Lab 3-RISC-V Processor

An Lihua
2022131001
Shanghaitech University, SIST
anlh2022@shanghaitech.edu.cn

***Abstract*—The lab aims to design a RISC-V processor supporting MAC (Multiply Accumulate) operation. The lab introduces a high-custom processor with a scalar core and a vector core.**

In this lab, I have completed both scalar core and vector core, and passed all the four tasks.

***Index Terms*—MAC, RISC-V, verilog**

I. Introduction

RISC-V is an open instruction set architecture (ISA) based on the principles of Reduced Instruction Set Computing (RISC), with V denoting the fifth generation of RISC (Reduced Instruction Set Computer), which represents the four previous generations of RISC processor prototype chips. Compared to most instruction sets, the RISC-V instruction set is free to be used for any purpose, allowing anyone to design, manufacture and market RISC-V chips and software.

RISC-V instruction set can be subdivided, including RV32I, RV32M, RV32F, RV32D, RV32A, RV32V etc. The instruction set RV32I is the fixed basic integer instruction set, which is also the core content of RISC-V. And the other instruction sets are different extended sets. RV32V is the set with vector extension. Moreover, you can add the custom instruction set following the instruction format. The first stable release of RISC-V vector extensions was released in September 2021.

The purpose of the designed RISC-V processor is to support the MAC operation. The expected operation is shown in Fig. 1. The dimensions of all matrices are 8 by 8 and each element of the matrices is 32-bits data. Matrix-A is the weight matrix. Matrix-B is the input matrix. Matrix-C is the bias matrix. Matrix-D is the output matrix.

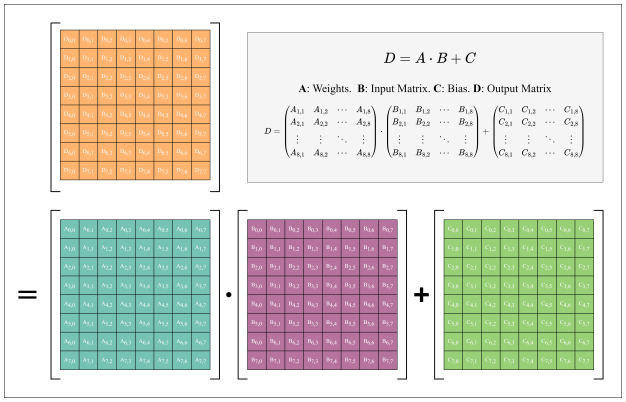

Fig. 1. MAC operation.

The top structure of the processor is given in Fig. 2. For scalar or vector core, each consists of Decoder, Execute (ALU), Memory and Write Back parts. Instruction fetch is to choose which core to use.

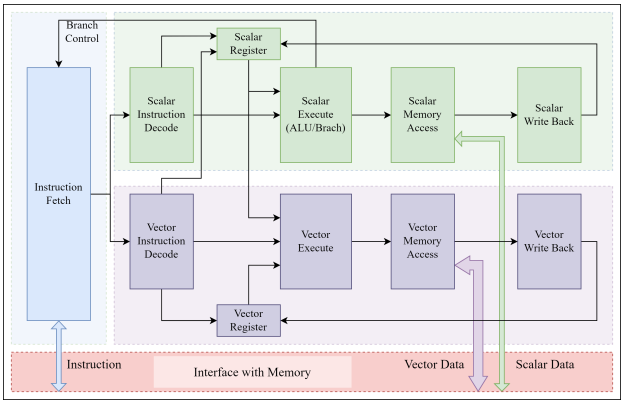

Fig. 2. Top structure.

II. Background

A. Scalar Processor

The main structure of a scalar processor consists of ALU and regfile. The input data are stored in regfile until read into the ALU to execute operations. This structure is not parallel, it can only do one operation one time. The simplified scalar processor's data flow is shown in Fig. 3.

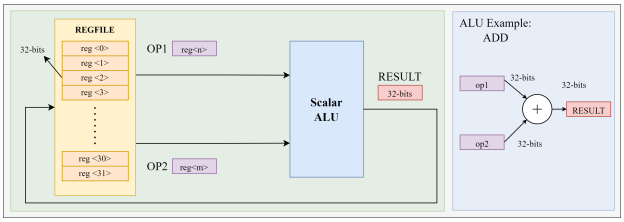

Fig. 3. Simplified scalar processor's data flow.

B. Vector Processor

The structure of a vector processor fits well with the problem of parallel computation of large amounts of data. A vector processor has multiple ALUs that are capable of performing the same operation many times at the same time. The basic idea of the vector architecture is to collect data elements from memory, put them sequentially into a large

set of registers, then operate on them sequentially using a pipelined execution unit, and finally write the result back to memory. The key feature of vector architecture is a set of vector registers.

Fig. 4 shows a simplified vector processor's data flow. The scalar operand in scalar architecture is expanded to vector operand in vector architecture. And the relative REGFILE and ALU are expended with vector feature.

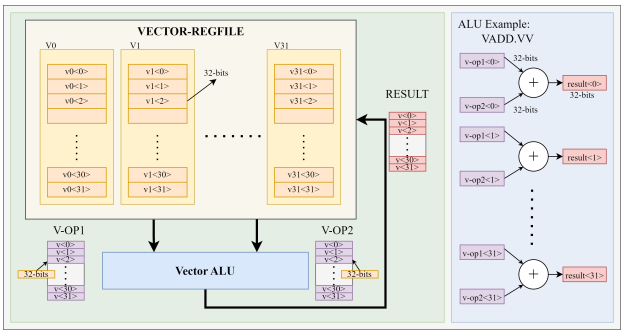

Fig. 4. Simplified vector processor's data flow.

C. Comparision

By comparing the software and hardware design implementations, several interesting points between the above two processor architecture are clearly noticeable.

For the software:

- The vector processor greatly reduces the bandwidth requirements for dynamic instructions, and the number of instructions in the vector version of the assembly is much smaller than the number of instructions in the scalar version of the assembly code. This is mainly due to the vector processor's ability to directly compute multiple sets of data in parallel, eliminating the need to use circular instructions.
- The pipeline blocking frequency is much lower in vector processors than in scalar processors.
- The vector processor takes less cycles to compute MAC with its parallel structure, since the scalar processor can only complete one operation in one cycle and takes longer cycles.

For the hardware:

- The basic concept of the hardware architechture is similar. On the other hand, the vector processor is based and expanded on the scalar processor.
- The vector processor uses more resources to realize its parallel computation.

III. Task 1

For task 1, we need to complete several instructions, basically including RV32I and RV32M. I will introduce each module in detail.

A. inst_decode

First of all, is the *inst_decode* module. It parses every scalar instructions and then decode them to control different functions.

We need to parse the different types of instructions using the table in Fig. 5 and the code of verilog is shown in Fig. 6.

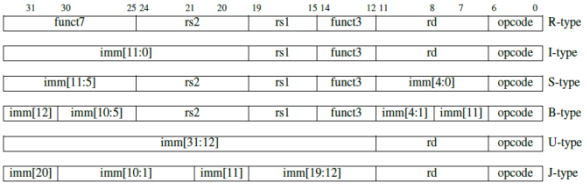

Fig. 5. Instruction format.

```
// ================================================
// Parse
// ================================================
assign imm_i    = inst_i[ 31 : 20 ] ;
// hint: to add imm_u, imm_b, imm_j, imm_s
assign imm_s    = {inst_i[31:25], inst_i[11:7]} ;
assign imm_b    = {inst_i[31], inst_i[7], inst_i[30:25], inst_i[11:8], 1'b0} ;
assign imm_j    = {inst_i[31], inst_i[19:12], inst_i[20], inst_i[30:21], 1'b0} ;
assign imm_u    = {inst_i[31:12], 12'b0} ;
```

Fig. 6. Instruction format in verilog.

Then we should initialize every instruction with the OP-CODE and FUNCTx defined before. After that, we begin to decode.

To control ALU, we need to assign proper instruction to *alu_opcode*, as well as the *rs1* and *rs2* signals. It is easy to find the relationship after understanding the meaning of each instruction.

Some instruction use immediate numbers, we need to assign them with expanded of *imm_x* signals we have parsed in Fig. 6. Similarly, we also need to assign the two branching signals properly.

Then signal *operand_rs1* and *operand_rs2* will be passed into ALU as inputs. *mem_x* and *wb_x* are connected with *mem* module and *write_back* module respectively, which will be introduced later.

B. execute

This module is actually the ALU part, which will decide different operations to execute. It is worth to notice that the output is depended on *jump_en*, if *jump_en* is pulled up, the output should be *current_pc + 4*, otherwise is the output address itself. We also need to pass the branching signals in this module, which is connected with *inst_fetch* module.

C. mem

This part is used to connect the output from ALU with RAM, it is really simple to realize. The only point we need to notice is that we pull up the write mask to disable it.

D. write_back

This module is to decide which data to write back to regfile. The critical enable signals have been specifed in *inst_decode* module.

E. inst_fetch

This module focus on the instruction of branching and jumping. It decides which number the current PC address needs to plus, 4 with branching (jumping) instruction and 1 with others.

F. regfile

This module is a simple register to store the temporary data.

G. Result

Fig. 7 shows the result from the terminal, which has passed all the testcases.

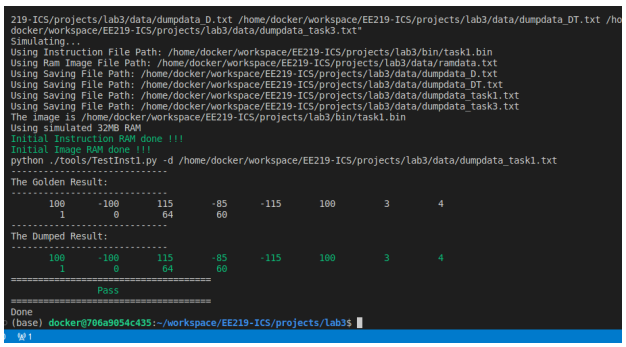

Fig. 7. Task1 Pass.

IV. Task 2

Fig. 8 shows the assemble code of Task 2. It is translated from C code and I do not use the transposed matrix format.

For line 5-13 is to setup the memory address. Then we use three loops to compute the whole MAC operation. Take the example of computing the first output D[0][0], line 27-34 is the third loop, complete the basic MAC operation for the first output, which means multiply all the 8 numbers in the first row of A and the first column of B, and accumulat them in reg[x20]. The second loop is from line 19-45, in this loop we load the data C and add it with reg[x20] to get the first output D[0][0]. After that, save the result to memory and update the addresses. The first loop is the largest loop, from line 16-50, which controls the addresses.

The printed information of MAC using assemble code is shown in Fig. 9, and the results are correct.

V. Task 3

Task 3 is very similar to Task 1, so I will only describe the different parts.

```
lab3 > asm > task2.asm
1   ; ====================================
2   ; task2
3   ; ====================================
4
5   lui     x1,     2148532224    ; x1 = 0x80100000
6   lui     x3,     2150629376    ; x3 = 0x80300000
7   lui     x5,     2152726528    ; x5 = 0x80500000
8   lui     x7,     2154823680    ; x7 = 0x80700000
9
10  addi    x9 ,    x1,     0     ; addr_A = A_baseaddr
11  addi    x10,    x3,     0     ; addr_B = B_baseaddr
12  addi    x11,    x5,     0     ; addr_C = C_baseaddr
13  addi    x12,    x7,     0     ; addr_D = D_baseaddr
14
15  addi    x13,    zero,   8     ; loop = 8
16  addi    x14,    zero,   0     ; first loop
17  loop1:
18
19  addi    x15,    zero,   0     ; second loop
20  loop2:
21
22  addi    x16,    zero,   0     ; third loop
23  addi    x22,    x10,    0     ; addr_B = reg[x10]
24  addi    x20,    zero,   0     ; clear reg[20]
25  loop3:
26
27  lw      x17,    0(x9)         ; load data_A
28  lw      x18,    0(x22)        ; load data_B
29  mul     x19,    x17,    x18   ; reg[x19] = data_A * data_B
30  add     x20,    x20,    x19   ; accumulation result
31  addi    x9,     x9,     4     ; addr_A = addr_A + 4
32  addi    x22,    x22,    32    ; addr_B = addr_B + 32  // addr_col(B) + 1
33  addi    x16,    x16,    1
34  blt     x16,    x13 ,   loop3 ; thrid loop end
35
36  lw      x21,    0(x11)        ; load data_C
37  add     x20,    x21,    x20   ; reg[20] = row(A)*col(B) + C
38  sw      x20,    0(x12)        ; save x20 into mem[x12]
39  addi    x20,    zero,   0     ; clear reg[20]
40  addi    x11,    x11,    4     ; addr_C = addr_C + 4
41  addi    x12,    x12,    4     ; addr_D = addr_D + 4
42  addi    x15,    x15,    1
43  addi    x9,     x9,     -32   ; addr_A = addr_A - 32
44  addi    x10,    x10,    4     ; reg[x10] = reg[x10] + 4
45  blt     x15,    x13,    loop2 ; second loop end
46
47  addi    x9,     x9,     32    ; addr_A = addr_A + 32
48  addi    x10,    x10,    -32   ; reg[x10] = reg[x10] - 32
49  addi    x14,    x14,    1
50  blt     x14,    x13,    loop1 ; first loop end
```

Fig. 8. Assemble code of Task 2.

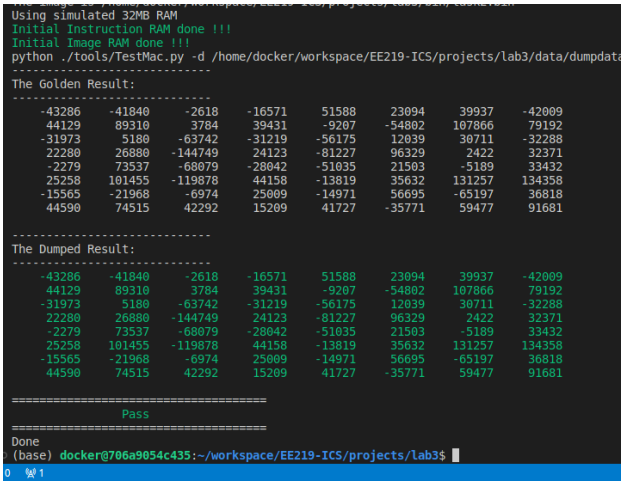

Fig. 9. Task2 Pass.

A. *v_inst_decode*

Simiarly, we use the instruction format shown in Fig. 10 to parse the instruction. And the other signals' realize in this module is similar to the Scalar one, which only needs simple modification to realize. Need to notice that *operand_vs1* has three different cases for classified.

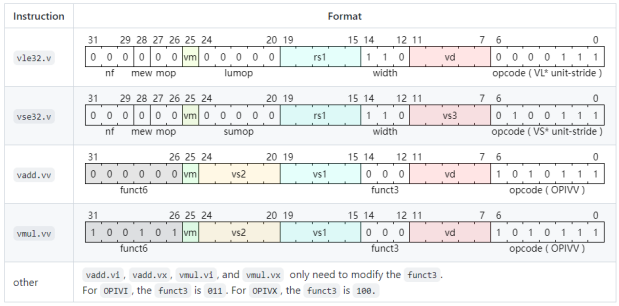

Fig. 10. Instruction format.

B. *v_execute*

This module is quite simple since it only has two operations need to complete. However, as the data in Vector processor is 256 bits (combined eight 32 bits data together), it needs a short loop to assign the correct results.

C. *others*

For the other modules, they are exactly the same as the Scalar one, so I will not talk about it again.

D. *Result*

Fig. 11 shows the result from the terminal, which has passed all the testcases.

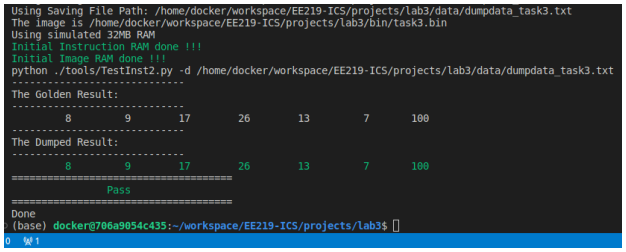

Fig. 11. Task3 Pass.

VI. Task 4

The algorithm is similar with task2, we load the address of every matrix first, and then use several loops to complete the MAC operation. Also, take the computing process of D[0][0] for example.

The critical loop is loop 3 from line 30-36, since we have used the vector operation in loop 2 to compute the A[0][0:7] * B[0:7][0] and have stored the results in a temporary address reg[x14], in loop 3 we need to splite the results and accumulate all of them, and then save the new result into origin address of matrix D. Here for the convience of address controlling, we use the transposed format for matrix B, and the other matrix are in common format. So far, we have used loop 1, 2 and 3 to complete A*B, next step is to finish VADD operation to get the final results.

We need to setup a new loop 4, since we have all the results of A*B in matrix D, we only need to fetch the data in D and C row by row, and simply use VADD operation to get the results of A*B+C. Finally, we save the final results in address of matrix D.

It is obvious that the vector operation is more efficient compared to the scalar process, since it is parallel to compute 8 numbers at one time.

lab3 > asm > task4.asm

```
4
5   lui     x1,     2148532224          ; x1 = 0x80100000
6   lui     x4,     2151677952          ; x3 = 0x80400000
7   lui     x5,     2152726528          ; x5 = 0x80500000
8   lui     x7,     2154823680          ; x7 = 0x80700000
9
10  addi    x9 ,    x1,     0           ; addr_A = A_baseaddr
11  addi    x10,    x4,     0           ; addr_B = B_baseaddr
12  addi    x11,    x5,     0           ; addr_C = C_baseaddr
13  addi    x12,    x7,     0           ; addr_D = D_baseaddr
14
15  addi    x13,    zero,   8           ; loop = 8
16  addi    x14,    x12,    256         ; extension of addr_D
17  addi    x21,    zero,   0           ; loop4
18  addi    x16,    zero,   0           ; loop1
19  loop1:
20
21  addi    x17,    zero,   0           ; loop2
22  addi    x18,    x10,    0           ; addr_B = reg[x10]
23  loop2:
24
25  vle32.v vx3,    x9,     1           ; vx3 = mem[addr_A]
26  vle32.v vx2,    x18,    1           ; vx2 = mem[addr_B]
27  vmul.vv vx4,    vx3,    vx2,    0   ; vx4 = vx2 * vx3
28  vse32.v vx4,    x14,    1           ; mem[addr_D_ext] = vx4
29
30  addi    x15,    zero,   0           ; loop3
31  loop3:
32  lw      x19,    0(x14)              ; load data_D_ext
33  add     x20,    x20,    x19         ; accumulation result
34  addi    x14,    x14,    4           ; addr_D_ext = addr_D_ext + 4
35  addi    x15,    x15,    1
36  blt     x15,    x13,    loop3       ; loop3 end
37
38  sw      x20,    0(x12)              ; save x20 into mem[x12]
39  addi    x12,    x12,    4           ; addr_D = addr_D + 4
40  addi    x20,    zero,   0           ; clear reg[20]
41  addi    x18,    x18,    32          ; addr_B = addr_B + 32
42  addi    x17,    x17,    1
43  blt     x17,    x13,    loop2       ; loop2 end
44
45  addi    x9,     x9,     32          ; addr_A = addr_A + 32
46  addi    x16,    x16,    1
47  blt     x16,    x13,    loop1       ; loop1 end
48
49  addi    x12,    x7,     0           ; reset reg[x12]
50  loop4:
51  vle32.v vx5,    x12,    1           ; vx5 = mem[addr_D]
52  vle32.v vx6,    x11,    1           ; vx6 = mem[addr_C]
53  vadd.vv vx6,    vx6,    vx5,    0   ; vx6 = vx6 + vx5
54  vse32.v vx6,    x12,    1           ; mem[addr_D] = vx6
55  addi    x11,    x11,    32          ; addr_C = addr_C + 32
56  addi    x12,    x12,    32          ; addr_D = addr_D + 32
57  addi    x21,    x21,    1
58  blt     x21,    x13,    loop4       ; loop4 end
```

Fig. 12. Assemble code of Task 4.

The printed information in the terminal of MAC using assemble code in Vector Processor is shown in Fig. 9, and the results are correct.

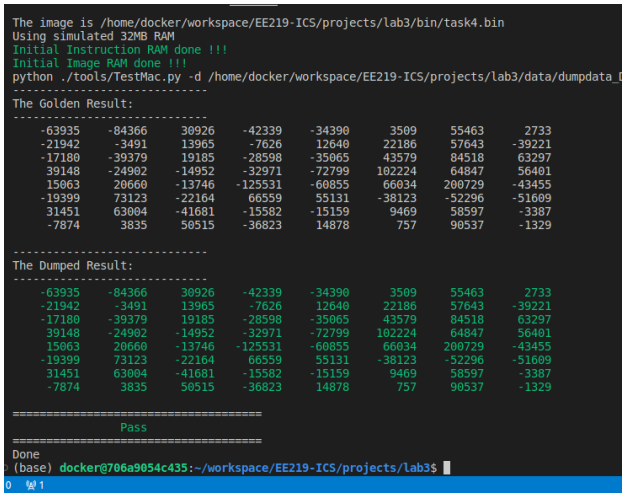

Fig. 13. Task4 Pass.

VII. Conclusion

In this lab, I have completed both scalar processor core and vector processor core, and successfully completed MAC operation using assembly code in both the two processor cores, also passed all the four tasks. By completing the Verilog codes and designing the corresponding assembly codes, I have a deeper understanding of computer architecture and the RISC-V instructions.

References

[1] Siting, Liu. EE219: ICS, 2022-Fall