Danmarks
Tekniske
Universitet

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:

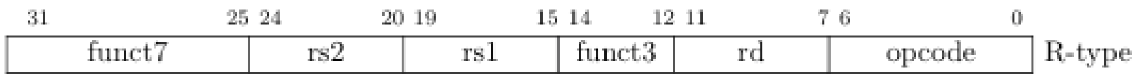
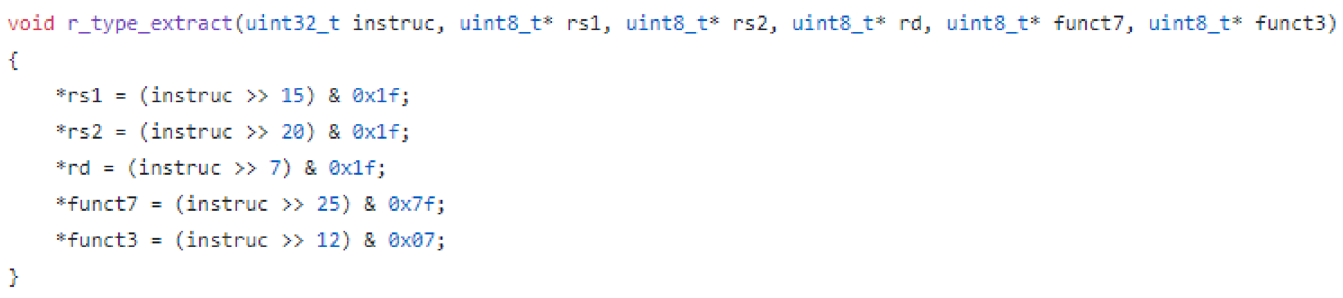


Figure 1: Fields in R-type instruction (source: RISC-V Instruction Set Manual)

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.

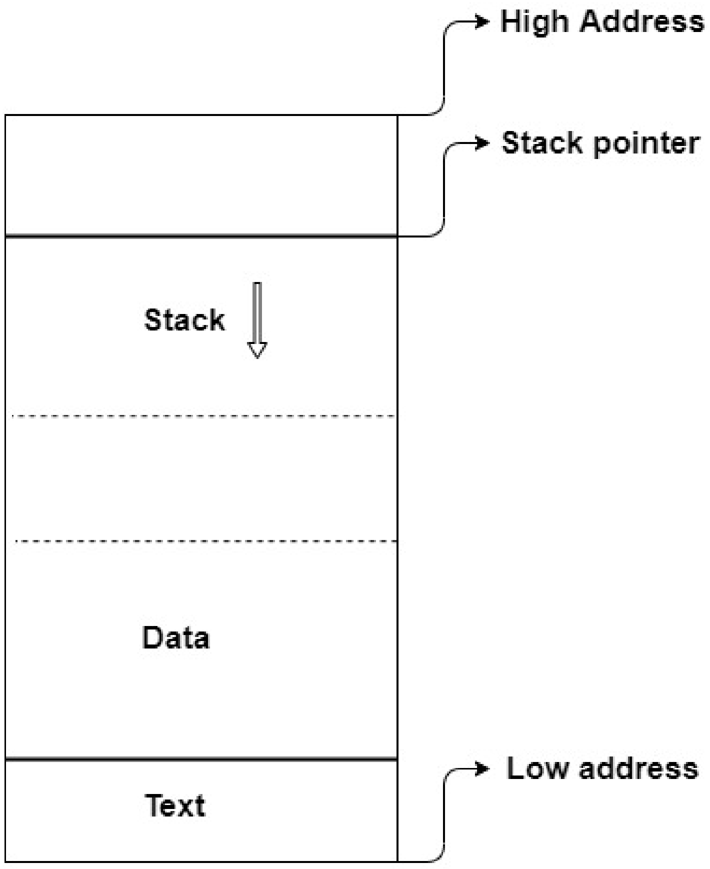


Figure 3: Memory layout

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`