CSC3050 Assignment 1 Report
ALU and Dot Products in RISC-V Assembly

Introduction

Objective:

The goal of this project was to implement a simple ALU (Arithmetic Logic Unit) and perform dot products using RISC-V assembly.

Importance of RISC-V Assembly:

Learning RISC-V assembly deepens understanding of low-level processor operations and how high-level code is translated into machine instructions. By working directly with assembly, we gain insights into how high-level programming languages are translated into machine instructions that processors can execute. This hands-on experience provides an essential foundation for understanding computer architecture and system-level programming.

ALU Implementation

Overview:

In this project, I implemented basic ALU functions including ADD, SUB, AND, OR, XOR, NOR, and bitwise shift instructions (SLL, SRL, SRA). The ALU manipulates registers and memory directly, performing arithmetic and logic operations.

Instruction Understanding:

Understanding instructions like add, sub, lw, sw, beq, bne was crucial. These instructions are the foundation for performing operations on data in registers and memory.

Challenges:

A major challenge was handling overflow in the ADD and SUB functions. In the ADD function, I first loaded the values from memory using lw, added them with the add instruction, and stored the result back using sw. To handle potential overflow, I checked the signs of the operands and the result using slti, which helped me determine if the values were negative or positive. I then used xor to compare the signs of the operands and the result. If the signs differed, it indicated a potential overflow, so I jumped to the overflow handler (check) that set the return value to 1. If no overflow was detected, the function returned 0, signaling a successful addition. The SUB function is just a tweaked version of the ADD function. Instead of directly subtracting, I used two's complement to convert the second operand into its negative equivalent by applying a bitwise not and then adding 1. The rest of the function follows the same pattern as in the ADD function.

Dot Product Implementation

Vector Dot Product

Algorithm Explanation:

The vector dot product is computed by multiplying the corresponding elements of two vectors and summing the products. In the assembly implementation, I used a loop to load each pair of elements, perform the multiplication, and accumulate the result in a register.

Implementation Details:

I implemented a loop that loads elements of the vectors with lw, multiplies them using mul, accumulates the result in a register, and then updates memory offsets using addi and sw. I found out how to deal with the base address and offset better at the matrix-vector dot product function.

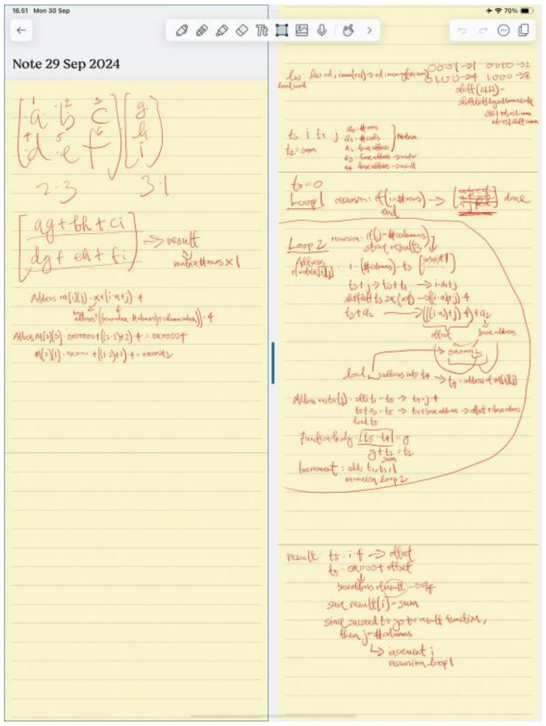
Matrix-Vector Dot Product

Algorithm Explanation:

The matrix-vector dot product is a more complex operation where each row of a matrix is multiplied by a vector, and the result is stored in a result vector.

Implementation Details:

For each row, I calculated the memory address of each matrix element by multiplying the row index by the number of columns and adding the column index. I then multiplied the matrix element by the corresponding vector element, accumulated the result, and stored it in the result vector. The function makes efficient use of registers and loops to calculate the dot product for each row. I will provide a glimpse of the thought process I jotted down below:



16.51 Mon 30 Sep

Note 29 Sep 2024

$$\begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix} \begin{bmatrix} g \\ h \\ i \end{bmatrix}$$

2·3 3·1

$$\begin{bmatrix} ag + bh + ci \\ dg + eh + fi \end{bmatrix} \rightarrow \text{result}$$

matrix #rows × 1

Address m[i][j] · x + (i·n + j) · 4

Address M[2][3] · 0x00000 + ((2·3) + 2) · 4 = 0x00024

m[2][1] · 0x0000 + ((1·2)+1) · 4 = [illegible]

lw: lw rd, imm(rs1) → rd : mem[rs1+imm]

load word

0001 → 1 0010 → 2

0100 → 4 1000 → 8

shift (sll) -

t0: i t1: j a0: #rows a1: #cols a2: base address → matrix

t2: sum a3: base address → vector

a4: base address → result

t0 = 0

Loop 1 recursion: if (i = #rows) → [illegible] done

end

Loop 2 recursion: if (j = #columns) store result

Address of matrix[i][j]: i · (#columns) → t3

t3 + j → t3 + t1 → i·a1 + j

shift left t3 2x (x4) → ((i·a1) + j) · 4

t3 + a2 → (((i·a1)+j)·4) + a2

offset base address

load address into t4 → t4 = address of m[i][j]

Address vector[j]: slli t1 → t5 → t5 = j·4

t5 + a3 → t5 → t5 + base address → offset + base address

load t5

function body: [t5 · t4] = y

y + t2 = t2

sum

Increment: addi t1, t1, 1

recursion loop 2

result t5: i·4 → offset

t5: 0x0000 + offset

base address of result → a4

save result[i] = sum

since succeed to go to result function,

then j = #columns

↳ increment i

recursion loop 1

Understanding RISC-V Assembly

Instruction Set Proficiency:

Mastering the RISC-V instruction set was key to successfully implementing the ALU and dot product functions. Using instructions like add, sub, lw, sw, mul, and control flow operations (beq, bne) allowed me to effectively move data, perform arithmetic operations, and manage program flow.

Similarities to High-Level Programming:

Though the syntax is different, RISC-V assembly retains many conceptual similarities to high-level languages, such as loops and conditionals. The primary difference is that in assembly, you manage registers and memory directly, requiring more attention to detail.

Register Management:

Efficient use of registers was essential, especially when implementing functions that involved multiple operands, such as the dot products. Proper allocation and management of registers ensured data was stored and processed correctly without overwriting important information.

Teamwork and Collaboration

Value of Peer Support:

Collaborating with my peers was invaluable in debugging and improving the ALU and dot product functions. My teammates helped spot errors in overflow detection and provided feedback on optimizing memory access. Working together allowed us to catch mistakes that were harder to notice individually. Discussing different approaches and solutions with my peers also helped me gain new insights into efficient assembly programming techniques.

Conclusion

This project enhanced my understanding of low-level programming and the RISC-V instruction set. Implementing an ALU and performing matrix-vector and vector-vector dot products gave me valuable hands-on experience with assembly-level computations. I improved my skills in register management and memory access. Additionally, the project reinforced the importance of careful planning and attention to detail when working at the assembly level. The skills developed here will be directly applicable in future projects involving embedded systems, performance optimization, and other areas where low-level programming is essential.