

CompSys - A5

Christian Bendix Fjordstrøm(wjx230),
William Wagner Lauritzen(mwg933),
Hold 2

December 18, 2022

1 Introduction

In this assignment, we have created a 32-bit RISC-V CPU emulator. This report describes our emulator, detailing our not only our implementation and associated testing, but also the shortcomings thereof. Also included is a user guide describing how to run the emulator.

2 User's Guide

In order to run the emulator, navigate to the **src** folder and run

```
$ make rebuild
```

Programs of the format **.dis** can then be run with the command

```
$ ./sim program_name.dis
```

The **src** folder includes four programs on which our implementation has been tested: **hello.dis**, **echo.dis**, **fib.dis**, **erat.dis**. In order to generate a log file that details the sequential execution of a program, and the register states after each instruction, run the command

```
$ ./sim program_name.dis -l log.txt.
```

The resulting log file contains the raw instructions and data of the program in the header, and below the actual log generated over the course of the execution of the program.

3 Implementation Scope

Per the assignment text, we have not implemented a *bona fide* simulator; that is to say, not every element of a RISC-V datapath is necessarily represented as an object in our implementation, and in some cases, sub-tasks are not carried out in the same order as they would be in the aforementioned datapath. We have, however, tried to model a RISC-V datapath fairly faithfully; theoretically, this should reduce implementation overhead relative to hard-coding each instruction, and make our implementation more agile. One of the strengths of RISC-V, for instance, is the ability to resolve pseudo-instructions; with our implementation, these can be executed without having been hard-coded.

Every instruction in the RISC-V 32 bit base instruction set, as well as the multiplication standard extension¹, have been implemented, save for the instructions **ebreak**, **fence**, **fence.i**, **CSRRLW**, **CSRRLS**, **CSRRLC**, **CSRRLWI**, **CSRRLSI** and **CSRRLCI**. The instruction **ecall** has been implemented with the specifications detailed in the assignment text.

¹See <https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf> for details.

4 Implementation

Whereas our implementation is not a perfect simulation of a RISC-V processor, we have attempted to roughly model the datapath thereof, as mentioned earlier. To this end, the execution of our `simulate` function carries out the following tasks, in order:

1. Fetching instruction
2. Decoding instruction and setting control signals
3. Generating immediate based on function and its opcode
4. Selecting ALU action based on control signals
5. Selecting ALU inputs based on control signals
6. Writing from memory to register (if relevant)
7. Writing ALU result to memory (if relevant)
8. Write-back to register (if relevant)
9. Updating PC based on branch control signal and write data

Our implementation makes extensive use of a helper function called `get_insn_field`, whose code can be seen below. The function uses a bit-mask to extract a desired,

```
65 int get_insn_field(int insn, int end, int start) {
66     int length = end-start+1;
67     int mask = power(2, length)-1;
68     return mask & (insn >> start);
69 }
```

contiguous sequence of bits out of an instructions binary encoding. This function is used initially to fetch the sub-fields of the instruction, as demonstrated below. An improvement on this particular implementation may have been to set

```
514 while (1) {
515     fprintf(log_file, "\n Instruction %d.\nPC = 0x%x\n", looping, PC);
516     // fetch instruction
517     int insn = memory_rd_w(mem, PC);
518     int opcode = get_insn_field(insn, 6, 0);
519     int rd = get_insn_field(insn, 11, 7);
520     int funct3 = get_insn_field(insn, 14, 12);
521     int rs1 = get_insn_field(insn, 19, 15);
522     int rs2 = get_insn_field(insn, 24, 20);
523     int funct7 = get_insn_field(insn, 31, 25);
524     const char* insn_a = assembly_get(as, PC);
525     fprintf(log_file, "Instruction: %s\n", insn_a);
526 }
```

these fields via a function, as we have done for our control signals. These have been implemented as integers, for the sake of simplicity, and are set with the `set_signals` function. As seen above, control signals are initialized to zero, and

```

538     int Branch = 0;
539     int MemRead = 0;
540     int MemToReg = 0;
541     int ALUOp0 = 0;
542     int ALUOp1 = 0;
543     int MemWrite = 0;
544     int ALUSrc = 0;
545     int RegWrite = 0;
546     enum bitsize s = word;
547
548     // decode instruction, set signals
549     set_signals(opcode, funct3, &Branch, &MemRead, &MemToReg, &ALUOp0, &ALUOp1, &MemWrite, &ALUSrc, &RegWrite, &s, log_file);
550

```

thus only control signals that have to be asserted are assigned the value 1 by `set_signals`. In the function, signals are generated on the basis of their opcode; see the figure below for example. Notice that JAL and JALR (in addition to a few other unseen instructions) have their own cases. We also use `set_signals` for

```

92     case JAL:
93         *ALUSrc = 1;
94         *RegWrite = 1;
95         break;
96
97     case JALR:
98         *Branch = 1;
99         *RegWrite = 1;
100        break;
101
102     case B:
103         *Branch = 1;
104         *ALUOp1 = 1;
105        break;

```

to assert whether the 'S' and 'L' (save and load) type instructions should save or load bytes, half-words or words (or unsigned equivalents in the case of 'L' type functions).

Thereafter, `get_imm_gen` is used to generate the relevant immediate, based on the opcode it is supplied with. 'R' format instructions return the value '-1' when this function is called. The function uses rather laborious bit-masking and bit-shifts to extract the desired immediates in accordance with the RISC-V specification, and the function likely could have been refactored to be more elegant. Consider the code snippet below, which generates the immediate for 'B' format instructions. Note that an if-statement assures that the case where the immediate is negative is handled through sign-extension. After getting the immediate, the `ALU_control` function is called. The resulting integer is used as input for the ALU itself, and determines which action it will execute. The function is implemented as a series of switches on the sub-fields extracted from

```

216     case B:
217         imm_0 = 0;
218         imm_11 = (get_insn_field(insn, 7, 7) << 11);
219         imm_4_to_1 = (get_insn_field(insn, 11, 8) << 1);
220         imm_10_to_5 = (get_insn_field(insn, 30, 25) << 5);
221         imm_12 = (get_insn_field(insn, 31, 31) << 12);
222         imm = (imm_12 | imm_11 | imm_10_to_5 | imm_4_to_1 | imm_0);
223         if((imm_12 >> 12) == 1){
224             return imm = (imm | 0b11111111111111111111000000000000);
225         }else{
226             return imm;
227         }
228     }

```

the instruction earlier; first a switch on the opcode to get the instruction format, and then switches on the funct3 and funct7 fields as necessary. See for example the code snippet getting the ALU input for 'I' type instructions. Before we can

```

else if (ALUOp1 == 1 && ALUOp0 == 0)
{
    if(opcode == I){
        fprintf(log_file, "I-format OPCODE.\n");
        switch (funct3) {
            case 0x0:
                return ALU_ADD;

            case 0x1:
                return ALU_SLL;

            case 0x2:
                return ALU_SLT;

            case 0x3:
                return ALU_SLTU;

            case 0x4:
                return ALU_XOR;

            case 0x5:
                if(bit30 == 0){
                    return ALU_SRL;
                } else{
                    return ALU_SRA;
                }

            case 0x6:
                return ALU_OR;

            case 0x7:
                return ALU_AND;

            default:
                return 0;
        }
    }
}

```

call the ALU execution function with this control signal, we need to determine which inputs the ALU should take. This is done through a simple if-statement on the `ALUSrc` signal that we determined the value of in `set_signals`. The `ALU_execute` function itself is very straightforward, and simply performs the expected operation on the two inputs provided.

```

544     int ALU_result;
545     if (ALUSrc) {
546         if (opcode == JAL) {
547             ALU_result = ALU_execute(PC, ImmGen, ALU_action, log_file);
548         }
549         else {
550             fprintf(log_file, "Calling ALU execute with inputs x[%d] and Imm.\n", rs1, ImmGen, ALU_action);
551             ALU_result = ALU_execute(x[rs1], ImmGen, ALU_action, log_file);
552         }
553     }
554     else {
555         fprintf(log_file, "Calling ALU execute with inputs: x[%d] and x[%d].\n", rs1, rs2);
556         ALU_result = ALU_execute(x[rs1], x[rs2], ALU_action, log_file);
557     }
558     fprintf(log_file, "ALU result = %d\n", ALU_result);
559

```

Then, a series of if statements on the values of MemToReg, MemWrite and RegWrite determine whether write_data is set to be equal to the ALU result or x[rs2], and where write_data should be written to. Finally, a short code segment handles branching and updating of the PC. This entire sequence of instructions is run in

```

641     // update PC depending on branch and ALU result
642     if (opcode == JAL) {
643         fprintf(log_file, "JAL jump.\n");
644         PC = write_data - 4;
645     } else if (opcode == JALR){
646         PC = write_data - 4;
647         fprintf(log_file, "JALR jump.\n");
648     }
649     if (Branch == 1){
650         if (ALU_result == 0){
651             fprintf(log_file, "Branch = 1, ALU = 0. Taking branch. PC += ImmGen.\n");
652             PC += ImmGen;
653         } else {
654             fprintf(log_file, "Branch = 1, ALU != 0. Skipping branch. PC += 4.\n");
655             PC += 4;
656         }
657     }
658     else {
659         PC += 4;
660     }

```

a while(1) loop, and will not terminate until the instruction ECALL is executed with value 3 or 93. Note that ECALL has been hard-coded as a function located near the beginning of the `simulate.c` file.

5 Testing

The implementation has been tested on all four supplied tests, and it executes each one correctly. To verify, run the commands

```
$ ./sim hello.dis
```

```
$ ./sim echo.dis
```

```
$ ./sim fib.dis
```

```
$ ./sim erat.dis
```

while in the `src` folder. Unfortunately, the `fib.dis` program returns a segmentation fault when run without generating an associated log file; when a log file is generated, the program executes without error. Since this behaviour emerged only shortly before the assignment deadline, it was not possible for us to correct. In general, the tests should be run with an associated log file, as per the example below, in order to ensure correct execution:

```
$ ./sim hello.dis -l log.txt.
```

Due to the amount of text printed to the log file, this means that execution of large programs (`erat.dis`, `fib.dis` called with a large number) results in a large log file, and long execution times.

The tests have been included in the folder itself for convenience. Note that the `erat.dis` program takes a while to execute. Below, screenshots of program outputs to the console have also been provided as verification.

When run with log file generation, we have not observed any functions that



```
william@william-virtual-machine: ~/compSys-e2022-pub/assignments/A5/A5/A5/src
|
|                                     unsigned int *
|                                     %x
gcc -g -Wall -Wextra -pedantic -std=gnu11 main.c *.o -o sim
william@william-virtual-machine:~/compSys-e2022-pub/assignments/A5/A5/A5/src$ ./sim hello.dis -l log.txt
Hello from RISC-V
william@william-virtual-machine:~/compSys-e2022-pub/assignments/A5/A5/A5/src$ ./sim echo.dis -l log.txt
Hej med dig - skriv noget, saa bliver jeg glad: verification that our emulator works
Du skrev: verification that our emulator works
william@william-virtual-machine:~/compSys-e2022-pub/assignments/A5/A5/A5/src$ ./sim fib.dis -l log.txt
Beregn fib() af hvad? 10
Beregner fib(10) = 55
william@william-virtual-machine:~/compSys-e2022-pub/assignments/A5/A5/A5/src$
```

do not exhibit correct behaviour when simulated on our implementation. However, due to time constraints, we have not manually created `.dis` files to test the execution of every single instruction in the RISC-V base instruction set and 'M' extension module. It is therefore possible that the emulator, if run on a custom-made `.dis` program which uses functions not included in the four test files specified above, may have erroneous behaviour.

```
william@william-virtual-machine: ~/compSys-e2022-pub/assignments/A5/A5/A5/src
william@william-virtual-machine:~/compSys-e2022-pub/assignments/A5/A5/A5/src$ ./sim erat.dis -l log.txt
Printal: 1 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101 103 107 109 113 127
131 137 139 149 151 157 163 167 173 179 181 191 193 197 199 211 223 227 229 233 239 241 251 257 263 269 27
1 277 281 283 293 307 311 313 317 331 337 347 349 353 359 367 373 379 383 389 397 401 409 419 421 431 433
439 443 449 457 461 463 467 479 487 491 499 503 509 521 523 541 547 557 563 569 571 577 587 593 599 601 60
7 613 617 619 631 641 643 647 653 659 661 673 677 683 691 701 709 719 727 733 739 743 751 757 761 769 773
787 797 809 811 821 823 827 829 839 853 857 859 863 877 881 883 887 907 911 919 929 937 941 947 953 967 97
1 977 983 991 997 1009 1013 1019 1021 1031 1033 1039 1049 1051 1061 1063 1069 1087 1091 1093 1097 1103 110
9 1117 1123 1129 1151 1153 1163 1171 1181 1187 1193 1201 1213 1217 1223 1229 1231 1237 1249 1259 1277 1279
1283 1289 1291 1297 1301 1303 1307 1319 1321 1327 1361 1367 1373 1381 1399 1409 1423 1427 1429 1433 1439
1447 1451 1453 1459 1471 1481 1483 1487 1489 1493 1499 1511 1523 1531 1543 1549 1553 1559 1567 1571 1579 1
583 1597 1601 1607 1609 1613 1619 1621 1627 1637 1657 1663 1667 1669 1693 1697 1699 1709 1721 1723 1733 17
41 1747 1753 1759 1777 1783 1787 1789 1801 1811 1823 1831 1847 1861 1867 1871 1873 1877 1879 1889 1901 190
7 1913 1931 1933 1949 1951 1973 1979 1987 1993 1997 1999 2003 2011 2017 2027 2029 2039 2053 2063 2069 2081
2083 2087 2089 2099 2111 2113 2129 2131 2137 2141 2143 2153 2161 2179 2203 2207 2213 2221 2237 2239 2243
2251 2267 2273 2281 2287 2293 2297 2309 2311 2333 2339 2341 2347 2351 2357 2371 2377 2381 2383 2389 2
393 2399 2411 2417 2423 2437 2441 2447 2459 2467 2473 2477 2503 2521 2531 2539 2543 2549 2551 2557 2579 25
91 2593 2609 2617 2621 2633 2647 2657 2659 2663 2671 2677 2683 2687 2689 2693 2699 2707 2711 2713 2719 272
9 2731 2741 2749 2753 2767 2777 2789 2791 2797 2801 2803 2819 2833 2837 2843 2851 2857 2861 2879 2887 2897
```

Figure 1: Snippet of output from erat

```
william@william-virtual-machine: ~/compSys-e2022-pub/assignments/A5/A5/A5/src
william@william-virtual-machine:~/compSys-e2022-pub/assignments/A5/A5/A5/src$ ./sim fib.dis -l log.txt
Beregn fib() af hvad? 1
Beregner fib(1) = 1
william@william-virtual-machine:~/compSys-e2022-pub/assignments/A5/A5/A5/src$ ./sim fib.dis -l log.txt
Beregn fib() af hvad? 5
Beregner fib(5) = 5
william@william-virtual-machine:~/compSys-e2022-pub/assignments/A5/A5/A5/src$ ./sim fib.dis -l log.txt
Beregn fib() af hvad? 8
Beregner fib(8) = 21
william@william-virtual-machine:~/compSys-e2022-pub/assignments/A5/A5/A5/src$ ./sim fib.dis -l log.txt
Beregn fib() af hvad? 20
Beregner fib(20) = 6765
william@william-virtual-machine:~/compSys-e2022-pub/assignments/A5/A5/A5/src$ ./sim fib.dis -l log.txt
Beregn fib() af hvad? 24
Beregner fib(24) = 46368
william@william-virtual-machine:~/compSys-e2022-pub/assignments/A5/A5/A5/src$
```

Figure 2: Sample outputs from fib.dis

6 Shortcomings

As mentioned, there are sections of our implementation that could benefit from being refactored. Some functionality in `simulate` could also perhaps benefit from being implemented as functions, such as the fetching of sub-fields, and the handling of branches. In addition, the implementation could certainly have been more concise; similarly, the log file output could also have been more structured and concise. Due to the amount of text output, the log file grows very large in size when generated for the `erat.dis` program. More systematic testing of inputs to the `echo.dis` and `fib.dis` files could also have been performed, although we, through log file inspection, noted that the programs will only output ASCII characters, and therefore have no reason to suspect that there is any possibility for erroneous behaviour. Finally, we would have benefited from creating custom-made `.dis` files to ensure compliance with the totality of the

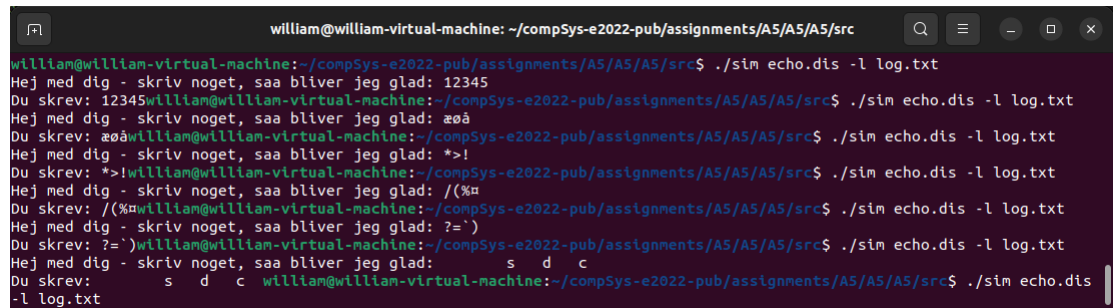
A terminal window titled 'william@william-virtual-machine: ~/compSys-e2022-pub/assignments/A5/A5/A5/src'. The window shows a series of commands and their outputs. The command './sim echo.dis -l log.txt' is run multiple times. The outputs are: 'Hej med dig - skriv noget, saa bliver jeg glad: 12345', 'Du skrev: 12345', 'Hej med dig - skriv noget, saa bliver jeg glad: æøå', 'Du skrev: æøå', 'Hej med dig - skriv noget, saa bliver jeg glad: *>!', 'Du skrev: *>!', 'Hej med dig - skriv noget, saa bliver jeg glad: /(%#', 'Du skrev: /(%#', 'Hej med dig - skriv noget, saa bliver jeg glad: ?=')', 'Du skrev: ?=')', 'Hej med dig - skriv noget, saa bliver jeg glad: s d c', 'Du skrev: s d c', and finally 'william@william-virtual-machine:~/compSys-e2022-pub/assignments/A5/A5/A5/src\$./sim echo.dis -l log.txt'.

Figure 3: Sample outputs from echo.dis

RISC-V 32-bit base instruction set and 'M' extension set, as noted.