CSC3050 Project 1 Report – William Hansen Loe/122040046

1. ALU

The purpose of the ALU implementation is to use a 32-bit RISC-V default instruction set and implement those instructions with proper memory management, thus creating a feature or abstraction like those of higher-level programming languages where they could implement ALU without worrying about how the memory would work or how it would return the value.

The memory management and format of each procedure follow the format given by the guide. Most of the changes made are exceptional cases (overflow for ADD and SUB, shift handling, and branches). At the same time, the logical operator procedures (AND, OR, NOR, XOR, SLT) follow the same format given in the guide, with the difference being the corresponding logical operator. The following is the description of each function:

1. ADD, SUB:
    i. Takes in a0, a1, and a2 as the indices where the values would be stored.
    ii. Shift left twice to represent the offset of the values and added it with register a3 (location of memory). In this case, a3 would store the address of the result memory.
    iii. Load a0 and a1 with their corresponding address and perform add/sub-operation where it would be stored at a2.
    iv. Store a2 at 0(a3).
    v. Overflow handling: check whether a0 and a1 have different signs or not. If the signs are the same (ADD) or signs are different (SUB), check for possible overflow. Else, end the program and return a0 as 0.
    vi. If the inputs have different signs, check if the sign of the first input and the result is different. If true, then an overflow has occurred, and return a0 as 1; else, return a0 as 0 since overflow did not occur.
2. AND, OR, NOR, XOR, SLT: follows the same exact format as given by the guidelines, with the change being the corresponding logical operators.
3. BNE, BEQ:
    i. Find the address of index of a0 and a1 by shifting and adding a3 (designated address) to a0 and a1.
    ii. Load a0 and a1 with the corresponding address of 0(a0) and 0(a1).
    iii. Check whether beq/bne a0, a1 is true or not.
    iv. If true, return a0 as 1; otherwise, return it as false.

2. Dot Product

The purpose of this program is to implement a vector dot product and matrix-vector dot product by handling existing arrays and creating a mechanism for loops to access the array. Thus, this part of the project teaches how loops and arrays work at a low level.

For the vector dot product, the function starts by initialising the registers necessary for implementing the loop and dot product. Details of the registers are as follows:

1. a0, a1, and a2 registers follow the specification of the given guide
2. t0: element at 0(a0), t1: elements at 0(a1), t2: multiplication of t0*t1, t3: total sum, t4: loop counter

All the temporary (t) registers are initialised as 0, and then the program jumps to the looping phase.

Looping phase:

```
vector_loop_end:
    mv a0, t3   # Set t3 as the return value
    jr ra       # Return

vector_loop:
    addi t4, t4, 1  # Increment counter by one
    lw t0, 0(a0)    # Load t0 and t1
    lw t1, 0(a1)
    mul t2, t0, t1  # Multiply t0 and t1 and add into total sum of t3
    add t3, t3, t2
    beq t4, a2, vector_loop_end # If counter reaches a2, end the process and return
    addi a0, a0, 4  # Increment each array by 4 bytes
    addi a1, a1, 4
    j vector_loop
```

During the looping phase, the procedure begins by incrementing the counter t4 by 1 and will continue the loop until t4 is equal to a2. During the loop, t0 and t1 are loaded with 0(a0) and 0(a1), respectively, and the dot product could be calculated iteratively. If the condition at the end of the loop is not met, then the looping procedure increments the address a0 and a1 by 4 bytes because the next element of an integer array is 4 bytes away. Once the condition in the beq is satisfied, the program immediately ends and returns the dot product value.

The matrix-vector dot product implementation is similar to the vector dot product implementation. The differences are how the memories are incremented and how it uses two registers to track whether the program has reached the end of a column in a matrix. The program starts by initialising the registers t0-t5 and s1. The details of the registers are as follows:

1. a0, a1, and a2 registers follow the specification of the given guide
2. t0: element at 0(a0), t1: elements at 0(a1), t2: multiplication of t0*t1, t3: total sum, t4: row counter, t5: column counter
3. s1: address that points to the beginning of the vector

The procedure immediately jumps to the looping phase once the necessary values are initialised.

Looping phase:

```
matrix_row_next:
    addi t4, t4, 1  # Increase row counter by 1
    sw t3, 0(a4)    # Save the result of the row-vector multiplication in the result address
    beq t4, a0, matrix_loop_end # If it reaches the end of row, end the program
    mv a3, s1       # Else, restore the vector pointer to the original position
    addi a2, a2, 4  # Increment the matrix array by 4 bytes
    addi a4, a4, 4  # Increment the result vector by 4 bytes
    li t3, 0        # Reinitialize the sum
    li t5, 0        # Reinitialize the column counter
    j matrix_loop

matrix_loop:
    addi t5, t5, 1  # Increment column counter by 1
    lw t0, 0(a2)    # Load values from arrays
    lw t1, 0(a3)
    mul t2, t0, t1
    add t3, t3, t2
    beq t5, a1, matrix_row_next # If the column counter reaches the end of the column handle the next
row
    addi a2, a2, 4  # Else, increment the column and vector by 4 bytes
    addi a3, a3, 4
    j matrix_loop
```

Like the vector dot product, the program initially calculates the row-vector dot product. However, once it reaches the end of the current column, the program handles an additional case by changing the row case. It starts by incrementing the row counter and saving the row-vector dot product in the result pointer. If the matrix is not fully calculated, i.e. if the row counter is not equal to the number of rows, the program restarts the row-vector process by initialising the matrix pointer to the next row, incrementing the result vector by 4 bytes to an empty memory to store the next product, restoring the pointer a3 to its original position, and beginning the row-vector dot product process again.

```
student@31558222ac5f:~$ riscv64-unknown-elf-gcc -march=rv32im -mabi=ilp32 -O3 alu.s -o alu.out
student@31558222ac5f:~$ qemu-riscv32 alu.out
Test 1: ADD
Pass
Test 2: ADD
Pass
Test 3: ADD
Pass
Test 4: SUB
Pass
Test 5: SUB
Pass
Test 6: SUB
Pass
Test 7: BEQ
Pass
Test 8: BEQ
Pass
Test 9: BEQ
Pass
Test 10: BNE
Pass
Test 11: BNE
Pass
Test 12: BNE
Pass
Test 13: SLT
Pass
Test 14: SLT
Pass
Test 15: SLT
Pass
Test 16: SLL
Pass
Test 17: SLL
Pass
Test 18: SLL
Pass
Test 19: SRL
Pass
Test 20: SRL
Pass
Test 21: SRL
Pass
Test 22: SRA
Pass
Test 23: SRA
Pass
Test 24: SRA
Pass
Test 25: NOR
Pass
Test 26: NOR
Pass
Test 27: NOR
Pass
Test 28: OR
Pass
Test 29: OR
Pass
Test 30: OR
Pass
Test 31: XOR
Pass
Test 32: XOR
Pass
Test 33: XOR
Pass
Test 34: AND
Pass
Test 35: AND
Pass
Test 36: AND
Pass
```

```
student@31558222ac5f:~$ riscv64-unknown-elf-gcc -march=rv32im -mabi=ilp32 -O3 dot_prod.s -o dot_prod.out
student@31558222ac5f:~$ qemu-riscv32 dot_prod.out
Test 1: Pass
Test 2: Pass
Test 3: Pass
Test 4: Pass
```