Justin Carlson

2/6/2023

Prof. Mealy

CPE 233-07

Experiment 3: Arithmetic Logic Unit

**Executive Summary**:

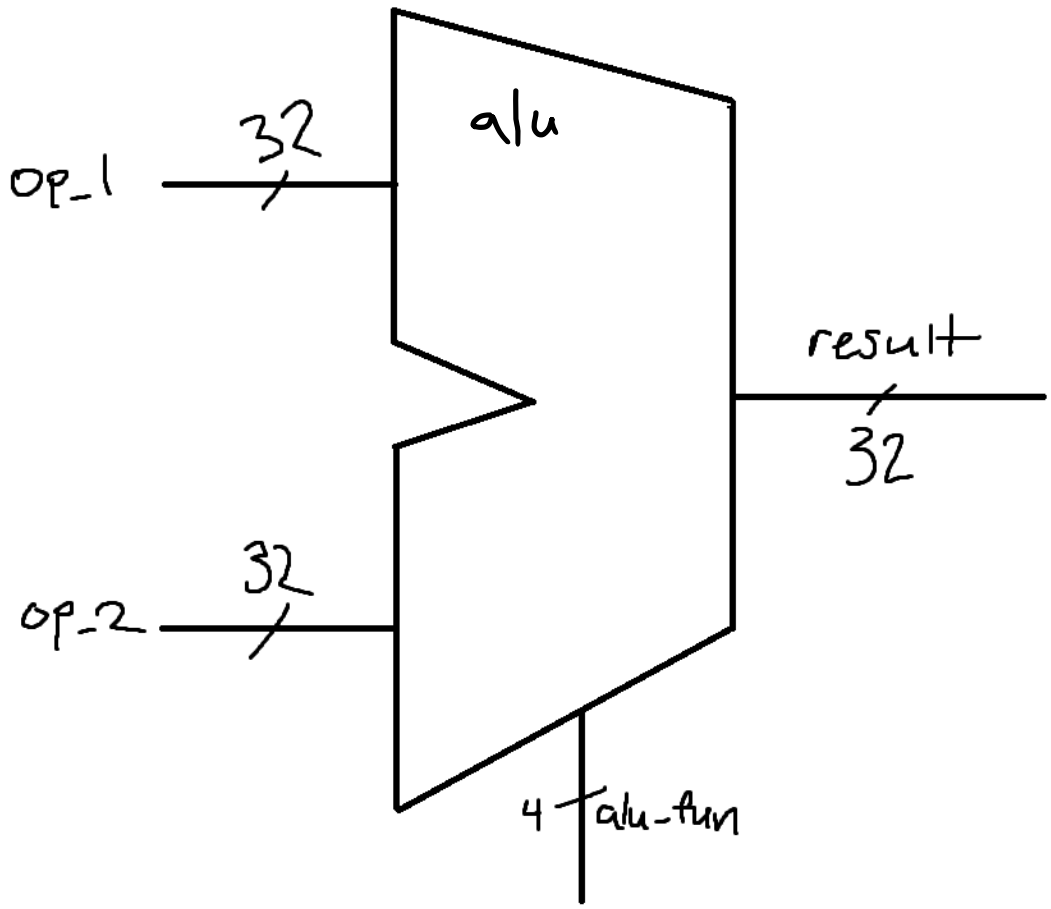
Experiment involves the implementation of the ALU, or arithmetic logic unit, that handles all the mathematical and logical operations for the Otter MCU. The ALU acts as a combinatorial decoder for the 11 possible operations. Two 32-bit inputs are fed into the module, and a 4-bit input called alu_fun controls the current operation.

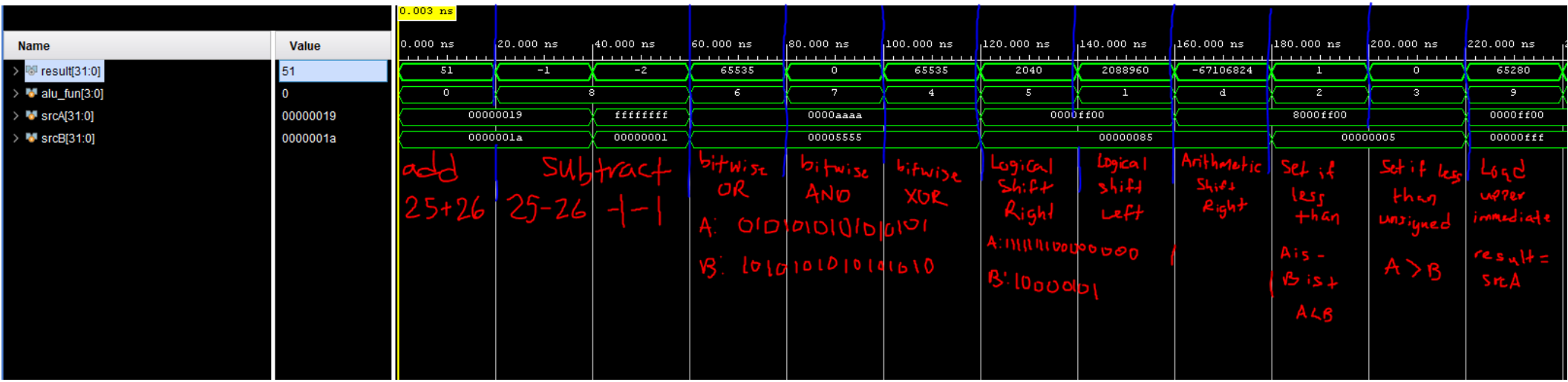
**ALU Verilog:**

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer: Justin Carlson
//
// Create Date: 01/23/2023 02:50:47 PM
// Design Name:
// Module Name: alu
// Project Name: alu
// Target Devices:
// Tool Versions:
// Description: Arithmetic Logic Unit for RISC-V Otter MCU
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////



module alu(
    input [31:0] op_1, op_2, // Two data inputs
    input [3:0] alu_fun, // Selector for the decoder
    output reg [31:0] result // Output
    );
```

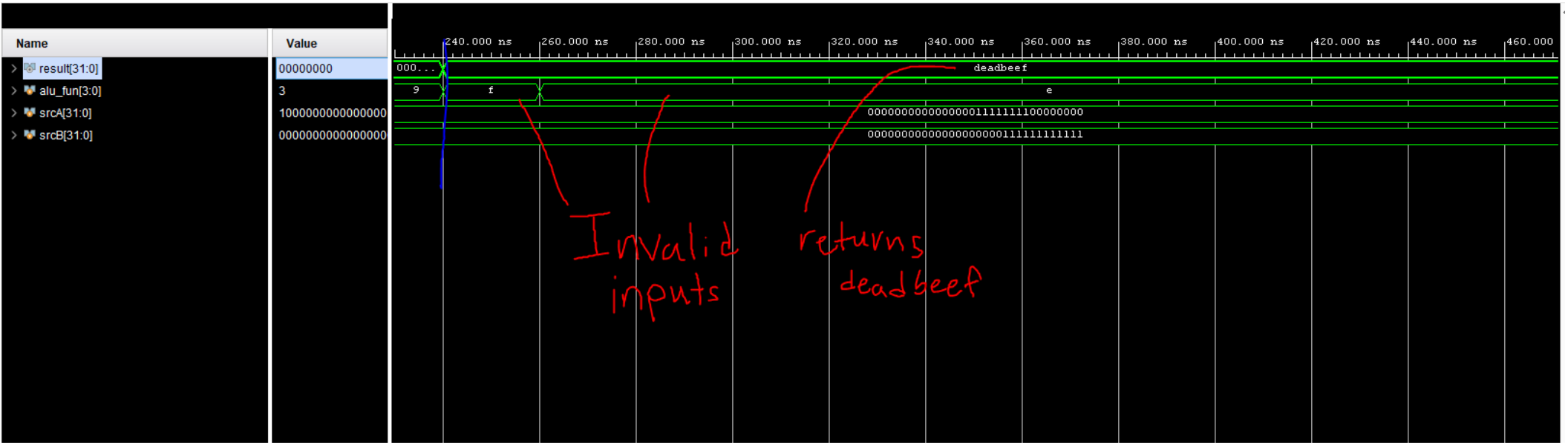
```
    always_comb
    begin
        case(alu_fun)
            4'b0000: result = op_1 + op_2; //Addition
            4'b1000: result = op_1 - op_2; //Subtraction
            4'b0110: result = op_1 | op_2; //OR operation
            4'b0111: result = op_1 & op_2; //AND operation
            4'b0100: result = op_1 ^ op_2; //XOR operation
            4'b0101: result = op_1 >> op_2[4:0]; //Logical shift right
            4'b0001: result = op_1 << op_2[4:0]; //Logical shift left
            4'b1101: result = $signed(op_1) >>> op_2[4:0];
//Arithmetic shift right
            4'b0010: result = ($signed(op_1) < $signed(op_2)); //Set
if less than
            4'b0011: result = op_1 < op_2; //Set if less than unsigned
            4'b1001: result = op_1; //Load upper-immediate
            default: result = 32'hDEADBEEF; //Non-functional ALU
inputs
        endcase
    end
endmodule
```

**Diagram**:






ALU Simulation Fig 1.




ALU Simulation Fig 2.

**Questions**:

1. Briefly describe at least two limitations of the RISC-V MCU ALU.

The ALU is not quickly capable of multiplication/division, and subroutines must be completed to execute these operations to maintain normal cycle times. Since some logical operands such as NAND, NOT, and XNOR are not present in the ALU, multiple operations must be done on the data to obtain the functionally equivalent Boolean equations.

2. Briefly describe what (or who) determines the number of instruction formats a given instruction set contains. No, the answer is not the ISA.

Since each format has a specific opcode and func3 combination, the number of instructions becomes the factor in charge of the number of formats so that there are enough combinations to support the ISA.

3. List the six RISC-V OTTER MCU instruction formats.

The Otter MCU has R, I, S, B, U, and J-type instructions.

4. Being that there are two add-type instructions (addi & add), briefly explain why there is only one “add” operation associated with the ALU.

Since the ALU is a combinatorial circuit, it will always just immediately add the operands in one cycle, hence no need for two different instructions.

5. The design of the RISC-V MCU ISA intentionally shares as many fields as possible in the six different instruction formats. Briefly explain the reason why the ISA designers did this.

This reduces the complexity of the design, and makes it more efficient to not swap bits between instructions.

6. You can typically use assembly language to make a given program written in a higher-level language run faster. Briefly but fully explain this concept.

If you can understand how your high-level code is written at a low level, you'll know the optimized and best ways to go about completing an operation better than someone with only an understanding of higher-level code. Pure programmers know what code does, but not why it does what it does, and sometimes their code won't be as efficient.

7. Briefly define the general purpose of assembler directives.

They tell the assembler what type of data that they're currently handling, like data, text, or labels. These directives aren't instructions and don't become part of the assembled code.

8. I have this strong desire to implement an instruction such as "add x3,x4,x5,x6", where the three right-most operands are summed and the result is stored in the left-most operand. Briefly but completely describe the changes in hardware (to existing RISC-V Otter modules only) that I would need to implement such an instruction.

The most significant architectural difference would be the requirement for a new instruction format that can handle four operands, and a lot of the hardware like the address gens, program memory, and ALU would need to change to accommodate handling these four operands.

9. Assembly languages are not considered “portable”. Briefly describe what this means and why this is the case.

Since assembly is tied to one type of computer architecture, it would not work on a different instruction architecture and would require modifications to work. As hardware architecture connects to different modules than it would in another, things would not work the way they would in a different type of computer.

10. Program control instructions rely on labels in the code to be encoded and subsequently work properly. If you look at the machine code for a given program, it contains no labels. Briefly explain how the machine code get away with not encoding labels.

Since the instructions contain the memory addresses for the labels if the instruction requires it, like a jump, then the assembler just needs to jump to the address that it needs to and ignore the human-readable part of the label.

11. Is it possible to have two of the same labels in an assembly language program? Briefly but completely explain. If this is possible, show an example in your explanation.

Yes, an assembly language program can contain two of the same label and they will both work, but only to an extent. You cannot call the lower duplicate label in a jump or a branch, and will only be able to

access the one with a smaller memory address. Since the system can't differentiate between two labels with the same name, it just chooses the first to access and ignores the other.

12. Is it possible to have two or more labels in an assembly language program with the same numerical value? Briefly but completely explain. If this is possible, show an example in your explanation.

Yes, this would be possible with the same stipulations as the problem before. All the labels are translated to numbers that are handled by the assembler when they need to jump or branch.

13. Can you write a RISC-V MCU assembly language program that "stops running"? Briefly explain your answer.

No, you can't necessarily just create a program that halts itself. The closest thing that can be done is to enter a subroutine that just jumps to itself in a loop, so no other code can be run, but in essence the computer is still doing something. The closest thing to being "done" with a program is to return from it, but you're just entering another subroutine giving the computer more work.

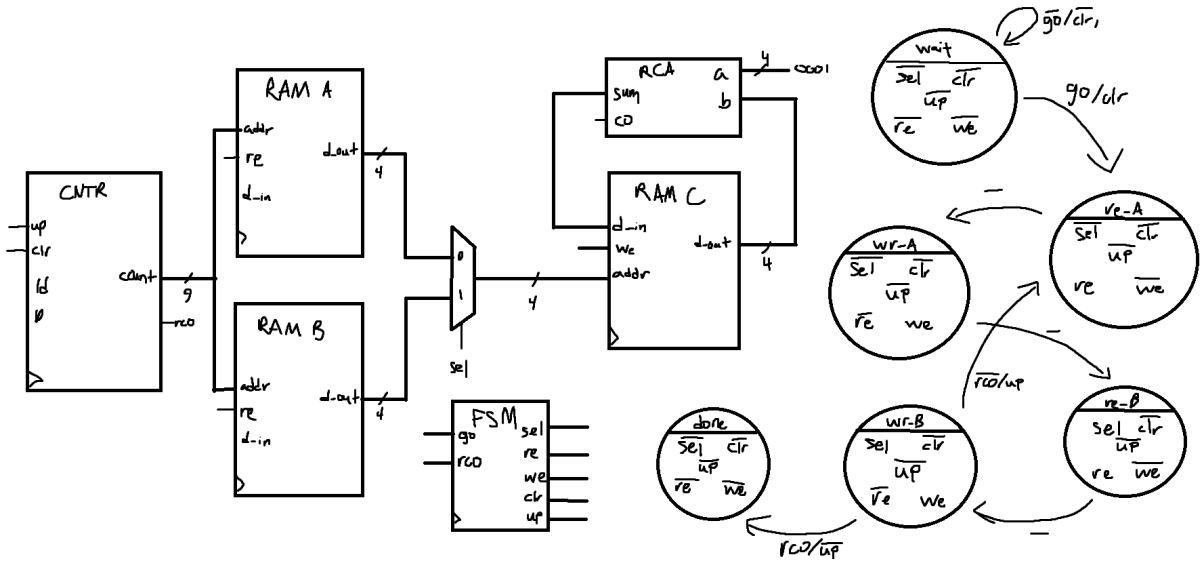
14. What is the significance of this value: -559038737. HINT: change the radix to hex. Describe an instance there using this value would be helpful to humans debugging your hardware.

In two's complement hex, this number becomes DEADBEEF, which is a very obvious way of saying that the invalid inputs on the ALU are supposed to not work. This clearly tells the programmer that this is an intended effect and it should possibly be left alone.

**Programming Assignment:**

```
500words:
init: li x20, 0xEEEE_FFFF // Memory Address
li x5, 500 // Loop counter
mv x10, 0 // Output register

sum: beq x5, x0, divide // Checks if loop is done
lw x11, 0(x20) // Loads word from memory
add x10, x10, x11 // Sums word to x10
addi x5, x5, -1 // Decrements loop counter
j sum

divide: ble x10, 65535, ret // Checks if x10 can be represented with
16 bits
srli x10, x10, 1 // If not, divides by 2
j divide // Repeats
```

**Hardware Assignment:**




RAM Value CNTR Fig 3.