CS204 : Computer Architecture

Course Project

Phase - I

Team Members :

- Tanuj Maheshwari - 2019CSB1125
- Aman Battan - 2019EEB1139
- Solanki Dhruv - 2019CSB1122
- Dhiresh Kumar Singh - 2019CSB1255
- Sudhanshu Kumar - 2019CSB1123

Project Overview

Aim of the Project - To build a simulator for the machine level execution of 32bit RISC-V ISA using a high level language.

The execution of each instruction is completed after going through the five stages as described in the RISC-V architecture.

Language/Technologies Used

The project is entirely in python (version 3), with no external modules/libraries.

Implementation Description

The program takes the input from the .mc file (in the required input format), and the instructions are executed as per their functional behaviour.

The text and the data segments are stored separately, and then, the execution goes through the following steps :

1. **Fetch -** Each instruction is fetched from the memory according to the address in PC, and written into IR. It is also determined if the instruction is the end of execution instruction or a wrong instruction.
2. **Decode -** In this stage, immediate value and control signals are generated. Also, register values are read from the register file and appropriate outputs are generated based on control signals to be given as inputs to ALU.
3. **Execute -** All ALU operations are performed in this stage, including generating effective addresses. Also, the PC is updated in this stage as opposed to the fetch stage because jump/branch instructions are possible only by following this description.
4. **Memory Access -** In this step, Memory is read and written according to the given instruction. The MemoryUnit module consists of both instruction and data memory as a single unit, but they are accessed separately.

5. **Register Update** - Also known as writeback stage, this is where the destination register is updated in the register file with the appropriate value. Also, the clock is incremented by 1 signifying that the instruction is over.

Logical Units

The main logical units/blocks that form the datapath :-

1. **Control Unit** - This module generates the different control signals which are required for working of the datapath module. Then, these control signals are fed as input to the different MUXes. The signals are :-
   - **MuxRs1** - If the select line is 0, return 5 bits corresponding to register number rs1, else if it is 1, return 5 bits corresponding to register rd. (used for implementing lui instruction)
   - **MuxA** - If the select line is 0, it selects value in RA, else it selects the value of PC. (used for implementing auipc instruction)
   - **MuxB** - If the select line is 0, return contents of RB as output, else if it is 1, select the immediate value.
   - **MuxY** - Selects return address if MuxY = 2, else if its value is 1, select data from MDR as input else select RZ(alu output) if it is zero.
   - **MuxPC** - If its value is 1, select PC+4 or PC+imm as input else select RA(return address) as new PC value.

2. **Memory Unit** - This module includes five subfunctions :-
   - **Prog_Load** - This functions loads the complete instructions set in the Memory following the Little Endian set of architecture. Also, it makes the memory byte addressable.
   - **Ins_Load** - This loads individual instructions in the memory. This function also shows error whenever the PC is out of range, or not a multiple of 4.
   - **Mem_write** - This function is used to write something on the memory. Also, the address of memory should be between the range (0x0000000,0x7fffffff). It writes the data provided in the MDR to address in MAR.
   - **Mem_read**[1] - It will read the data available at the address pointed by the MAR and stores that into the MDR for further use.

[1] Note that while loading single byte or halfword into registers, they are not sign extended

- **display_status** - This function is used to print the data available in any part of the memory during the execution of the program. We can also input the address at which we want to see data , provided it lies in the given range.

3. **ALU** - The arithmetic and logic unit is the heart of the execute state. All the arithmetic instructions, like add, sub, mul, div, and, or, etc are conducted by this unit. It uses the basic_functions & ALU_utility_functions modules for working, where each arithmetic instruction is executed in a bit-by-bit fashion (ripple carry adder for add, two's complement for sub, etc). The ALU also provides the control signal for MuxINC to IAG for conditional branch instructions.

4. **IAG** - The Instruction Address Generator unit updates the value of the PC based on control signals given by the control unit and ALU. It also maintains a register PC_temp which is then passed on to MuxY as an input.

5. **Register File** - The register file maintains the registers x0 to x31, and reads/updates their values accordingly. This module is active in the Decode and the WriteBack stage.

• Helper Functions

These function help the main logic functions to do the required tasks-

1. **ALU Functions** - These are the functions required by the ALU unit to perform the ALU operations. It includes functions like bin_add, bin_sub, etc. which takes 2 binary inputs and one boolean input(to check for signed) and gives the required binary output after performing the ALU operation.

2. **Basic Functions -** It contains functions required for basic operations on input data. In includes-
    - **extend_to_32_bits** - It converts the input binary instruction to its 32 bit signed or unsigned extended value.
    - **unsigned_extend** - It performs an unsigned extension.
    - **hex_to_bin -** It converts the hexadecimal instruction into binary string.
    - **twos_complement -** It performs 2's complement of the binary string.
    - **bin_to_dec -** It converts the binary string into decimal number.
    - **dec_to_bin -** It converts decimal to binary string.

3. **MC_to_dict -** This function converts the given instructions.mc file to a dictionary of instructions in which Address (byte-wise) is the key and Machine Code is the value corresponding to key.

I/O Details and Terminal Interface

The input of the simulator would be the **instructions.mc** file with the following format:

<PC> <space> <machine_code_of_instruction> [<space> <comments>]

…

<PC> <space> 0xFFFFFFFF [<space> <comment>]

<blank_line>

<Address> <space> <data_to_be_stored> [<space> <comments>]

…

An example is stated below :-

```
0x0 0x0900006F #comment1

0x4 0x00200293 #comment2

0x8 0x0055D463

...

0xcc 0x00500013

0xd0 0xFFFFFFFF #End of Text segment


0x10000000 0x5 #comment3

0x10000004 0x7

...
```

The instruction **0xFFFFFFFF** would signify the end of the text segment

The data segment of the code would be present before the starting or after the end of the text segment.

After running the program, the instructions would be implemented one-by-one starting from the first instruction.

The current instruction would be displayed and the user would be given a choice to:

1. **Run the complete code and skip to the end**
2. **To step to the next instruction**
3. **Inspect the memory**
4. **Skip a certain number of clock cycles.**

In the terminal, it would be shown as:

```
Current instruction :- 0x10000517

Enter an integer according to your choice below :-
a) Enter 0 to run the code and skip to end
b) Enter 1 to step to next instruction
c) Enter -1 to inspect memory
d) Enter any other integer to skip that many clock cycles

Enter the number :-
```

If the user requests to step to the **next instruction** or **skips some instructions**, then the register file after execution of those instruction(s) would be displayed. For example:

```
Register File status after clock cycle  12  :-

x 0  :-  0x0
x 1  :-  0x0
x 2  :-  0x7ffffff0
    ...
    ...
x 31  :-  0xface
```

If the user wants to run the code and **skip to the end**, then the Final Register File would be displayed along with the total number of clock cycles. For example:

```
Final Register File status (total 400 clock cycles, including terminating instruction):-

x 0  :-  0x0
x 1  :-  0xa4
x 2  :-  0x7ffffff0
    ...
    ...
x 31  :-  0xcafe
```

The user can also **inspect the memory** status after any instruction, or after the end of execution. The user can jump at an address of his/her choice. For example :-

```
Want to inspect memory (y/n)? y

Jump to address (hex) :- 0x10000000

  Address      +0  +1  +2  +3

0x10000014   04  00  00  00
0x10000010   05  00  00  00
0x1000000C   06  00  00  00
0x10000008   07  00  00  00
0x10000004   08  00  00  00
0x10000000   09  00  00  00
0x0FFFFFFC   00  00  00  00
0x0FFFFFF8   00  00  00  00
0x0FFFFFF4   00  00  00  00
0x0FFFFFF0   00  00  00  00
0x0FFFFFEC   00  00  00  00
```

After the execution of the code, an output file **logs.txt** would be generated inside the folder **output_dump**. This file would display the values of registers associated with each stage of instruction execution. For example :-

```
Clock cycle 1 :-
Fetch:
PC :     0x00000000
IR  :     0x0900006f
Decode:
RA :     0x00000000
RB :     0x00000000
Execute:
RZ :     0x00000000
PC :     0x00000090 (for next instruction)
Memory Access:
MAR :  0x00000000
MDR :  0x00000000 (before memory access)
MDR :  0x00000000 (after memory access)
Write Back:
RY :     0x00000004
```