Danmarks Tekniske Universitet

DTU

02155 - Computer Architecture and Engineering Fall 2019

Assignment 3

Authors

Sumanth Varambally - s191562
M V A Suhas Kumar - s191382

November 2019

Contents

1 Introduction

In this assignment we implement a simulator for a subset of the **RISC-V** instruction set, namely the integer instruction set **RV32I**, using the programming language C.

The simulator virtually runs all the instructions in software, and emulates all real processor states like **program counter(PC)**, **32 registers** and **memory** to hold instructions and data.

When a program is executed, the simulator performs the following operations:

- Reads the binary file from disk and copies it to a temporary buffer.
- Copies the instructions from the temporary buffer to the beginning (top) of RAM.
- Initialises the program counter to the address of the first instruction, and decodes and executes the instructions like a single clock cycle processor.
- The instruction opcode is first decoded to determine which type of instruction it is, and then the appropriate 'extract' function is used to extract the specifics of the instruction like the operation to be performed, source and destination registers and offsets.
- The corresponding operation is performed. The appropriate registers are updated in the register file, the program counter (PC) is updated (as need may be), and the contents of memory are changed as required.
- After execution of the program, the simulator stores the values of the 32 registers into `dump.res`.

All the general purpose registers `x0-x31` are 32-bits long and hold the values that are interpreted by the instructions as 2's complement signed binary integers. We also ensure that the register `x0` is enforced to 0. We simulate the program counter (PC) which stores the address of current instruction in execution. It is implemented as a C pointer.

2 Description of Source Files

We have divided the source code of the simulator into multiple files in order to better organise it and isolate different tasks. The main source files are as follows:

- `main.c`: The file that contains the main function. It contains code to read the instructions from disk, copy them to memory, initialize the program counter and registers, and run the main loop, which executes every instruction in memory.
- `instruc_exec.c`: The file that contains code to execute the different instructions. It uses a switch statement to distinguish between the different opcodes. The opcode and arguments are extracted from the input instruction using appropriate bit-masks and shifts (using functions from `instruc_utils.c` and the correct corresponding operation is performed.
- `instruc_utils.c`: The file that contains utility functions used for the extraction of the appropriate source, destination and functionality from the 32-bit instruction.

3 Design and Implementation of the simulator

When the simulator is run, memory is allocated for the programs being run in the simulator. By default, the memory size used is 32 Megabytes. Further, the binary code is copied onto the top of memory. The stack pointer is initialized to point 1 Megabyte from the top. This effectively means that the stack has a size of (1MB).

3.1 Decoding the instruction

In the decoding stage we first extract the opcode and then depending on the type of instruction, various other values. We use bit-shifting and bitwise operations with appropriate bitmasks to extract the required fields from the instruction. To explain how we performed decoding, we illustrate with an example

Example: We consider the decoding of an R-type instruction.
Instruction: `add x15,x14,x15`
Binary representation of instruction: `0000 0000 1111 0111 0000 0111 1011 0011`
Fields:

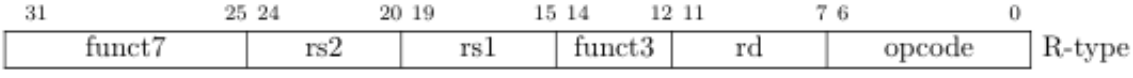

Figure 1: Fields in R-type instruction (source: RISC-V Instruction Set Manual)

Extraction:

```
void r_type_extract(uint32_t instruc, uint8_t* rs1, uint8_t* rs2, uint8_t* rd, uint8_t* funct7, uint8_t* funct3)
{
    *rs1 = (instruc >> 15) & 0x1f;
    *rs2 = (instruc >> 20) & 0x1f;
    *rd = (instruc >> 7) & 0x1f;
    *funct7 = (instruc >> 25) & 0x7f;
    *funct3 = (instruc >> 12) & 0x07;
}
```

Figure 2: Extracting R type instruction

We illustrate the extraction of the value of `rd`, which is located in bits 8 to 12 in the instruction. In the above figure we see that to extract `rd` we first shifted the instruction to the right by 7 bits and used the bitwise AND operation with the bit mask `0x1f` to get the last five bits.

Binary format of instruction: `0000 0000 1111 0111 0000 0111 1011 0011`
Right shifted output: `0000 0000 0000 0001 1110 1110 0000 1111`
Output after masking: `0000 0000 0000 0000 0000 0000 0000 1111`, which is 15.
We obtain the value of 15 for `rd` which is what we expect for the given instruction.

In the similar manner we extracted for other types of Instructions.

3.2 Execution of instructions

After decoding the instruction and obtaining the required fields, we next implement the operation execution based on the instruction using C operators. First, the type of instruction is determined using the opcode. The opcode is obtained by performing the bitwise AND operation with the bit-mask `0x7f`. Then depending on the type of instruction, the appropriate operations are implemented.

3.2.1 R-type instructions

For these instruction we extract the values of `rs1`, `rs2`, `rd`, `funct3` and `funct7`. Here, based on the values of `funct3` and `funct7`, we operate on the values stored in register `rs1` and `rs2` and store the output in the location `rd`. For the unsigned instructions like `sltu` we cast the the value of the operands as unsigned integers using the inbuilt C casting operation.

3.2.2 I-type instructions

For the I-Type instructions, we extract the values of `rs1`, `rd`, `funct3` and `imm`. The exact operation is determined used the value of `funct3`.

3.2.3 U-type instructions

The instructions implemented here are AUIPC and LUI. The values of `rd` and `imm` are extracted for the computation. While LUI sets the lower 12bits of the `rd` register, AUIPC sets the `rd` to the appropriate address.

3.2.4 Load instructions

The Load instructions are encoded as I-Type instructions, and hence we extract the values of `rs1`, `rd`, `funct3` and `imm` in a similar fashion. The value stored in memory is extracted from the source address and stored in the target register after bit shifting and bit masking, according to the size (load word vs load half-word vs load byte)

3.2.5 S-type instructions

The store insturctions are encoded as S-type instructions, and we extract `rs1`, `rs2`, `funct3` and `imm` fields. The correct destination in memory is calculated, and the value from the source register is stored in memory.

3.2.6 J-type instructions

The J-type instruction is used to implement the `jal` instruction. The `rd` and `imm` fields are read and an unconditional jump to the appropriate address is performed by adding the correct offset to the `pc`. The address of the next instruction after the jump is also written to the `rd` register.

3.2.7 jalr instruction

The `jalr` instruction is implemented as an I-type instruction, and hence `rs1`, `rd`, `funct3`, `imm` are extracted. The address of the next instruction following the current instruction is written to `rd`.

3.2.8 B-type instruction

The branch instructions are implemented as B-type instructions, and the appropriate fields `rs1`, `rs2`, `funct3`, and `imm` are extracted. Depending on the value of `funct3`, the appropriate type of branch instruction is chosen. The two source registers are compared and depending on the result of the operation, the appropriate address of the next instruction is written to the `pc` register. Note that the immediate values are expressed in multiples of 2 bytes, and hence this has to be accounted for while calculating the correct address for the next instruction.

3.2.9 Environment Calls

A few environment calls are also implemented:

ID(a0)	Name	Description
10	`exit`	Ends the program
1	`print_int`	print intergers in a1
4	`print_string`	prints the null-terminated string whose address is in a1
11	`print_character`	prints ASCII character in a1
17	`exit_2`	ends the program with return code in a1

3.3 Memory layout:

3.3.1 General purpose registers:

In an actual RISC-V processor, general purpose registers are stored in the CPU. In order to simulate the same situation in the simulator, we allocated a separate space of 128 bytes other than the memory used for real memory operations.

3.3.2 Memory layout for program:

In the simulator we allocated a total space of 32MB RAM for the microprocessor. We allocated memory for mainly text (or code), data and stack. The order of allocation would be i.e starting from address 0 to the text block. Data can be stored anywhere in the memory other than text or stack. Due to the large size of memory, there is very less possibility for collision and stack pointer is initialised to a high value to have sufficient space for stack.

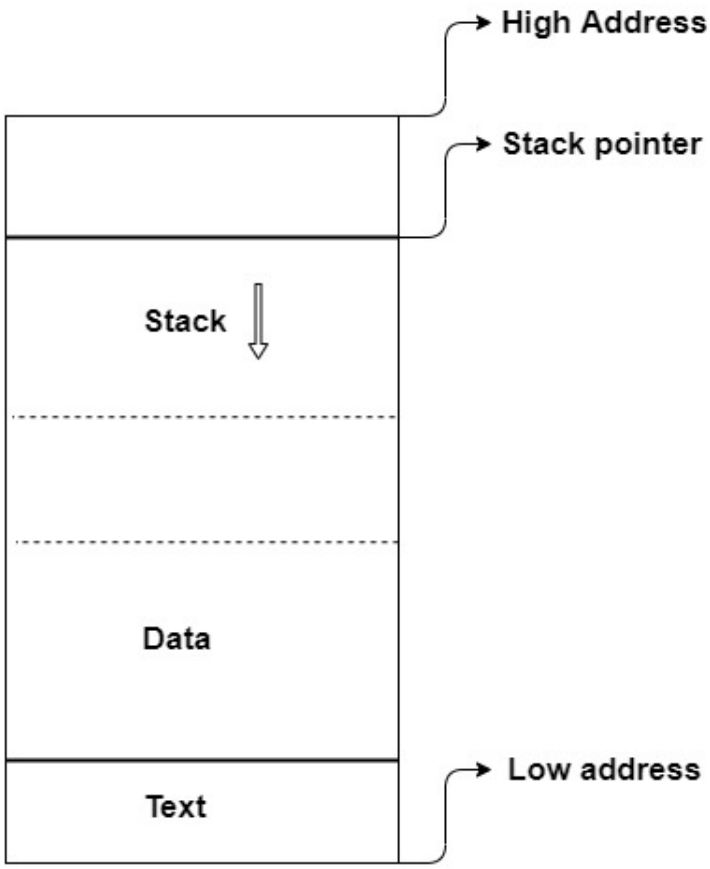

Figure 3: Memory layout

4 Building and running the simulator

To build the simulator, the `gcc-multilib` library is required, since the simulator is run as a 32-bit binary. Once the libary is installed, run `make`.
To run a binary file, type `./risc-simul <binary file>`
Any outputs, as well as the final register states are printed. A binary dump of the final register contents are also stored in `dump.res`

5 Appendix: Source code

5.1 main.c:

```c
#include <stdio.h>

//for fixed size data types
#include <stdint.h>
#include <stdlib.h>
#include "instruc_utils.h"
#include "instruc_exec.h"
#define MAX_BUFFER_SIZE 1024*1024
#define MAX_INSTRUCTIONS MAX_BUFFER_SIZE / 4
//32 MB
#define MEMORY_SIZE 32 * 1024 * 1024

int main(int argc, char const *argv[])
{
    uint32_t *instructions = NULL;
    int n_instructions=0;

    if(argc != 2)
    {
        printf("ERROR: Improper usage.\n");
        printf("Syntax: riscsimul <program name>\n");
        return -1;
    }
    else
    {
        //open the file
        FILE *fb;
        fb = fopen(argv[1], "rb");
        if(fb==NULL)
        {
            printf("ERROR: Could not open file %s\n", argv[1]);
            return -2;
        }
        //read all the instructions
        instructions = malloc(MAX_BUFFER_SIZE);
        while(fread(&instructions[n_instructions++], 4, 1, fb) == 1)
        {
            //do nothing
```

```
39          }
40          n_instructions -= 1;
41      }
42
43      int32_t *pc = 0;
44      int32_t registers[32];
45
46
47      int32_t *ram = (int32_t*)(calloc(MEMORY_SIZE/4, 4));
48      //initialise registers to 0
49      for (int i = 0; i < 32; i++)
50          registers[i] = 0;
51
52      //initialize stack pointer such that we have 1MB stack
53      registers[2] = (int32_t) (ram + (MEMORY_SIZE - 1024*1024)/4);
54
55      //copy the code to memory
56      for(int i = 0; i < n_instructions; i++)
57      {
58          ram[i] = instructions[i];
59      }
60      free(instructions);
61
62      pc = ram;
63
64      for (; ; )
65      {
66          int32_t instr = *pc;
67          //printf("%s\n", );
68          pc = process_instruction(instr, registers, ram, pc);
69
70          //test for program end
71          if(pc > ram+n_instructions)
72              break;
73          //print_registers(registers);
74          //getchar();
75      }
76      printf("\nPrinting register contents:\n" );
77      print_registers(registers);
78      free(ram);
79      return 0;
80  }
```

5.2 instruc_utils.c

```
1
2  #include "instruc_utils.h"
3  #include <stdio.h>
4
5
6  void print_registers(int32_t *registers)
7  {
8      FILE *write_ptr;
9      for(int i = 0; i < 32; i++)
10     {
11         printf("x%d:\t\t%d\n", i, registers[i]);
12     }
13     write_ptr = fopen("dump.res","wb");  // w for write, b for binary
14     fwrite(registers, 4*32, 1, write_ptr);
15
16 }
17
18 int32_t sign_extend(int32_t num, int bits)
19 {
20     uint32_t left_most = num >> (bits-1);
21     return left_most == 0x01 ? ((0xffffffff<<bits) + num ) : num;
22 }
23
24
25 void r_type_extract(uint32_t instruc, uint8_t* rs1, uint8_t* rs2, uint8_t*
       rd, uint8_t* funct7, uint8_t* funct3)
26 {
27     rs1 = (instruc >> 15) & 0x1f;
28     rs2 = (instruc >> 20) & 0x1f;
29     rd = (instruc >> 7) & 0x1f;
30     funct7 = (instruc >> 25) & 0x7f;
31     funct3 = (instruc >> 12) & 0x07;
32     opcode = instruc&0x7f;
33 }
34
35 void i_type_extract(uint32_t instruc, uint8_t* rs1, uint8_t* rd, uint8_t*
       funct3, int32_t *imm)
36 {
37     *rd = (instruc >> 7) & 0x1f;
38     *funct3 = (instruc >> 12) & 0x07;
39     *rs1 = (instruc >> 15) & 0x1f;
```

```
40      *imm = sign_extend(instruc >> 20, 12);
41  }
42
43  void u_type_extract(uint32_t instruc, uint8_t* rd, int32_t *imm)
44  {
45      *rd = (instruc>>7) & 0x1f;
46      *imm = (instruc >> 12);
47  }
48
49  void b_type_extract(uint32_t instruc, uint8_t* rs1, uint8_t* rs2, uint8_t*
        funct3, int32_t*imm)
50  {
51      *rs1 = (instruc>>15) & 0x1f;
52      *rs2 = (instruc>>20) & 0x1f;
53      *funct3 =(instruc>>12) &0x07;
54      int32_t imm_12 = (((instruc>>31)&0x01)<<11);
55      int32_t imm_11 = (((instruc>>7)&0x01)<<10);
56      int32_t imm_10_5 =(((instruc>>25)&0x3f)<<4);
57      int32_t imm_4_1 = ((instruc>>8)&0x0f);
58      *imm = sign_extend(imm_12+imm_11+imm_10_5+imm_4_1,12);
59  }
60
61  void jalr_type_extract(uint32_t instruc, uint8_t* rs1, uint8_t* rd, uint8_t*
        funct3, int32_t *imm)
62  {
63      *rd = (instruc >> 7) & 0x1f;
64      *funct3 = (instruc >> 12) & 0x07;
65      *rs1 = (instruc >> 15) & 0x1f;
66      *imm = sign_extend(instruc >> 20, 12);
67  }
68
69  void j_type_extract(uint32_t instruc,uint8_t* rd,int32_t* imm)
70  {
71      *rd = (instruc >> 7) & 0x1f;
72      int32_t imm_20 = (((instruc>>31)&0x01)<<19);
73      int32_t imm_19_12 = (((instruc>>12)&0xff)<<11);
74      int32_t imm_11 = (((instruc>>20)&0x01)<<10);
75      int32_t imm_10_1 = ((instruc>>21)&0x3ff);
76      *imm = sign_extend(imm_20 + imm_19_12 + imm_11 + imm_10_1, 20);
77  }
78
79  void load_type_extract(uint32_t instruc, uint8_t* rs1, uint8_t* rd, uint8_t*
        funct3, int32_t* imm)
80  {
```

```
81      *rd = (instruc >> 7) & 0x1f;
82      *funct3 = (instruc >> 12) & 0x07;
83      *rs1 = (instruc >> 15) & 0x1f;
84      *imm = sign_extend(instruc >> 20, 12);
85  }
86
87  void s_type_extract(uint32_t instruc, uint8_t* rs2, uint8_t* rs1, uint8_t*
        funct3, int32_t* imm)
88  {
89      *rs2 = (instruc>>20)&0x1f;
90      *rs1 = (instruc>>15)&0x1f;
91      *funct3 = (instruc>>12)&0x07;
92      int32_t imm_11_5 = (((instruc>>25)&0x7f)<<5);
93      int32_t imm_4_0 = ((instruc>>7)&0x1f) ;
94      *imm = sign_extend(imm_11_5 + imm_4_0, 12);
95  }
```

5.3 instruc_utils.h

```
1   #ifndef INSTRUC_UTILS
2   #define INSTRUC_UTILS
3
4   #include <stdint.h>
5
6   void print_registers(int32_t *registers);
7   void r_type_extract(uint32_t instruc, uint8_t* rs1, uint8_t* rs2, uint8_t*
        rd, uint8_t* funct7, uint8_t* funct3);
8   void i_type_extract(uint32_t instruc, uint8_t* rs1, uint8_t* rd, uint8_t*
        funct3, int32_t *imm);
9   void u_type_extract(uint32_t instruc, uint8_t* rd, int32_t *imm);
10  void b_type_extract(uint32_t instruc, uint8_t* rs1, uint8_t* rs2, uint8_t*
        funct3, int32_t*imm);
11  int32_t sign_extend(int32_t num, int bits);
12  void j_type_extract(uint32_t instruc,uint8_t* rd,int32_t* imm);
13  void load_type_extract(uint32_t instruc, uint8_t* rs1, uint8_t* rd, uint8_t*
        funct3, int32_t *imm);
14  void s_type_extract(uint32_t instruc, uint8_t* rs2, uint8_t* rs1, uint8_t*
        funct3, int32_t* imm);
15
16  #endif
```

5.4 instruc_exec.c

```c
#include "instruc_exec.h"
#include "instruc_utils.h"

int32_t* process_instruction(uint32_t instruc, int32_t *registers, int32_t *
    ram, int32_t *pc)
{
    //extract the opcode from the instruction
    uint32_t opcode = instruc & 0x7f;
    uint8_t rs1, rs2, rd, funct3, funct7;
    int32_t imm;
    switch (opcode)
    {
    case 0x13:  //I type instruction

        i_type_extract(instruc, &rs1, &rd, &funct3, &imm);
        switch (funct3)
        {
        case 0:
            //addi
            registers[rd] = registers[rs1] + imm;
            break;
        case 1:
            //slli

            registers[rd] = (uint32_t)registers[rs1] << (uint32_t)(imm&0x1f)
                ;

            break;

        case 2:
            //slti
            registers[rd] = registers[rs1] < imm;
            break;

        case 3:
            //sltiu
            registers[rd] = (uint32_t)registers[rs1] < (uint32_t)imm;
            break;

        case 4:
            //xori
```

```
40              registers[rd] = registers[rs1] ^ imm;
41              break;
42
43          case 5:
44              //srli/srai
45
46              if((imm>>5) == 0)
47              {
48                  //srli
49                  registers[rd] = (uint32_t)registers[rs1] >> (uint32_t)(imm &
                        0x1f);
50              }
51              else if((imm>>5) == 32)
52              {
53                  //srai
54                  registers[rd] = registers[rs1] >> (imm & 0x1f);
55              }
56              break;
57
58          case 6:
59              //ori
60              registers[rd] = registers[rs1] | imm;
61              break;
62
63          case 7:
64              //andi
65              registers[rd] = registers[rs1] & imm;
66              break;
67          }
68          pc++;
69          break;
70
71      case 0x33: ; //R type instruction
72          r_type_extract(instruc, &rs1, &rs2, &rd, &funct7, &funct3);
73
74
75          switch (funct3)
76          {
77          case 0:
78              //add or sub
79              if(funct7 == 0)
80              {
81                  //add
82                  registers[rd] = registers[rs1] + registers[rs2];
```

```
83                  ;
84              }
85              else if(funct7 == 32)
86              {
87                  //subtract
88                  registers[rd] = registers[rs1] - registers[rs2];
89              }
90              break;
91
92          case 1:
93              //sll
94              //get lower 5 bits of registers[rs2]
95              registers[rd] = (uint32_t)registers[rs1] << (uint32_t)(registers
                    [rs2] & 0x1f);
96              break;
97
98          case 2:
99              //slt
100             registers[rd] = registers[rs1] < registers[rs2];
101             break;
102
103         case 3:
104             //sltu
105             registers[rd] = (uint32_t)registers[rs1] < (uint32_t)registers[
                    rs2];
106             break;
107
108         case 4:
109             //xor
110             registers[rd] = registers[rs1] ^ registers[rs2];
111             break;
112
113         case 5:
114             //srl or sra
115             if(funct7 == 0)
116             {
117                 //srl
118                 registers[rd] = (uint32_t)registers[rs1] >> (uint32_t)(
                        registers[rs2]& 0x1f);
119             }
120             else if(funct7 == 32)
121             {
122                 //sra
123                 registers[rd] = registers[rs1] >> (registers[rs2]& 0x1f);
```

```
124             }
125             break;
126 
127         case 6:
128             //or
129             registers[rd] = registers[rs1] | registers[rs2];
130             break;
131 
132         case 7:
133             //and
134             registers[rd] = registers[rs1] & registers[rs2];
135             break;
136         }
137         pc++;
138         break;
139 
140     case 0x17:
141         ;
142         //auipc
143         u_type_extract(instruc, &rd, &imm);
144         if(rd!=0)
145             registers[rd] = (pc-ram)*4 + (imm << 12);
146 
147         pc++;
148         break;
149 
150     case 0x37:
151         ;
152         //lui
153         u_type_extract(instruc, &rd, &imm);
154 
155         registers[rd] = imm << 12;
156 
157         pc++;
158         break;
159 
160     case 0x63:
161         ;
162         b_type_extract(instruc, &rs1, &rs2, &funct3, &imm);
163         switch(funct3)
164         {
165         case 0:
166             ;
167             //beq
```

```
168
169             if (registers[rs1] == registers[rs2])
170             {
171                 pc = pc + (imm/2);
172                 //The 12-bit B-immediate encodes signed offsets in multiples
                        of 2 bytes.
173                 //Also 4 bytes equal to 1 instruction .
174             }
175             else
176                 pc++;
177             break;
178         case 1:
179             ;
180             //bne
181
182             if (registers[rs1] != registers[rs2])
183             {
184                 pc = pc + (imm/2);
185             }
186             else {
187                 pc++;
188             }
189             break;
190         case 4:
191             ;
192             //blt
193
194             if (registers[rs1] < registers[rs2])
195             {
196                 pc = pc + (imm/2);
197             }
198             else
199             {
200                 pc++;
201             }
202             break;
203
204         case 5:
205             ;
206             //bge
207
208             if (registers[rs1] > registers[rs2])
209                 pc = pc +(imm/2);
210             else
```

```
211                 pc ++;
212             break;
213
214         case 6:
215             ;
216             //bltu
217             if ((uint32_t)(registers[rs1]) < (uint32_t)(registers[rs2]))
218             {
219                 pc =  pc + (imm/2);
220             }
221             else
222                 pc++;
223
224             break;
225
226         case 7:
227             ;
228             //bgeu
229             if ( (uint32_t)(registers[rs1]) > (uint32_t)(registers[rs2]))
230                 pc = pc + (imm/2) ;
231
232             else
233                 pc++;
234             break;
235         }
236         break;
237
238     case 0x67:
239     {   //jalr
240         i_type_extract(instruc, &rs1, &rd, &funct3, &imm);
241         switch(funct3)
242         {
243         case 0:
244             ;
245             if(rd!=0)
246                 registers[rd] = (int32_t)(pc -ram + 1); //storing pointer
                       value in register
247             pc = ram + ((registers[rs1] + imm)&0xfffffffe)/4;
248             break;
249         }
250         break;
251     }
252
253     case 0x6f:
```

```
254     {   //jal (J type instruction)
255         j_type_extract(instruc,&rd,&imm);
256         if(rd!=0)
257             registers[rd] = (int32_t)(pc - ram + 1) * 4; //storing pointer
                    value in register
258         pc = pc + (imm/2);
259         break;
260     }
261 
262     case 0x03:
263         load_type_extract(instruc,&rs1, &rd, &funct3, &imm);
264         switch(funct3)
265         {
266         case 0:
267             //lb
268             registers[rd] = sign_extend((ram[(registers[rs1] + imm)/4])&0xff
                    , 8);
269             pc++;
270             break;
271 
272         case 1:
273             //lh
274             registers[rd] = sign_extend((ram[(registers[rs1] + imm)/4])&0
                    xffff, 16);
275             pc++;
276             break;
277 
278         case 2:
279             //lw
280 
281 
282             registers[rd] = ram[(registers[rs1] + imm)/4] ;
283 
284             pc++;
285             break;
286 
287         case 3:
288             //ld
289             printf("ERROR: Trying to use ld instruction in 32-bit system\n")
                    ;
290             pc++;
291             break;
292 
293         case 4:
```

```
294             //lbu
295             registers[rd] = ram[(registers[rs1] + imm)/4]&0xff;
296             pc++;
297             break;
298 
299         case 5:
300             //lhu
301             registers[rd] = ram[(registers[rs1] + imm)/4] & 0xffff;
302             pc++;
303             break;
304 
305         case 6:
306             //lwu
307             registers[rd] = ram[(registers[rs1] + imm)/4] & 0xffffffff;
308             pc++;
309             break;
310         }
311         break;
312 
313     case 0x23:
314         s_type_extract(instruc, &rs2, &rs1, &funct3, &imm);
315 
316         switch(funct3)
317         {
318         case 0:
319             //sb
320             ram[(registers[rs1] + imm)/4] = registers[rs2]&0xff;
321             pc++;
322             break;
323         case 1:
324             //sh
325             ram[(registers[rs1] + imm)/4] = registers[rs2]&0xffff;
326             pc++;
327             break;
328 
329         case 2:
330             //sw
331             ram[(registers[rs1] + imm)/4] = registers[rs2]&0xffffffff;
332             pc++;
333             break;
334 
335         case 3:
336             //sd
337             printf("ERROR: Trying to use 64bit instruction sd in 32-bit
```

```
                simulator\n");
338             pc++;
339             break;
340         }
341         break;
342
343     case 0x73: ;//ecall
344         switch (registers[10])
345         {
346         case 10:
347             //print registers
348             printf("Exiting the program...printing registers\n");
349             print_registers(registers);
350             exit(0);
351         case 1:
352             //print integer in a1
353             printf("Printing integer in a1: ");
354             printf("%d\n", registers[11] );
355             break;
356         case 4:
357                 // printing string whose adress is a1
358                 printf("Printing string whose adress a1: " );
359                 printf("%s\n",(char *)ram[registers[11]/4] );
360                 break;
361         case 11:
362                 //printing ascii character in a1
363                 printf("Printing ascii character in a1: ");
364                 printf("%c\n",(char)registers[11] );
365                 break;
366         case 17:
367                 //exit2
368             printf("Exiting the program...printing registers\n");
369             print_registers(registers);
370             exit(registers[11]);
371         }
372     default:
373         printf("Unknown instruction with opcode: %u\n", opcode);
374         pc++;
375     }
376
377     return pc;
378 }
```

5.5 instruc_exec.h

```
#ifndef INSTRUC_EXEC
#define INSTRUC_EXEC

#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include "instruc_utils.h"

int32_t* process_instruction(uint32_t instruc, int32_t *registers, int32_t *
    ram, int32_t *pc);

#endif
```

5.6 Makefile

```
CC = gcc
CFLAGS = -std=c99 -m32 -g
SRC_FILES = main.c instruc_utils.c instruc_exec.c
EXEC = riscsimul

all:
        $(CC) $(SRC_FILES) $(CFLAGS) -o $(EXEC)

clean:
        rm $(EXEC) dump.res
```