logic [15:0] d0, d1: Two 16-bit data inputs.

**Output:**

logic [15:0] out: The selected 16-bit data output.

**Operation:**

Based on the select signal, this module outputs either d0 or d1. If select is 0, out is d0; if select is 1, out is d1.

**Module Name:**

mux\_2\_to\_1\_4bits

**Inputs:**

logic select: A single-bit selection signal.

logic [3:0] d0, d1: Two 4-bit data inputs.

**Output:**

logic [3:0] out: The selected 4-bit data output.

**Operation:**

Operates similarly to the 16-bit version but with 4-bit data inputs and output.

**Module Name:**

mux\_2\_to\_1\_3bits

**Inputs:**

logic select: A single-bit selection signal.

logic [2:0] d0, d1: Two 3-bit data inputs.

**Output:**

logic [2:0] out: The selected 3-bit data output.

**Operation:**

Operates similarly to the 16-bit version but with 3-bit data inputs and output.

**Module Name:**

mux\_4\_to\_1\_16bits

**Inputs:**

logic [1:0] select: A 2-bit selection signal.

logic [15:0] d0, d1, d2, d3: Four 16-bit data inputs.

**Output:**

logic [15:0] out: The selected 16-bit data output.

**Operation:**

This module selects one of the four 16-bit data inputs based on the 2-bit select signal and outputs the selected data. The select value of 00, 01, 10, or 11 selects d0, d1, d2, or d3, respectively.

**Module Name:**

reg\_16bit

Inputs:

logic clk: Clock signal.

logic reset: Reset signal, active high.

logic load: Load signal, active high.

logic [15:0] din: 16-bit data input.

**Output:**

logic [15:0] dout: 16-bit data output.

**Description:**

The reg\_16bit module is a 16-bit register that stores or updates its value based on the input signals. On the rising edge of the clock (clk), if the reset signal is active (high), the register's output (dout) is cleared (set to 0). If reset is not active but the load signal is active, the value from din (data input) is loaded into the register, updating its output to match din. If neither reset nor load is active, the register maintains its current value.

**Operation:**

Reset: When the reset signal is high at the clock's rising edge, the register clears its output to zero (16'h0000), disregarding the load signal and din value.

Load: If reset is low and load is high at the clock's rising edge, the register updates its output to the value present on din at that clock edge.

Hold: If both reset and load are low, the output remains unchanged, effectively holding the last value.

**Module Name:**

REG\_FILE

**Inputs:**

logic CLK: Clock signal.

logic LDREG: Load Register signal, active high.

logic Reset: Reset signal, active high.

logic [15:0] bus: Data bus used for input to and output from the register file.

logic [2:0] DR: Destination Register address for write operations.

logic [2:0] SR1, SR2: Source Register addresses for read operations.

**Outputs:**

logic [15:0] SR1\_OUT, SR2\_OUT: Outputs of the Source Registers specified by SR1 and SR2.

**Description:**

The REG\_FILE module represents a simple register file with 8 registers, each 16 bits wide. It supports reading from two source registers (SR1 and SR2) simultaneously and writing to a destination register (DR) under control of the LDREG signal. The module resets all registers to 0 upon an active Reset signal. Data transfer to a selected register is performed through the bus, allowing for versatile data handling within a larger system such as a CPU.

**Operation:**

Reset: On the rising edge of CLK, if Reset is active, all 8 registers in reg\_memory are cleared to 16'h0000.

Load Data: If Reset is inactive but LDREG is active, the module writes the data from the bus into the register specified by the DR address. This operation is also synchronized with the rising edge of CLK.

Read Data: Independent of the clock and not subject to synchronization, the module continuously outputs the contents of the registers specified by SR1 and SR2 addresses on SR1\_OUT and SR2\_OUT, respectively.

### Description of the operation of the ISDU (Instruction Sequence Decoder Unit)

**Module Name:**

ISDU

**Inputs:**

logic Clk: Clock signal.

logic Reset: Asynchronous reset signal.

logic Run: Signal to start the operation.

logic Continue: Signal to continue operation from a paused state.

logic [3:0] Opcode: Operation code determining the type of instruction.

logic IR\_5, IR\_11: Specific bits of the Instruction Register that may alter the behavior of certain instructions.

logic BEN: Branch Enable signal, indicating whether branch conditions have been met.

**Outputs:**

logic LD\_MAR, LD\_MDR, LD\_IR, LD\_BEN, LD\_CC, LD\_REG, LD\_PC, LD\_LED: Control signals for loading values into various registers and LEDs (for pause instruction).

logic GatePC, GateMDR, GateALU, GateMARMUX: Control signals for gating the outputs of the PC, MDR, ALU, and address mux to the bus.

logic [1:0] PCMUX: Control signal for selecting the source for the Program Counter update.

logic DRMUX, SR1MUX, SR2MUX, ADDR1MUX: Control signals for multiplexer selections influencing data path configurations.

logic [1:0] ADDR2MUX, ALUK: Control signals for selecting the ALU operation and addressing modes.

logic Mem\_CE, Mem\_UB, Mem\_LB, Mem\_OE, Mem\_WE: Memory control signals for chip enable, upper and lower byte enable, output enable, and write enable.

**Description:**

The ISDU module functions as the control center for the SLC-3 processor, interpreting the current instruction's opcode and other signals to issue a series of control signals that orchestrate the execution of instructions across the processor's data path. It uses a finite state machine (FSM) to transition through various states corresponding to different phases of instruction execution such as fetching, decoding, executing, and writing back results.

**Operation:**

Initialization: Upon a reset, the ISDU initializes into a halted state, awaiting a Run signal to start executing instructions.

**State Transitions:** Based on the Clk and Reset inputs, the ISDU transitions between states designed to fetch, decode, and execute instructions. State transitions are also influenced by Run and Continue signals for starting and pausing execution.

**Control Signal Generation:** In each state, the ISDU generates specific control signals that direct the flow of data within the processor. This includes loading values into registers (LD\_xxx signals), selecting data paths (MUX signals), and managing memory operations.

**Instruction Execution:** The execution phase is dependent on the opcode and possibly additional bits from the instruction register. The ISDU decodes this information and moves through specific states tailored to the instruction, such as arithmetic operations, memory access, or branch instructions.

**Branching and Continuation:** For branch instructions, the BEN signal, along with decoded instruction info, determines whether to update the program counter to a new address. The Continue signal is used to resume execution from a paused state, particularly useful for debugging or step-by-step execution.

**Memory Access:** Memory read/write operations are controlled through enabling the memory control signals and gating the appropriate paths to and from memory.