Assignment 1: Creating a Simple ALU

Kareem Abu humra

Department of Computer Science, Study.com

Computer Science 306

June 24, 2024

**Phase One: Design and build an 8-bit ALU**

Objective**:** Addition

The objective of this phase is to design and build an 8-bit Arithmetic Logic Unit (ALU) using Logisim. The ALU should be capable of performing 14 different instructions, including basic arithmetic, logical operations, and shifts.

Addition

**A diagram of a machine

Description automatically generated**

Objective: Subtraction   
  
 The goal is to implement the subtraction operation in our 8-bit ALU using Logisim. This operation will allow the ALU to subtract one 8-bit number from another.

Design Approach:  
  
To perform subtraction in binary, we use the two's complement method. This involves inverting the bits of the number to be subtracted and adding one, which effectively converts the subtraction into an addition problem.

Implementation in Logisim:

1. **Setting Up Inputs**:
   * Place 8 input pins for operand A.
   * Place 8 input pins for operand B.
   * Add a single input pin for the initial Carry In, which we will set to 1 for this operation.
2. **Adding Components**:
   * Place an 8-bit NOT gate to invert the bits of operand B.
   * Place an 8-bit adder to add the inverted B and operand A.
3. **Connecting the Circuit**:
   * Connect the 8-bit input pins for operand A directly to the adder.
   * Connect the 8-bit input pins for operand B to the 8-bit NOT gate.
   * Connect the output of the 8-bit NOT gate to one input of the adder.
   * Set the carry-in input of the adder to 1 to complete the two's complement.
   * 8-bit adder: To add the inverted bits of B and the carry-in (which represents adding 1 for two's complement) to operand A.

Subtraction

**A diagram of a circuit

Description automatically generated**

By using the two's complement method, I was able to implement the subtraction operation in the 8-bit ALU. This allows the ALU to correctly subtract one 8-bit operand from another by converting the problem into an addition operation. The design is efficient and leverages existing adder circuitry to perform the operation.

Objective: Multiplication

The goal is to implement the multiplication operation in the 8-bit ALU using Logisim. This operation will enable the ALU to multiply two 8-bit operands.

Design Approach:

Binary multiplication is like decimal multiplication and involves adding shifted versions of the multiplicand based on the multiplier’s bits. This process can be implemented using shift registers and adders.  
  
Detailed Steps to Implement Multiplication:

1. Partial Products:
   * Each bit of the multiplier is used to create partial products. If the bit is 1, the multiplicand is shifted and added to the result. If the bit is 0, the partial product is 0.
2. Shift and Add:
   * For each bit in the multiplier, shift the multiplicand accordingly and add the result to the accumulated sum.

Implementation in Logisim:

1. Setting Up Inputs:
   * Place 8 input pins for operand A (multiplicand).
   * Place 8 input pins for operand B (multiplier).
2. Adding Components:
   * Use shift registers to handle the shifting of the multiplicand.
   * Use an array of adders to accumulate the partial products.
   * Use AND gates to create partial products by multiplying each bit of the multiplier with the entire multiplicand.
3. Connecting the Circuit:
   * Connect the input pins for operand A to the shift register.
   * Connect the input pins for operand B to the AND gates, which will control whether the shifted multiplicand is added to the result.
   * Use adders to sum the partial products generated by the AND gates.

Multiplication

A diagram of a computer

Description automatically generated

By using shift registers and adders, I implemented the multiplication operation in the 8-bit ALU. This allows the ALU to multiply two 8-bit operands efficiently by summing the appropriate shifted multiplicands based on the multiplier bits. The design is robust and leverages logical operations to achieve the desired functionality.

**Phase Two: Design the Instruction Set for the ALU/CPU**

Objective**:** The objective of this phase is to design the instruction set for the 8-bit ALU, create an opcode table, and implement a multiplexer to select operations based on the opcode. This will ensure that the ALU can perform a variety of operations efficiently.

|  |  |  |
| --- | --- | --- |
| Instruction | Opcode (Binary) | Description |
| Addition | 0000 | Adds two 8-bit operands (op1 + op2). |
| Increment | 0001 | Increments an 8-bit operand by 1 (op1 + 1). |
| Decrement | 0010 | Decrements an 8-bit operand by 1 (op1 - 1). |
| Comparison | 0011 | Compares two 8-bit operands and sets flags for equal, less than, or greater than. |
| Bitwise NOT | 0100 | Performs a bitwise NOT operation on an 8-bit operand (~op1). |
| Bitwise AND | 0101 | Performs a bitwise AND operation between two 8-bit operands (op1 & op2). |
| Bitwise OR | 0110 | Performs a bitwise OR operation between two 8-bit operands (op1 |
| Right Shift | 0111 | Performs a logical right shift on an 8-bit operand (op1 >> 1). |
| Left Shift | 1000 | Performs a logical left shift on an 8-bit operand (op1 << 1). |
| Subtraction | 1001 | Subtracts the second 8-bit operand from the first (op1 - op2). |
| Multiplication | 1010 | Multiplies two 8-bit operands (op1 \* op2). |
| Division | 1011 | Divides the first 8-bit operand by the second (op1 / op2). |
| Bitwise XOR | 1100 | Performs a bitwise XOR operation between two 8-bit operands (op1 ^ op2). |
| Arithmetic Right Shift | 1101 | Performs an arithmetic right shift on an 8-bit operand, preserving the sign bit (op1 >>> 1). |

In Phase Two, I successfully designed the instruction set for the 8-bit ALU and created an opcode table to define the binary codes for each operation. I also implemented a multiplexer in the ALU circuit to select the appropriate operation based on the opcode. The use of two operands (op1 and op2) was justified for its simplicity, efficiency, and consistency with common CPU architectures. This setup ensures that the ALU can perform a wide range of operations efficiently.

**Phase Three: Implement the Control Unit**

Objective: The objective of this phase is to design and implement the control unit for the ALU using Logisim. The control unit will manage the execution of operations by generating the necessary control signals based on the opcode.

A diagram of a machine

Description automatically generated

In Phase Three, I successfully designed and implemented the control unit for the 8-bit ALU using Logisim. The control unit decodes the opcode and generates control signals to select the appropriate operation in the ALU. This setup ensures that the ALU can perform addition, bitwise AND, and right logic shift operations efficiently based on the provided opcode.

**Phase Four: Writing Assembly Language Programs for the CPU**

Objective: The objective of this phase is to write assembly language programs for the CPU built in the previous phases. These programs will demonstrate the functionality of the ALU and control unit by performing specific tasks using the designed instruction set.

Program 1: Add Operands Until the New Value to be Added is 0  
This program will continuously add operands to a register until the new operand to be added is zero.

; Initialize registers

LOAD R1, #0 ; Initialize sum register R1 to 0

LOAD R2, Operand ; Load the first operand into R2

ADD\_LOOP:

ADD R1, R1, R2 ; Add the value in R2 to R1

LOAD R2, NextOperand ; Load the next operand into R2

CMP R2, #0 ; Compare the next operand with 0

JNZ ADD\_LOOP ; If the operand is not zero, repeat the loop

END:

HALT ; Halt the program

Program 2: Shift Register Content Until the Least Significant Bit is 0  
This program will shift the content of a register to the right until the least significant bit (LSB) is 0. If the content of the register is 11111111, the program will stop shifting.

; Initialize registers

LOAD R1, Operand ; Load the operand into R1

SHIFT\_LOOP:

RSHIFT R1, R1 ; Perform a logical right shift on R1

AND R2, R1, #1 ; Perform bitwise AND on R1 with 1, store result in R2

CMP R2, #0 ; Compare the result with 0

JNZ SHIFT\_LOOP ; If the LSB is not zero, repeat the loop

CMP R1, #255 ; Check if R1 is 11111111

JE END ; If R1 is 11111111, jump to end

HALT ; Halt the program

In this program, I load an operand into register R1. The program then enters a loop where it performs a logical right shift on R1. After each shift, it uses a bitwise AND operation to check if the least significant bit of R1 is 0. If it is not zero, the loop continues. Additionally, there is a check to see if the content of R1 has become 11111111. If this is the case, the program jumps to the end to avoid an infinite loop of shifting. The program halts when the LSB of R1 is zero.

In Phase Four, I successfully wrote two assembly language programs that utilize the ALU and control unit designed in previous phases. These programs demonstrate the functionality and flexibility of the instruction set by performing addition and shifting operations based on specific conditions.

**References**

Study.com. (n.d.). Building an ALU Using Logisim. Retrieved from <https://study.com>

Study.com. (n.d.). Creating an Assembly Language Using an Instruction Set. Retrieved from <https://study.com>

Study.com. (n.d.). Practical Application for Computer Architecture: Instruction Set Architecture. Retrieved from <https://study.com>

Study.com. (n.d.). What is APA Formatting. Retrieved from <https://study.com>

Study.com. (n.d.). How to Format APA Citations. Retrieved from <https://study.com>