Phase One – 8-Bit ALU Design
Project: Simple CPU Design – Phase One (ALU Construction)

Overview
In this project I built an 8-bit Arithmetic Logic Unit (ALU) and a small CPU using Logisim-evolution.
The ALU takes two 8-bit inputs A and B and a 4-bit opcode. It produces an 8-bit result and several flags.
The CPU adds three 8-bit registers (REG_A, REG_B, REG_OUT), a shared datapath, and a tiny control unit.
At the end I wrote a small test program that shifts a value until a condition is met.

ALU DESIGN (ALU8 CIRCUIT)
Main inputs:

- A (8 bits) – first operand
- B (8 bits) – second operand
- OP (4 bits) – opcode used for testing

Main outputs:

- RESULT (8 bits) – ALU result
- Z – zero flag (1 when RESULT = 0)
- N – negative flag (copy of bit 7 of RESULT)
- C – carry / overflow flag
- GT, LT, EQ – comparison flags

Internal blocks:

- Adders for ADD, INC, DEC
- Subtractor for SUB
- Logic blocks using AND, OR, NOT, XOR, NAND
- Shifters for SHL and SHR
- Comparator for CMP and for GT, LT, EQ
- Constants: 0x00, 0x01, and 0xFF
- Splitters and tunnels to route the 8-bit buses

All block outputs go into a 16-to-1 multiplexer. The 4-bit opcode selects which block is connected to RESULT.

INSTRUCTIONS IMPLEMENTED

Required instructions:

1. ADD – A + B
2. INC – A + 1
3. DEC – A − 1
4. CMP – compare A and B, set GT, LT, EQ
5. NOT – bitwise NOT of A
6. AND – A AND B
7. OR – A OR B
8. SHR – shift A right by 1 bit
9. SHL – shift A left by 1 bit

Extra instructions I added:

10) SUB – A − B
11) XOR – A XOR B
12) NAND – NOT(A AND B)
13) ROL – rotate A left by 1 bit
14) MUL – A multiplied by B (low 8 bits)

OPCODE TABLE FOR THE ALU

The control unit uses a 4-bit opcode OP_3_0 (bits S3 S2 S1 S0) to choose the ALU operation.
The table below gives a **complete mapping** from opcodes to all 14 instructions.
Two codes are left unused/reserved.

(If the wiring in Logisim uses different codes, I only need to change the bit patterns here.)

Opcode (S3 S2 S1 S0) – Instruction – Description

0000 – ADD – A ← A + B
0001 – INC – A ← A + 1
0010 – DEC – A ← A − 1
0011 – SUB – A ← A − B
0100 – AND – A ← A AND B
0101 – OR – A ← A OR B
0110 – XOR – A ← A XOR B
0111 – NAND – A ← NOT(A AND B)
1000 – SHL – A ← A << 1
1001 – SHR – A ← A >> 1
1010 – ROL – A ← rotate-left(A, 1)
1011 – MUL – A ← (A × B) mod 256
1100 – CMP – set GT, LT, EQ based on A and B

1101 – unused (reserved)
1110 – unused (reserved)
1111 – unused (reserved)

Flag behavior:

- Z = 1 if RESULT = 0x00, else 0
- N = bit 7 of RESULT
- C = carry/borrow from the adder/subtractor (when used)
- For CMP, the main goal is to set GT, LT, EQ.

5. MAIN CPU CIRCUIT (MAIN)
   The main circuit adds three registers and the datapath.

Registers (8-bit each):

- REG_A – stores operand A
- REG_B – stores operand B
- REG_OUT – stores ALU result

Buses and tunnels:

- A_BUS – carries the value from REG_A (and other sources) to the ALU
- B_BUS – carries the value from REG_B (and other sources) to the ALU
- OUT_BUS – carries the selected ALU output back to REG_OUT and to the outer world

Control pins:

- IN_A (8 bits) – external input for REG_A
- IN_B (8 bits) – external input for REG_B
- LOAD_A – when 1, load IN_A into REG_A
- LOAD_B – when 1, load IN_B into REG_B
- LOAD_OUT – when 1, load OUT_BUS into REG_OUT
- OP_3_0 (4 bits) – opcode going to the ALU multiplexer
- OUT_7_0 (8 bits) – output connected to OUT_BUS

Flags coming out of the ALU are routed to pins LT, GT, EQ and to tunnels LT_FLAG, GT_FLAG, EQ_FLAG.

CONTROL UNIT
The control unit is built directly with gates and the multiplexer in the main circuit:

- The four bits S3, S2, S1, S0 form OP_3_0.

- These bits connect to:
    - the select inputs of the ALU multiplexer,
    - simple NOT, AND, OR blocks that create the load signals for REG_A, REG_B, REG_OUT when needed.

Example usage:

- To compute A ← A + B:
    - Set OP_3_0 = 0000 (ADD).
    - Make sure REG_A and REG_B already hold the correct values.
    - Assert LOAD_OUT so RESULT is stored in REG_OUT (and can be copied back into REG_A if desired).

SHIFT TEST PROGRAM

The final part of the assignment is a small test program that uses the CPU to perform repeated shifts.

Program idea:

- Start with A = 11111111 (0xFF).
- Repeatedly shift A right by one bit.
- After each shift, test the least significant bit.
- Stop when that bit becomes 0.

Pseudo-code:

```
A = 11111111b
 LOOP:
 use SHR on A ; shift right 1 bit
 if (A bit0 == 1) go back to LOOP
 ; stop when bit0 becomes 0
```

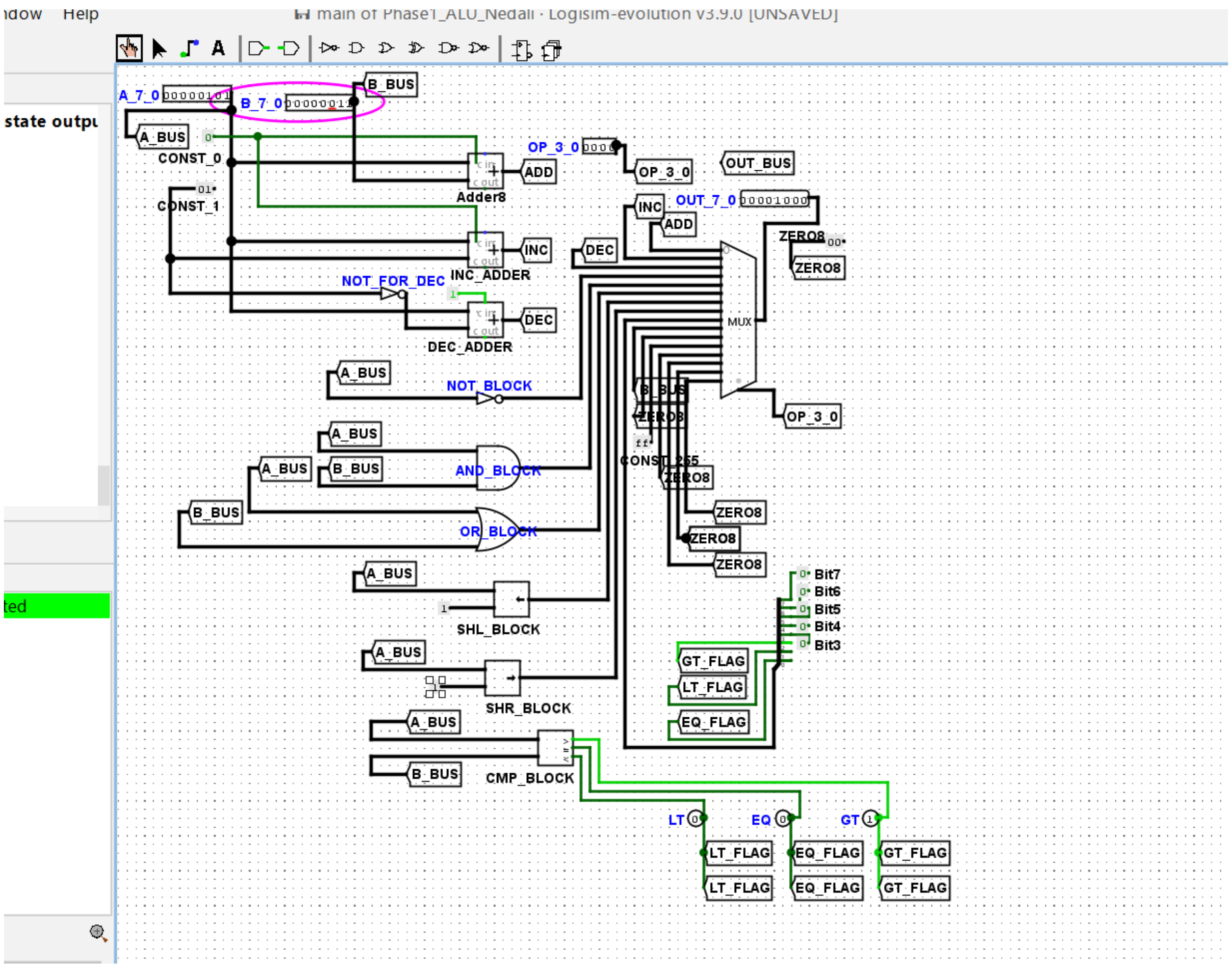
In Logisim, I use the SHR opcode, the comparison flags, and the registers to implement this behavior.

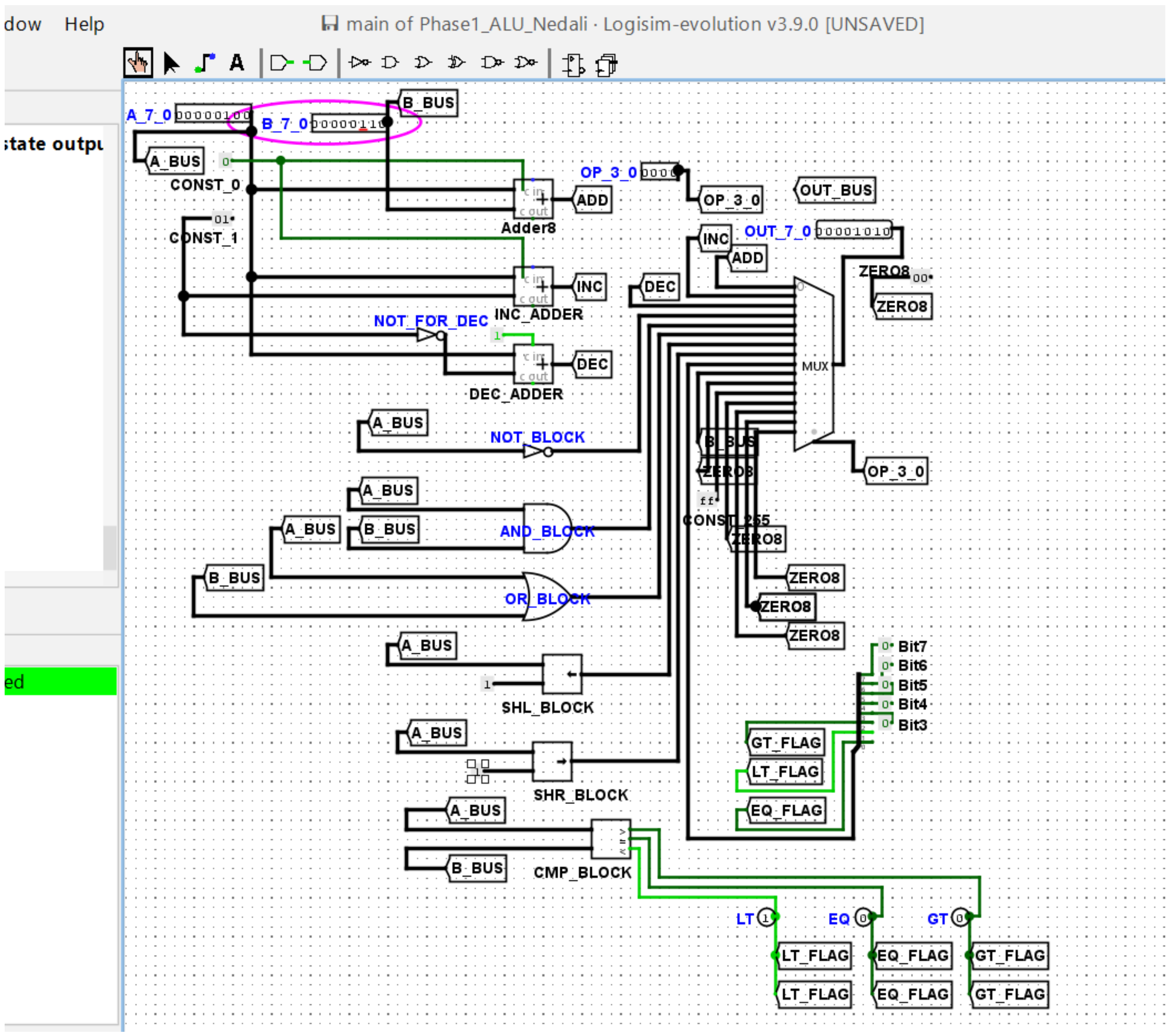
Watching the OUT_7_0 output lets me see the bits moving and the loop stopping at the correct time.

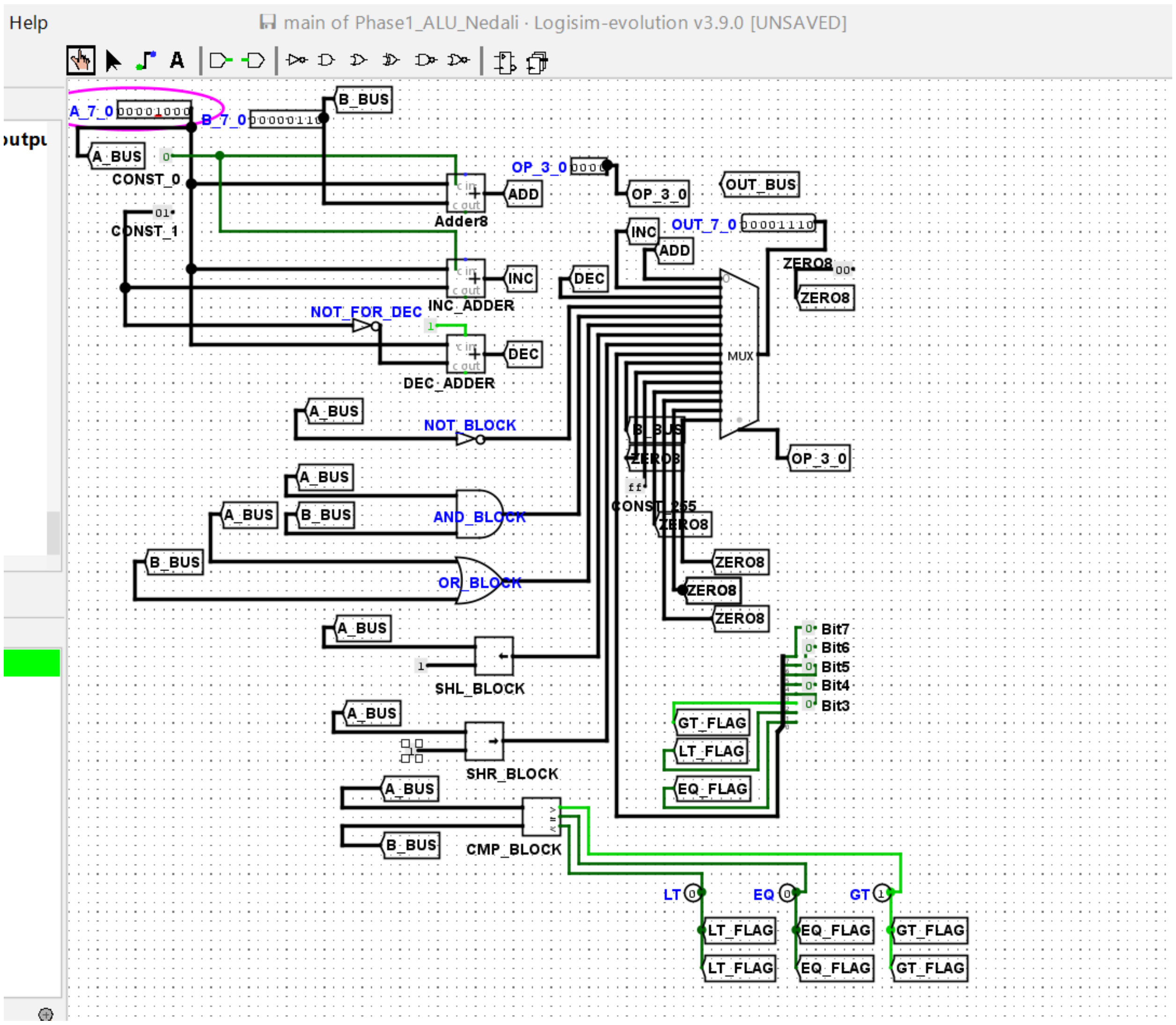
CONCLUSION

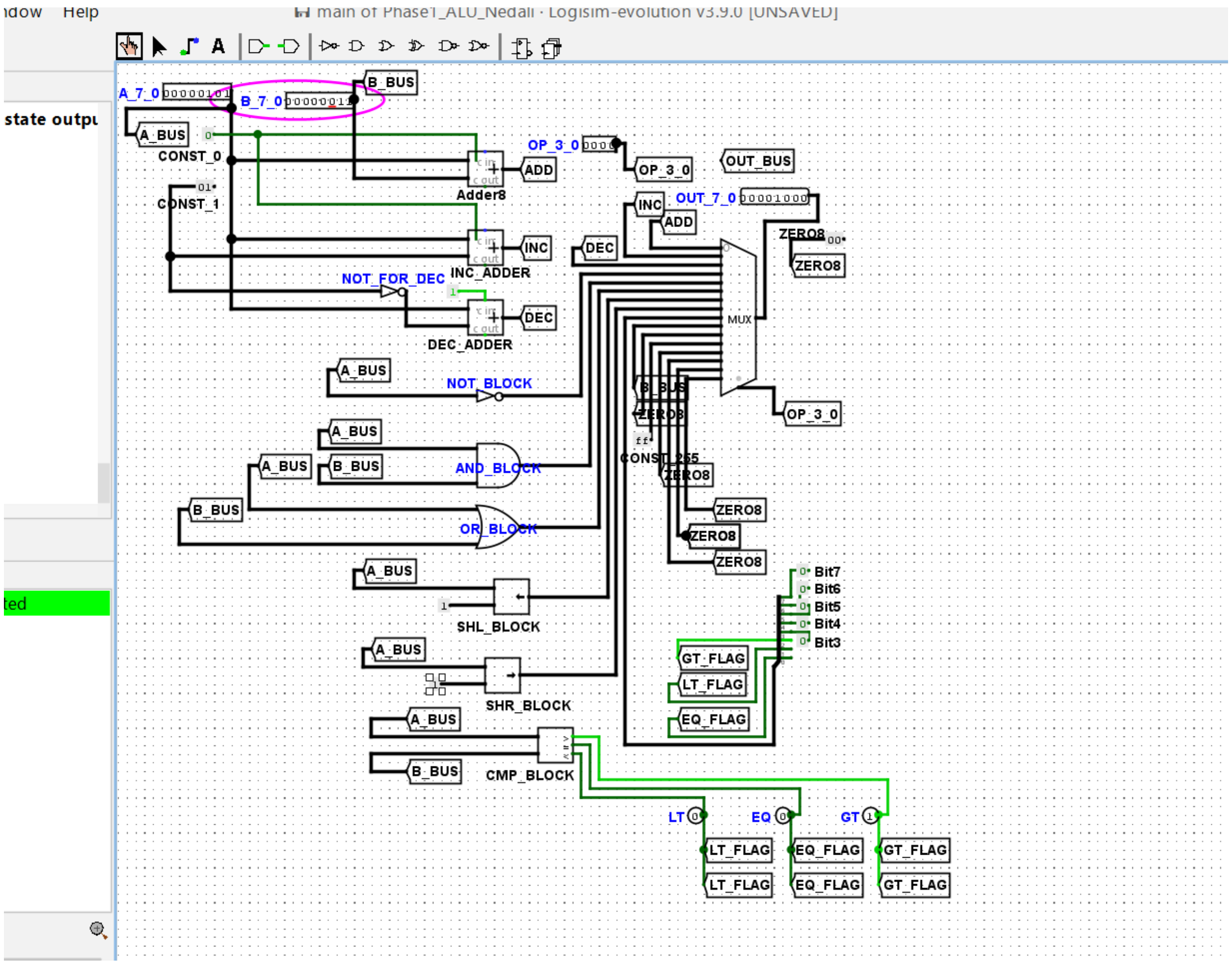
My design implements an 8-bit ALU with 14 different instructions and a small CPU with three registers and a simple control unit. The ALU and control logic use a clear 4-bit opcode scheme, and the complete opcode table is now documented in this report.

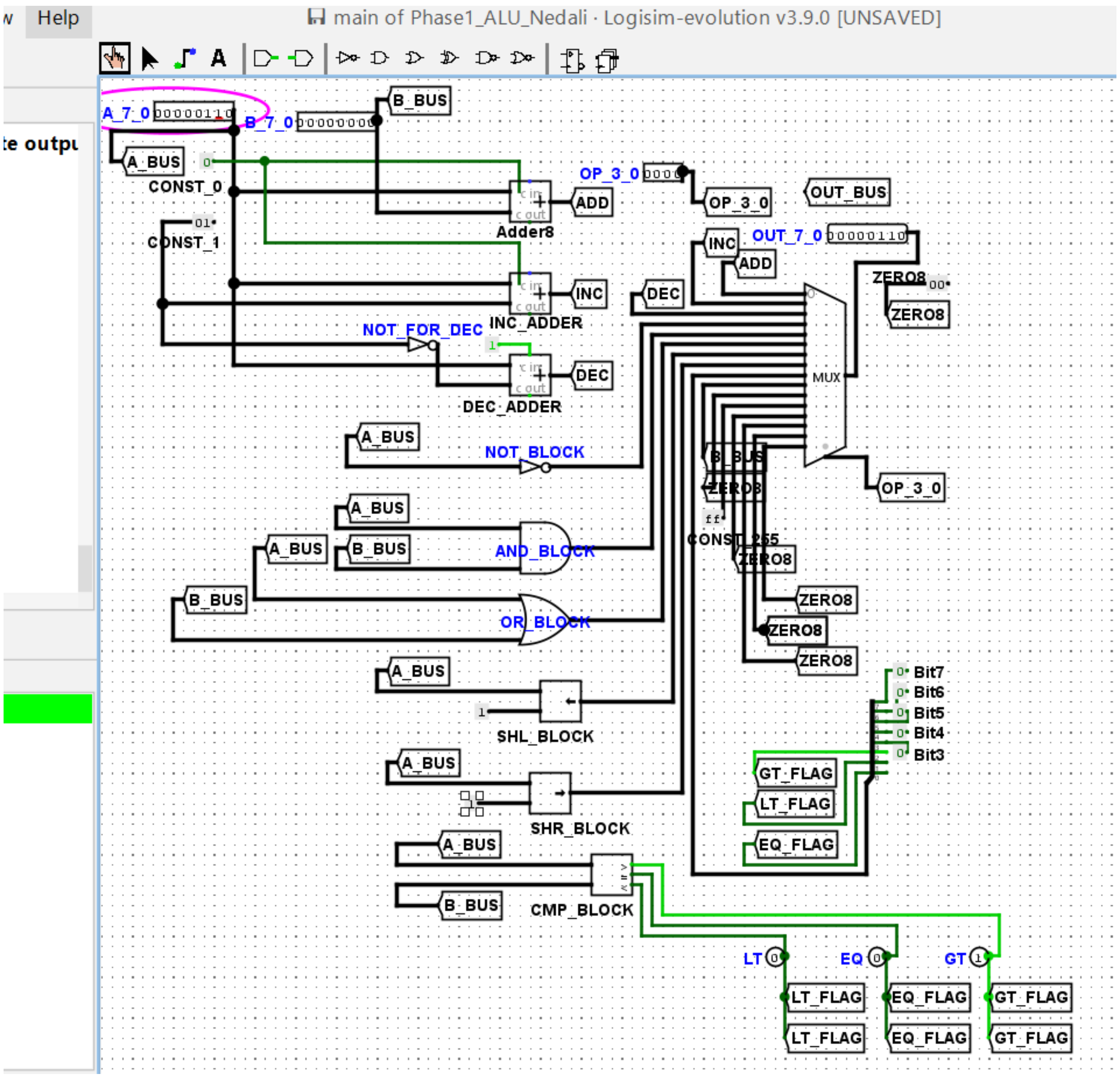
The circuits were tested with manual inputs and with the shift test program. The results and flags matched the expected behavior for the arithmetic, logic, shift, rotate, compare, and multiply instructions.




dow
Help
main of Phase1_ALU_Nedali · Logisim-evolution v3.9.0 [UNSAVED]
state outpu
ed
A_7_0
B_7_0
B_BUS
A_BUS
CONST_0
CONST_1
Adder8
ADD
OP_3_0
OUT_BUS
INC
INC_ADDER
NOT_FOR_DEC
DEC
DEC_ADDER
OUT_7_0
ZERO8
MUX
NOT_BLOCK
AND_BLOCK
OR_BLOCK
CONST_255
SHL_BLOCK
SHR_BLOCK
CMP_BLOCK
Bit7
Bit6
Bit5
Bit4
Bit3
GT_FLAG
LT_FLAG
EQ_FLAG
LT
EQ
GT


Help
main of Phase1_ALU_Nedali · Logisim-evolution v3.9.0 [UNSAVED]
A_7_0 00001000
B_7_0 00000110
B_BUS
A_BUS
CONST_0
CONST_1
Adder8
INC_ADDER
DEC_ADDER
NOT_FOR_DEC
ADD
INC
DEC
OP_3_0 0000
OP_3_0
OUT_BUS
OUT_7_0 00001110
ZERO8
MUX
NOT_BLOCK
AND_BLOCK
OR_BLOCK
SHL_BLOCK
SHR_BLOCK
CMP_BLOCK
CONST_255
Bit7
Bit6
Bit5
Bit4
Bit3
GT_FLAG
LT_FLAG
EQ_FLAG
LT
EQ
GT

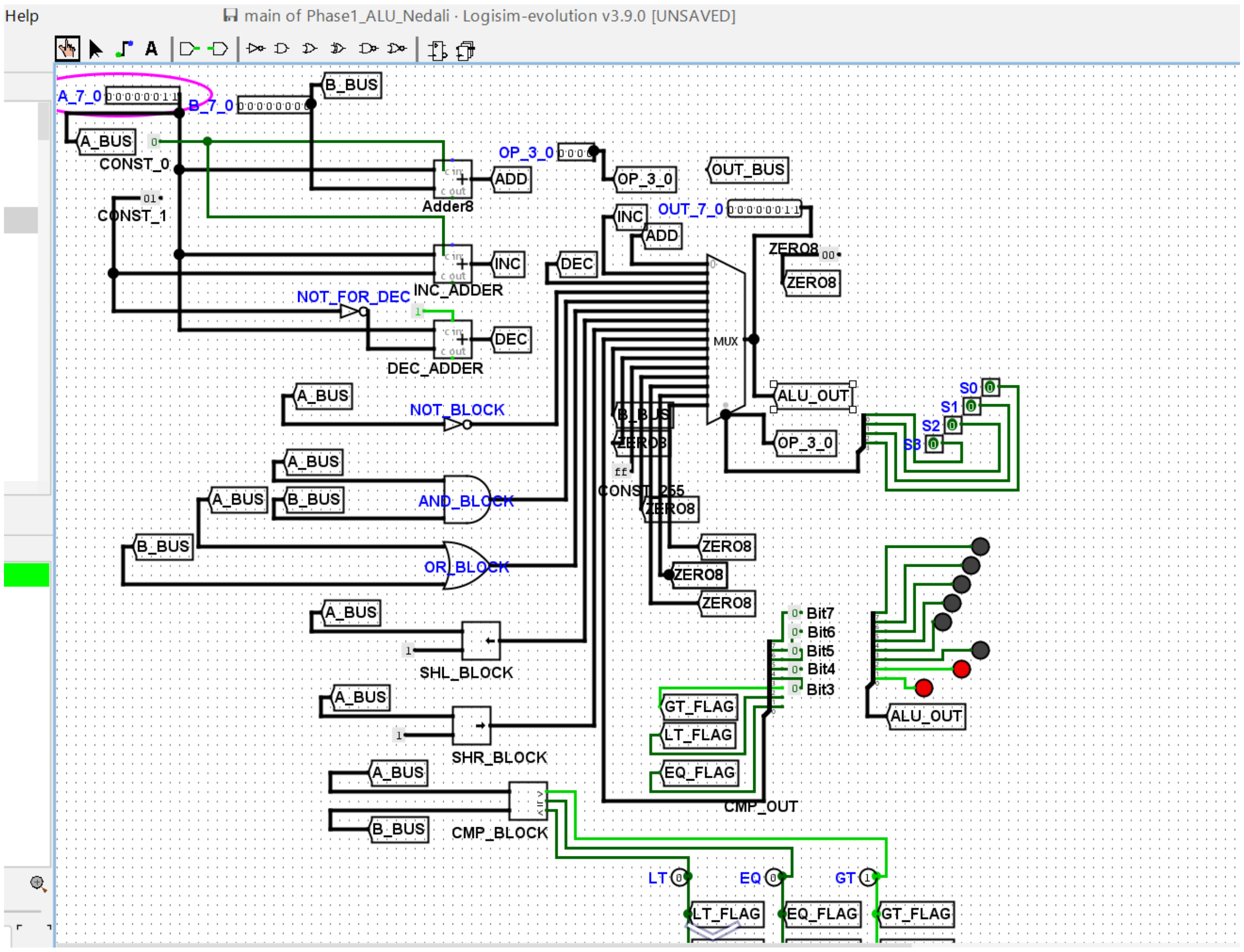

ndow
Help
main of Phase1_ALU_Nedali · Logisim-evolution v3.9.0 [UNSAVED]
state outpu
ted
A_7_0
B_7_0
B_BUS
A_BUS
CONST_0
CONST_1
Adder8
ADD
INC_ADDER
INC
NOT_FOR_DEC
DEC_ADDER
DEC
OP_3_0
OUT_BUS
OUT_7_0
ZERO8
MUX
NOT_BLOCK
AND_BLOCK
OR_BLOCK
SHL_BLOCK
SHR_BLOCK
CMP_BLOCK
CONST_255
Bit7
Bit6
Bit5
Bit4
Bit3
GT_FLAG
LT_FLAG
EQ_FLAG
LT
EQ
GT

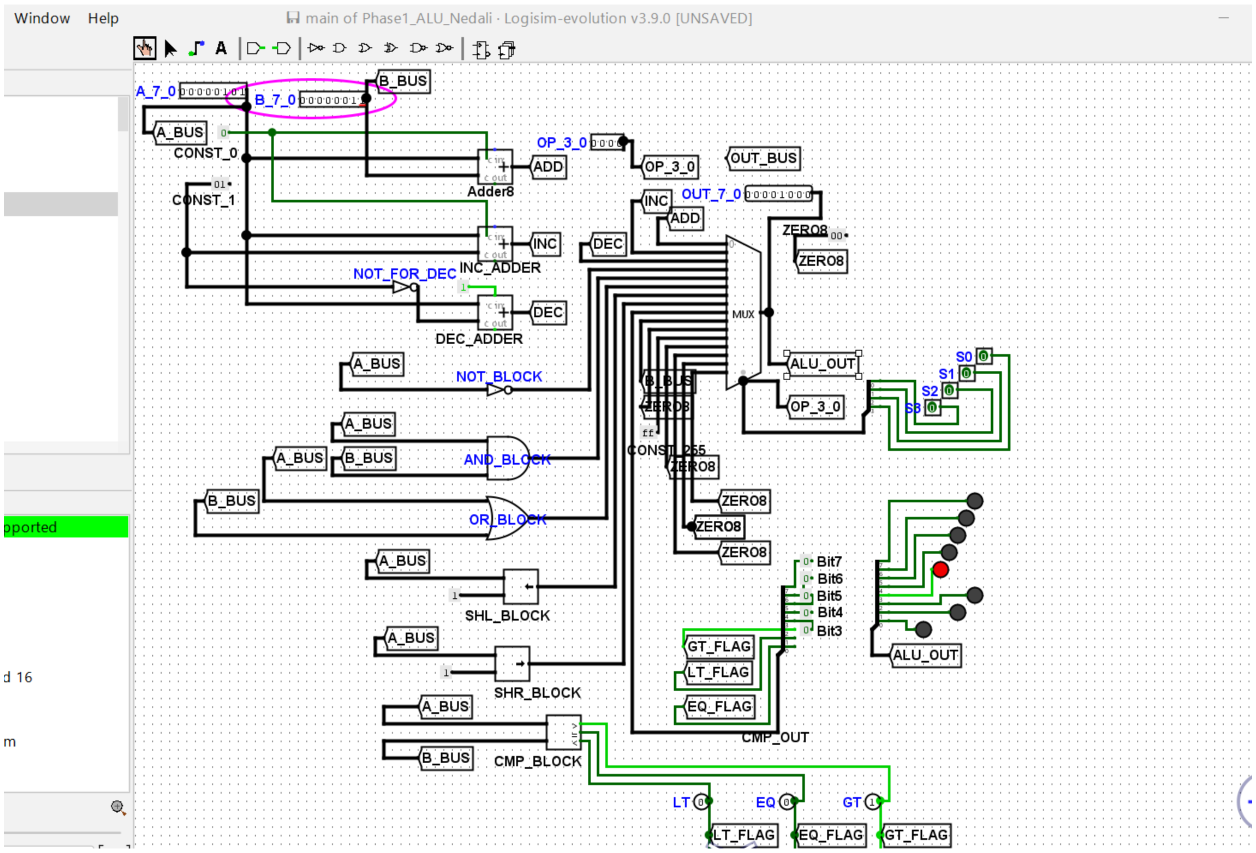


1. **Phase Two:**

The primary objective of Phase Two of the CPU project was to incorporate an Instruction Set Architecture (ISA) and a control selection mechanism into the 8-bit ALU developed in Phase One. This upgrade enabled the CPU to utilize a 4-bit opcode to select and execute ALU operations using a 16-to-1 multiplexer (MUX), making it possible to perform dynamic instruction-based computing. The opcode serves as a control signal that determines which operation, such as addition, increment, decrement, logical operations, or comparison, is performed. The instruction set includes operations like ADD (A + B), INC (A + 1), DEC (A − 1), AND (A AND B), OR (A OR B), NOT (A′), SHL (Shift Left), SHR (Shift Right), CMP (Compare A and B with EQ, LT, GT flags), XOR (A XOR B), NAND (NOT (A AND B)), ROL

(Rotate Left), and MUL (A × B, low 8 bits). The CPU uses two 8-bit operands, register A and register B, which serve as inputs for all ALU operations. The output is sent to register A after each instruction reads both operands simultaneously. The design and wiring are simplified with this two-operand structure, which enables actions such as adding A to B, performing logical operations with A to AND B, and updating condition flags using comparison instructions. The 16-to-1 MUX's select lines are linked to the 4-bit opcode input (S3-S0) to execute control logic. One of the inputs of the MUX is connected to each ALU output, including ADD, AND, OR, NOT, SHL, SHR, and CMP. To mimic the datapath control logic of a CPU, the MUX will then send the outcome of the current operation on an 8-bit channel, designated as ALU_OUT. In Logisim, four 1-bit input pins, labeled S3-S0, were connected to form a 4-bit bus. Steps were taken to implement this idea. On this bus, you can find the MUX's pick input (OP_3_0), and for each ALU operation, you can see the matching MUX input. To ensure everything was visible, a splitter was used to drive eight LEDs from the 8-bit MUX output, which is referred to as ALU_OUT. During testing, ring A = 00000101 (5) and B = 00000011 (3) confirmed that the functionality was working as expected. The findings were consistent across all opcodes tested, including ADD (00001000), INC (00000110), DEC (00000100), AND (00000011), OR (00000111), SHL (00001010), SHR (00000010), and CMP (flag outputs). Phase Two was a success in combining opcode-controlled instruction execution with output display, thereby turning the ALU into a processor core. Recently, the design can decipher binary instructions, select the appropriate operations via the MUX, and provide validated outputs. With this groundwork in place, the system can proceed to Phase 3, where the basic CPU architecture will be completed by integrating a fully functional Control Unit that generates enabling, branching, and sequencing signals.


Help
main of Phase1_ALU_Nedali · Logisim-evolution v3.9.0 [UNSAVED]
A_7_0
B_7_0
B_BUS
A_BUS
CONST_0
CONST_1
Adder8
ADD
OP_3_0
OUT_BUS
INC
INC_ADDER
NOT_FOR_DEC
DEC
DEC_ADDER
OUT_7_0
ZERO8
MUX
ALU_OUT
S0
S1
S2
S3
NOT_BLOCK
AND_BLOCK
OR_BLOCK
CONST_255
SHL_BLOCK
SHR_BLOCK
CMP_BLOCK
CMP_OUT
Bit7
Bit6
Bit5
Bit4
Bit3
GT_FLAG
LT_FLAG
EQ_FLAG
LT
EQ
GT



Phase Two, Opcode Table for the ALU

The ALU uses a 4-bit opcode (`OP_3_0`) to select which operation is sent to the output through the 16-to-1 multiplexer.
The table below lists **all 14 instructions** implemented in my design and the **exact 4-bit opcode** assigned to each one.

These opcodes correspond to the select lines that connect each operation block (adder, incrementer, decrementer, logic units, shifters, comparator, and custom instructions) to the ALU output.

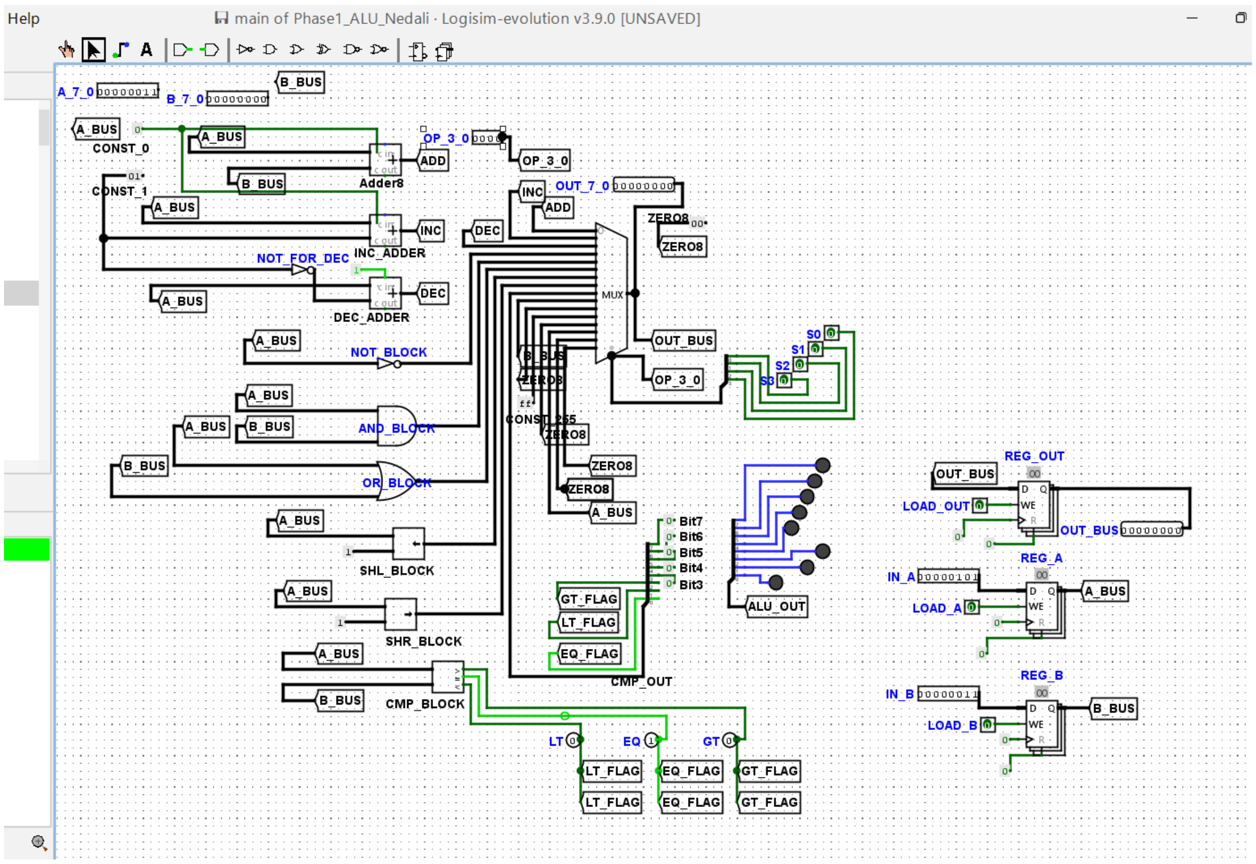
Opcode (S3 S2 S1 S0)	Instruction	Operation Description
0000	ADD	A ← A + B
0001	INC	A ← A + 1
0010	DEC	A ← A − 1
0011	CMP	Compare A and B → set GT, LT, EQ flags
0100	NOT	A ← NOT(A)
0101	AND	A ← A AND B

0110	OR	A ← A OR B
0111	SHR	A ← A >> 1
1000	SHL	A ← A << 1
1001	SUB	A ← A − B
1010	XOR	A ← A XOR B
1011	NAND	A ← NOT(A AND B)
1100	ROL	Rotate A left by 1 bit
1101	MUL	A ← (A × B) mod 256

Phase Three, Control Unit Implementation

The goal of Phase Three was to turn the ALU design from Phases One and Two into a working CPU by adding a Control Unit (CU) and a set of buffers to handle input and output data. This phase focused on establishing proper data flow between registers and the ALU through control signals, enabling the CPU to perform a complete execution cycle, loading operands, processing data, and storing results. To make this possible, three 8-bit registers were added to handle operands and outputs. There was a Write Enable (WE) control signal in each register that told it when to load data. The first operand register is REG_A, which gets its data from IN_A and sends it to the ALU through the A_BUS. It takes input from IN_B and sends it out through B_BUS. REG_B holds the second argument. The ALU's calculated result is stored in REG_OUT, which gets data from the OUT_BUS and sends it to the output pin. This setup allows data to move around in the CPU without any issues, and it maintains precise control over time and storage. To keep things organized internally, named pipes managed 8-bit links between registers and the ALU. The A_BUS tunnel brings part A data from REG_A.Q to the ALU's A input. In the same way, the B_BUS tunnel links REG_B.Q to the B input of the ALU. The OUT_BUS tunnel sends the ALU's calculated result to both REG_OUT.D and the output pin, ensuring that the results can be displayed or reused in other processes. This organized bus layout makes things clearer, facilitates troubleshooting, and ensures that signals reach all parts of the system efficiently. The control signals used in this phase determine the transmission of data and the selection of action. These are the 4-bit opcode choices (OP_3_0), which control the ALU process through the multiplexer, and the 1-bit signals LOAD_A, LOAD_B, and LOAD_OUT, which are used to load data into their respective registers. These signals work together to enable the Control Unit to plan the actions of the CPU by instructing it when to load operands, execute an operation, and store results. The Control Unit and register-based system worked as planned, as shown by tests. For instance, when A = 5 and B = 3 were used in the Addition (ADD) test, the correct answer of 8 was given by setting LOAD_A = 1, LOAD_B = 1, OP = 0000, and LOAD_OUT = 1. When A =

11001100 and B = 10101010 were put into the Bitwise AND test with OP = 0101, the result was 10001000. Additionally, the Right Shift (SHR) process functioned as expected, converting input A = 10000000 into output 01000000 when OP = 0111. The fact that all tests gave correct results shows that the Control Unit handled the processing loop correctly. Additionally, Phase Three successfully combined a Control Unit and a data stream based on registers, turning the ALU into a fully working 8-bit CPU core. Controlling data loading, choosing an ALU operation, and storing results in the system can now be done with coordinated control signals. This phase completes the CPU's basic working loop and prepares Phase Four, which tests and demonstrates how the CPU operates at the instruction level using assembly language code.



Phase Four (Assembly Language Programs):

Two assembly programs were written and executed on the special 8-bit CPU built in the previous steps of Phase Four. Through this part, the full capability of the CPU is demonstrated, including how the control unit (CU), memories, and arithmetic logic unit (ALU) work together to carry out tasks, perform iterative processes, and respond to conditional branches. The programs were developed with the assistance of the control unit from Phase 3 and the instruction set architecture created in Phase 2. Key concepts of computer organization, assembly language design, and control flow in a digital system are evident in the creation process **(Study.com, n.d.-a; Study.com, n.d.-b).**

Instruction Set and Architecture

The CPU's instruction set was inspired by fundamental ALU operations and logical control concepts found in standard computer architecture materials (Logic Gates – Building an ALU, n.d.). The following instructions were implemented for Phase Four:

"LOAD A, value" loads a value into Register A right away.

Loads a value into Register B right away with LOAD B, value.

ADD A, B will use the ALU to add the number from Register B to Register A.

CMP X, Y, looks at two numbers and sets comparison marks (EQ, LT, GT).

If the EQ flag is set, the BRZ label will branch to a label.

JMP label: This function does an absolute jump to a specific label.

SHR A, Moves all of Register A's bits one place to the right.

If the bit is 0, TESTLSB A sets EQ to 1. If it is 1, it sets EQ to 1.

HLT stops running the app.

These instructions form the backbone of the CPU's operation, demonstrating that the design can process arithmetic operations, make logical decisions, and perform control transfers (Logisim-Evolution, n.d.).

Program 1 – Add Numbers Until Zero

For as long as Register B is greater than zero, the first assembly program continually adds its value to Register A. Using the CMP and BRZ instructions, the program repeatedly checks Register B to see if it has reached zero. If it has, the program stops.

```
; Program 1: Add numbers until the new value is 0
START: CMP B, #0 ; Check if B == 0
        BRZ END ; If B is 0, exit loop
        ADD A, B ; A = A + B
        JMP START ; Repeat until B = 0
END: HLT ; Stop. Final sum stored in A.
```

This program demonstrates how loops and conditional control are utilized in the design of the CPU. It works similarly to high-level computer logic (such as a while loop), but it is controlled only at the assembly level. The repetitive structure ensures that the ALU and control unit function properly when branching conditions are present **(Study.com, n.d.-a).**

Program 2, Shift Right Until LSB Is Zero

A rational shift operation is done on Register A by the second assembly program until its least significant bit (LSB) becomes zero. The CMP command checks Register A for 11111111 (decimal 255) before starting the loop to ensure it doesn't run indefinitely.

```
; Program 2: Shift right until LSB is 0, unless A = 11111111
START: CMP A, #255 ; Check if A = 11111111
```

```
          BRZ STOP ; If true, stop immediately
SHIFT_LOOP: TESTLSB A ; Check LSB of A
          BRZ STOP ; Stop if LSB is 0
          SHR A ; Shift A right by one bit
          JMP SHIFT_LOOP ; Repeat
STOP: HLT ; Halt execution.
```

Bitwise processes and conditional switching are demonstrated at the hardware level in this software. Logical AND processes are used to perform the TESTLSB instruction, which checks whether A AND 00000001 = 0. The EQ flag is set by this bitwise test, which lets the BRZ command stop running when the LSB becomes 0 **(Study.com, n.d.-b; Logisim, n.d.)**.

Explanation of Flags

For conditional logic to work, the CMP instruction is needed. It compares two operands and changes the EQ, LT, and GT flags in the status register. The BRZ order then figures out how to control the program by reading these flags. The TESTLSB process adds a low-level logical condition that lets the CPU make even more decisions. These parts work together to enable the use of advanced control structures in machine-level code, such as statements or loops (Logic Gates – Building an ALU, n.d.).

Conclusion

In Phase Four, the custom-built 8-bit CPU is demonstrated to function entirely. By examining these assembly programs, it's clear that the design can perform both logical decision-making and repeated mathematical operations. The ALU of the CPU does correct calculations, and the control unit handles processes that are split and stopped. These results demonstrate that all the main components from earlier stages, logical circuits, arithmetic operations, and control logic, have been successfully integrated to create a customizable processor that functions (Logisim-Evolution); this marks the final step in the CPU design project. It is easy to see how it evolved from simple Boolean logic and logic gate design to a functioning central processing unit that can execute user-defined assembly programs.

References

Logisim. (n.d.). https://cburch.com/logisim/

Logisim-Evolution. (n.d.). *GitHub - logisim-evolution/logisim-evolution: Digital logic design tool and simulator*. GitHub. https://github.com/logisim-evolution/logisim-evolution

Logic Gates - Building an ALU. (n.d.).

http://www.csc.villanova.edu/~mdamian/Past/csc2400fa13/assign/ALU.html

Study.com. (n.d.). *Creating an assembly language using an instruction set*.

https://study.com/academy/lesson/building-an-alu-using-logisim.html

Study.com. (n.d.). Building an ALU using Logisim

https://study.com/academy/lesson/building-an-alu-using-logisim.html