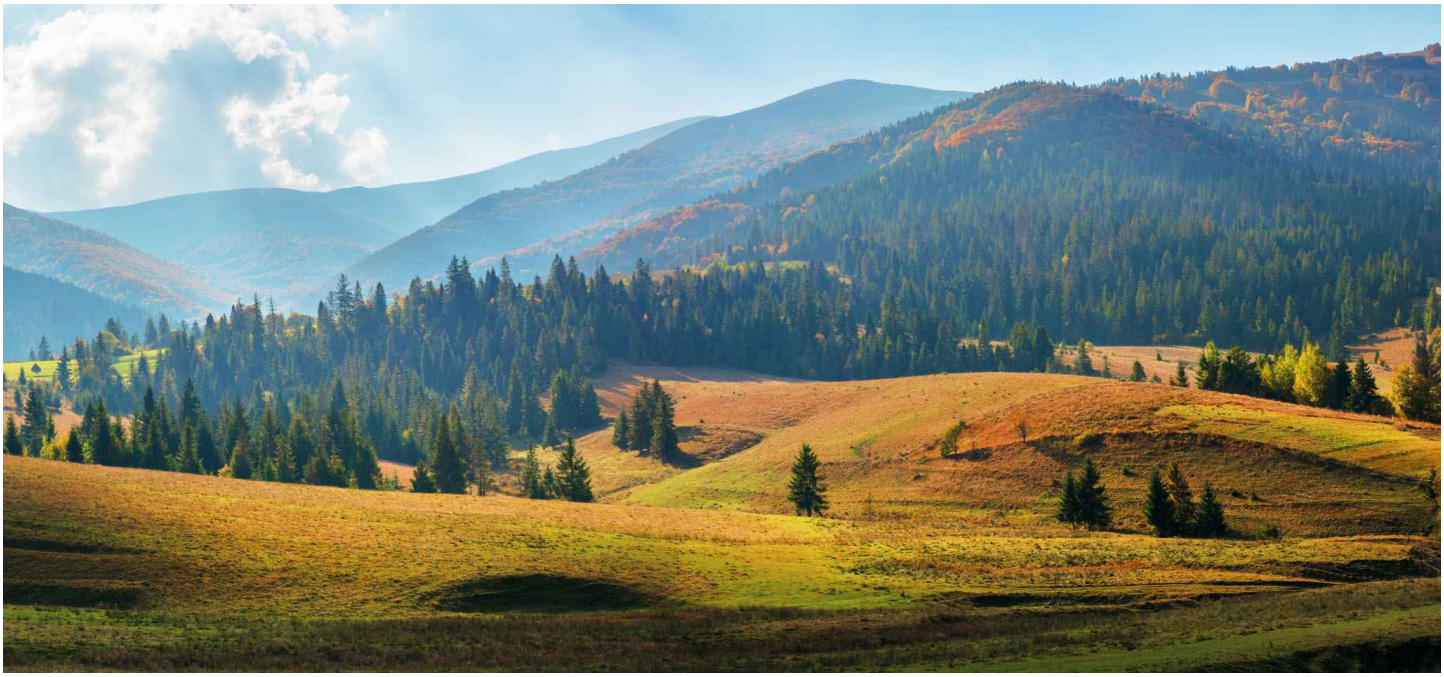

Lab0 Report

Basic Information

- Name :冯潇
- E-mail:fengxiao2023@shanghaitech.edu.cn
- Student ID:2023234296

Analysis of Original Code Structure

The diagram above shows the source code structure of the simulator, with each file primarily responsible for the functionality indicated by its filename.

- Module BranchPredictor is responsible for branch prediction.
- Module Cache is responsible for implementing the cache.
- Module MainCache is responsible for tracking cache execution and outputting miss rates under various cache configurations.
- Module MainCacheOptimization is responsible for implementing optimizations for the cache.
- Module Debug is responsible for implementing debug functionality.
- Module MemoryManager is used to manage memory.
- Module Simulator is the main implementation of the simulator, including instruction handling, simulation of the pipeline, and other related functionalities.

Because the main focus of this lab is to modify the simulator modules, further analysis will be conducted on their implementation.

Instruction Translation and Implementation

During the fetch stage, instructions are retrieved from memory. In RISC-V, instructions are fetched in 4 bytes units . This means that the fetch operation retrieves instructions in chunks of 4 bytes at a time.

In addition, it is important to check whether the program counter (PC), which holds the address of the next instruction, is aligned. Alignment refers to whether the address is divisible by the size of the instruction being fetched. In this case, the alignment check ensures that the PC is divisible by 4, indicating that the instruction is being fetched on a 4-byte boundary.

During the instruction decoding stage, the binary representation of the instruction is parsed and interpreted according to the RISC-V assembly format. The decoding process involves identifying the opcode, determining the operands, and understanding the specific operation and instruction type.

The decoded instruction is then stored in specific variables or data structures for further processing and execution. These variables typically hold information such as the opcode, source and destination registers, immediate values, and other relevant data associated with the instruction.Branch prediction also plays a role in this stage.

The execution stage implements the corresponding computations based on the functionality of each instruction, and it also checks for data hazards, control hazards, and memory access hazards, taking appropriate actions such as data forwarding or inserting bubbles.

During the memory access stage of the pipeline, instructions that require memory access, such as load and store instructions, are executed. This stage involves reading data from memory or writing data to memory locations.

Similar to the execution stage, the memory access stage also checks for data hazards caused by dependencies between instructions. If a data hazard is detected, data forwarding techniques may be employed to forward the required data from the producing instruction to the consuming instruction, allowing for correct execution and avoiding stalls.

However, it is important to note that in terms of data forwarding priority, the execution stage takes precedence over the memory access stage. This means that if there is a conflict between data forwarding from the execution stage and the memory access stage, the data forwarded by the execution stage will be prioritized.

By prioritizing data forwarding from the execution stage, it ensures that the most up-to-date and relevant data is available for instructions executing in the execution stage, reducing the likelihood of pipeline stalls and improving overall performance.

Additionally, when implementing fmadd.s and fmsub.s, we need to read an extra register. After that, we not only need to increment the variables in the pipeline registers, but also need to check the newly added operands to determine if there will be data hazards and provide data forwarding for them.

Storage and Transmission of Data

In this simulator, general-purpose registers are implemented using an array of type int64_t, and pipeline registers are set up to enable data transfer between different stages of the pipeline.The data is uniformly transmitted using a single data type throughout the process, and it is processed according to its specific size only where processing is required. This simplifies data transmission.

Pipeline

The simulator implements a five-stage pipeline and incorporates branch prediction, data forwarding, and stall mechanisms to ensure correct and efficient operation of the pipeline.

Description of My Design

Floating-point Register

Similar to general-purpose integer registers, I have chosen int64_t as my floating-point storage unit. Because the simulator and real processors are different, their execution can be seen as taking place in the same unit. We only need to differentiate between various operations during computation. Therefore, this design is advantageous for simplifying the data transfer process.

Because the same register numbers can represent different registers in integer instructions and floating-point instructions, for the convenience of handling data hazards and other operations later on, I have placed floating-point registers and general-purpose registers in the same array and renumbered the floating-point registers after decoding.

Instruction Implementation

Most floating-point instructions can be implemented by referencing the implementation methods for integers. However, for fmadd.s and fmsub.s instructions, an additional register needs to be read during the decoding stage, and certain pipeline registers need to include additional fields to store more information.However, it is important to note that the same register numbers may have different meanings in floating-point instructions compared to integer instructions, and this distinction needs to be made during the decoding stage.

Using Floating-Point Representation to Utilize int64_t

In order to facilitate transmission, I stored floating-point numbers in the lower 32 bits of int64_t addresses. However, I encountered difficulties when performing calculations on them. With the help of my classmates, teaching assistants, and search engines, I found three solutions:

- Union: use a union data structure that allows overlapping memory storage for different types. You can define a union with an int64_t member and a float or double member. By storing the floating-point number in the float or double member and accessing it through the int64_t member, you can perform calculations on the floating-point value while preserving its memory representation.
- Using inline assembly to move the lower 32 bits of int64_t to a float for calculations.
- Using the memcpy function from the C standard library for data movement.

I ultimately chose to use the first method to implement my calculations.

System Call

This simulator implements system calls using instructions. Although system call parameters are relatively simple, data hazards can occur during this process. The original author correctly handled this situation. However, when we introduce a system call that outputs floating-point numbers, things become more complicated.

In the original system call implementation, the parameters a0 and a7 registers are used, with a0 holding the data to be processed and a7 holding the system call number. However, when a7 is equal to 6, indicating a floating-point instruction, the data to be processed will be placed in the fa0 register by the compiler.

Therefore, when a data hazard occurs with a7, the previous instruction must modify the data it operates on at the same time it updates the value of a7, in order to ensure correct output. Otherwise, the first floating-point print instruction will not work correctly, while the subsequent floating-point print instructions may work correctly.

To handle this situation properly, the simulator needs to detect when a system call involves floating-point instructions and appropriately handle the data dependencies and register mappings to ensure the correct execution and output of the system call. This may involve additional checks, modifications to the pipeline stages, or other techniques to handle the complexity introduced by the mixed use of integer and floating-point instructions in system calls.

```
1     if(dRegNew.inst==ECALL && out==6 && flag==1){
2       this->dRegNew.op1=reg[REG_FA0];
3       this->dRegNew.rs1+=32;
4     }
```

The code above shows the conditional statement I added after the forwarding stage to handle this issue. Although this approach may not be the most elegant solution.

getComponentUsed()

The latency classification for each instruction is as follows:

- FLW: 1 cycle (dataMem)
- FSW: 1 cycle (dataMem)
- FADD_S: 3 cycles (fmaAdd)
- FSUB_S: 3 cycles (fmaAdd)
- FMUL_S: 6 cycles (fmaMul)
- FDIV_S: 24 cycles (fpDiv)
- FSQRT_S: 24 cycles (fpDiv)
- FCVT_W_S: 1 cycle (int2FP)
- FMV_X_W: 1 cycle (int2FP)
- FCVT_S_W: 1 cycle (int2FP)
- FMV_W_X: 1 cycle (int2FP)
- FMADD_S: 6 cycles (fmaMul)
- FMSUB_S: 6 cycles (fmaMul)

My Results

The original tests

```
./Simulator ../riscv-elf/helloworld.riscv
./Simulator ../riscv-elf/test_arithmetic.riscv
./Simulator ../riscv-elf/test_syscall.riscv
./Simulator ../riscv-elf/test_branch.riscv
./Simulator ../riscv-elf/quicksort.riscv
./Simulator ../riscv-elf/matrixmulti.riscv
./Simulator ../riscv-elf/ackermann.riscv
```

```
$ ./Simulator ../riscv-elf/helloworld.riscv
Hello, World!
Program exit from an exit() system call
------------ STATISTICS -----------
Number of Instructions: 140
Number of Cycles: 231
Avg Cycles per Instrcution: 1.6500
Branch Perdiction Accuacy: 0.5833 (Strategy: Always Not Taken)
Number of Control Hazards: 23
Number of Data Hazards: 78
Number of Memory Hazards: 1
-----------------------------------
```

```
$ ./Simulator ../riscv-elf/quicksort.riscv
Prev A: 5 3 5 6 7 1 3 5 6 1
Sorted A: 1 1 3 3 5 5 5 6 6 7
Prev B: 100 99 98 97 96 95 94 93 92 91 90 89 88 87 86 85 84 83 82 81 80
79 78 77 76 75 74 73 72 71 70 69 68 67 66 65 64 63 62 61 60 59 58 57 56
55 54 53 52 51 50 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7
6 5 4 3 2 1
Sorted B: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48
 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72
 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96
 97 98 99 100
Program exit from an exit() system call
------------ STATISTICS -----------
Number of Instructions: 103670
Number of Cycles: 144531
Avg Cycles per Instrcution: 1.3941
Branch Perdiction Accuacy: 0.4926 (Strategy: Always Not Taken)
Number of Control Hazards: 7314
Number of Data Hazards: 93856
Number of Memory Hazards: 23398
-----------------------------------
```

test_float.c

```c
#include "lib.h"
int main() {
    float a = 3.14159, b = 2.653,
    float x = a + b;
    print_f(x);
    print_c('\n');
    float y = a - b;
    print_f(y);
```

```
    print_c('\n');
    float z = c * d;
    print_f(z);
    print_c('\n');
    x = d / c;
    print_f(x);
    print_c('\n');
    exit_proc();
}
```

```
# fxiao @ fxiao-XiaoXinPro-16-ARH7 in ~/CS211/lab0/cs211_lab0/CS211/build on git:master x [14:31:57]
$ ./Simulator ../riscv-elf/test_float.riscv

5.794590
0.488590
370.349976
41.149998
Program exit from an exit() system call
------------ STATISTICS ----------
Number of Instructions: 255
Number of Cycles: 447
Avg Cycles per Instrcution: 1.7529
Branch Perdiction Accuacy: 0.3846 (Strategy: Always Not Taken)
Number of Control Hazards: 40
Number of Data Hazards: 147
Number of Memory Hazards: 10
```

In the assembly file generated from compiling this test code, there are floating-point instructions such as

> flw, fsw, fadd.s, fsub.s, fmul.s, fdiv.s, and ecall for floating-point number printing.

test_float_fcvt.c

```
#include "lib.h"
int main()
{
    int x = 1;
    print_d(x);
    print_c('\n');
    float a = x;
    float b = 2;
    float c = 3;
    float out = a * b + c;
    print_f(out);
    print_c('\n');
    out = c - a * b;
    print_f(out);
    print_c('\n');
    x = (int)out;
    print_d(x);
    print_c('\n');
    exit_proc();
}
```

```
11/lab0/cs211_lab0/CS211/build on git:master x [15:00:07]
$ ./Simulator ../riscv-elf/test_float_fcvt.riscv
1
5.000000
1.000000
1
Program exit from an exit() system call
------------ STATISTICS ----------
Number of Instructions: 250
Number of Cycles: 437
Avg Cycles per Instrcution: 1.7480
Branch Perdiction Accuacy: 0.5000 (Strategy: Always Not Taken)
Number of Control Hazards: 38
Number of Data Hazards: 136
Number of Memory Hazards: 12
----------------------------------
```

In the assembly file generated from compiling this test code, there are floating-point instructions such as

> fcvt.w.s ,fcvt.s.w

test_float_asmMOP.c

```c
#include "lib.h"

int main() {
    float out;
    float intout;
    int in=0x3F800000;
    float a = 2.0f;
    float b = 3.0f;
    float c = 4.0f;
    asm volatile (
        "fmadd.s %0, %1, %2, %3\n'
        : "=f" (out)
        : "f" (a), "f" (b), "f" (c
    );
    print_f(out);
    print_c('\n');
    asm volatile (
        "fmsub.s %0, %1, %2, %3\n'
        : "=f" (out)
        : "f" (a), "f" (b), "f" (c
    );
    print_f(out);
    print_c('\n');
    asm volatile (
        "fsqrt.s %0, %1\n"  // out
        : "=f" (out)
        : "f" (c)
    );
    print_f(out);
    print_c('\n');
    asm volatile (
        "fmv.x.w %0, %1\n"
        : "=r" (intout)
        : "f" (a)
    );
    print_d(intout);
    print_c('\n');
    asm volatile (
```

```
7 in ~/CS211/lab0/cs211_lab0/CS21
1/build on git:master x [17:46:08
]
$./Simulator ../riscv-elf/test_f
loat_asmMOP.riscv
10.000000
2.000000
2.000000
2
1.000000
Program exit from an exit() syste
m call
------------ STATISTICS ---------
--
Number of Instructions: 383
Number of Cycles: 677
Avg Cycles per Instrcution: 1.767
6
Branch Perdiction Accuacy: 0.5556
 (Strategy: Always Not Taken)
Number of Control Hazards: 56
Number of Data Hazards: 208
Number of Memory Hazards: 12
-------------------------------
--
```

```
        "fmv.w.x %0, %1\n"
        : "=f" (out)
        : "r" (in)
    );
    print_f(out);
    print_c('\n');
    return 0;
}
```

In the assembly file generated from compiling this test code, there are floating-point instructions such as

> fmadd.s f.msub.s fsqrt.s fmv.x.w fmv.w.x