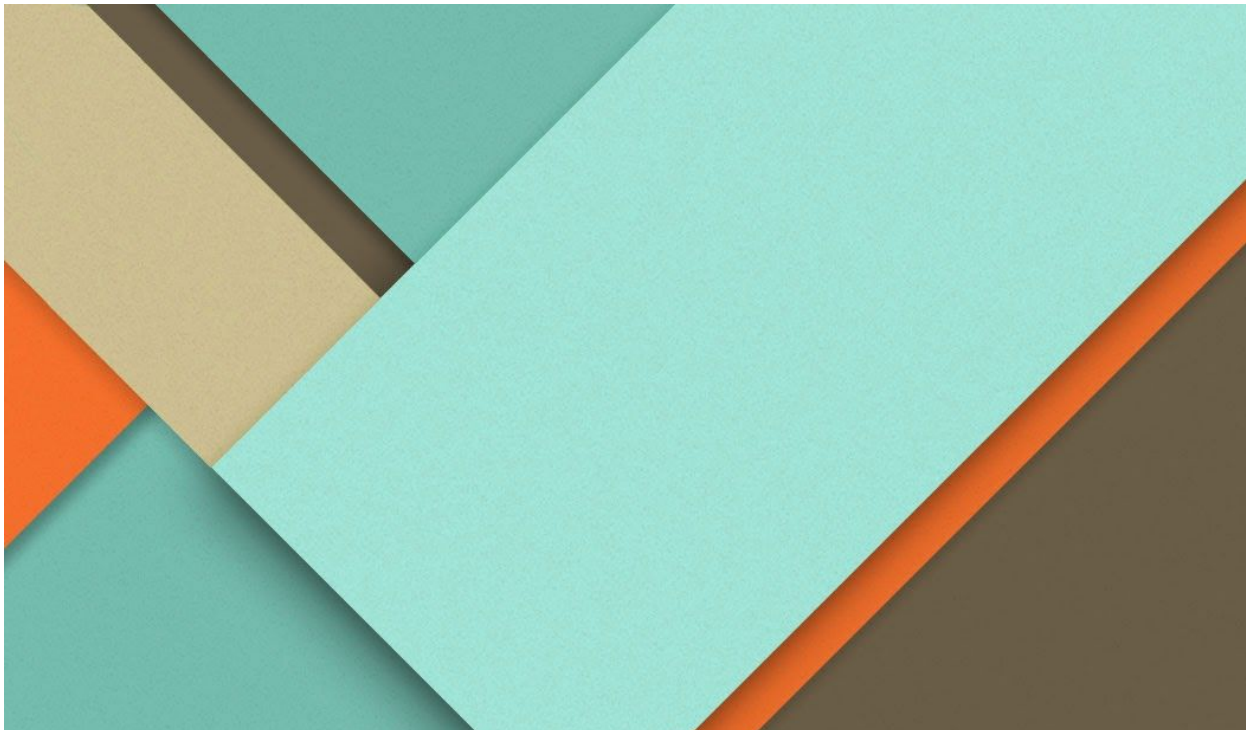


ECEC 355 - Computer Architecture

Single cycle RISC-V simulation

02.20.2020

Hanh Do Phung

Quang Anh Hoang

Electrical and Computer Engineering Department
Drexel University

Overview

This report summarizes our works on Project 1: Single-cycle RISC-V Simulation, which is part of ECEC 355: Computer Architecture course. The target of this project is to simulate the behavior of a single-cycle RISC-V processor, from parsing assembly codes, line by line, into binary instructions that the processor can understand and execute using its hardware components. A sample set of assembly instructions will be executed to validate functionality of our simulator, as the results will be shown in this report.

Background

The simulator is designed to follow the RISC-V ISA and Microarchitecture to the closest. The microarchitecture used is illustrated in the below figure.

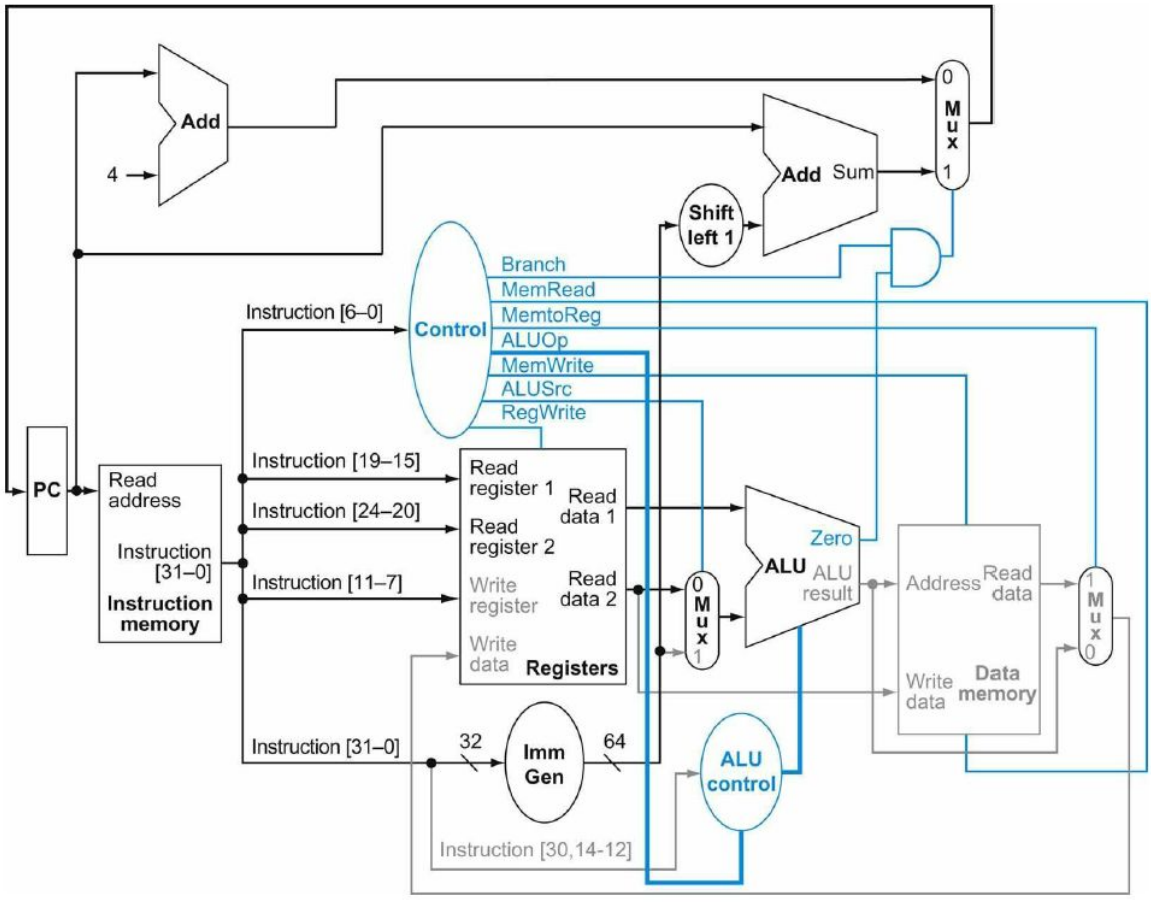


Fig. 1: Single cycle RISC-V CPU microarchitecture

As a part of the project, a parser is written to convert RISC-V assembly code text to binary abide by the RISC-V green card included in *Computer Organization and Design RISC-V Edition: The Hardware Software Interface (The Morgan Kaufmann Series in Computer Architecture and Design) 1st Edition.*

Link to a pdf version of the green card:
https://drive.google.com/file/d/1piZ6gktnmst1E0C6JmZpoGz1SJcmh8yO/view?usp=sharing

Problem analysis

The problem of designing a RISC-V can be divided into 3 tasks:

1. Designing a parser to translate RISC-V assembly to binary
2. Creating components of the architecture
3. Linking the components of the architecture and simulate test codes

The completion of task 1 and 2 can be concurrent, while task 3 can only begin after the completion of the other 2 tasks. Recognizing this, we each did 1 task and helped the other on the longer task.

To address the collaborative nature of this project where certain tasks can be done concurrently, github repository version control system was used to keep track of and exchange code files.

Solution design

I. Parser

The parser is the combination of functions: loadInstruction() and helper functions to parse RISC-V assembly code.

loadInstructions() takes in a pointer to the instruction memory and an assembly code file (trace). It reads the text line by line, parses them into binary then stores them into the instruction memory.

The parser supports instructions of types R, I, SB, S, and UJ. The binary instruction is created by “concatenating” smaller binary strings that contain specific information such as the opcode, the destination (rd), operands (rs_1 and rs_2) and the immediates (imm). The concatenation is done using logic bit operations and the return value is a 32-bit unsigned integer.

II. Components of architecture

A. Instruction memory:

This component stores binary instructions, which are parsed from assembly code, so the single-cycle processor can read and execute one instruction at a time. In every clock cycle, an instruction which its address is pointed at by the Program Counter is decoded and fetched into other components, which then execute the instruction and compute the output for it.

Another component we deemed appropriate to put in as a helper function for the instruction memory block is the Decoder. As the instruction memory needs to transfer small contiguous chunks of bits on each bus (Instruction[6-0], instruction[19-15], instruction[24-20], etc.), it needs a function to split the binary instructions into smaller parts. In the actual hardware, these buses just need to be wired with the correct signals, but for the simulator, without the wires and the instructions is stored as a 32-bit integer, Decoder is created for this partitioning.

B. Registers:

This part is a small but fast storage where the processor keeps its temporary variable and outputs of logical and arithmetic computations for the next instructions to refer to.

In our simulator, registers block is a defined structure with an array with 32 elements, and each of them is a 64-bit integer, as it resembles a RISC-V CPU with 32 64-bit registers. The structure also has 3 pointers, *pt_rd, *pt_rs1, and *pt_rs2 that points to the decoded value of three operands indicated in instruction, and these pointers are used by Registers to pass the value of operands to ALU as well as to write back the output of ALU or data memory.

C. Immediate Generator (Imm Gen):

This component executes sign extension on the immediate value of the fetched instruction, as RISC-V architecture uses 64-bit integers, so immediate values need to be extended from 32 bits (or less) to 64 bits, based on its sign (padding 0s for positive numbers and 1s for negative numbers).

In this project, we also combined the Immediate decoding feature to this block function, as it would extract immediate from the instruction and extend it to 64 bits. Since each type of instruction has a different bit width of Immediate, Imm Gen also has to identify the type of the instruction and bases on that to extend to exactly 64 bits.

D. Controller

Controller takes the opcode from the instruction as the input, and based on that it generates controlling signals to manage the flow of data going to and from other components of processor:

a. Branch: this signal is 1 if the process is set back or forward to an instruction other than the following one of the executed instruction (so-called jumping on another branch). Otherwise, it will be 0, and the next line of instruction will be executed.
b. MemRead: this signal is 1 if the process reads data from the memory, as the load function reads data from memory and store it in a register.
c. MemtoReg: this signal decides if the data written back to register is data from memory or the result of ALU computation (1 if data is from memory, 0 if it is from ALU).
d. ALUOp: based on opcode part of the instruction, the Controller generates a 2-bit signal to ALU Control to inform it with the general type of execution that ALU needs to run (R-type, I-type, SB-type, etc.)
e. MemWrite: similar to MemRead, MemWrite controls if the process requires data to be written to memory or not. MemWrite=1 allows the process to write value of a register to a memory at a specific address (store function)
f. ALUSrc: this signal controls the source of the second operand to be fed into ALU, as it is the input signal to a 2x1 mux with one input is from the 2nd source register, and the other is from the immediate value generated by Immediate Generator of the processor (1 if second operand is immediate value, and 0 if it is the value of second register).
g. RegWrite: this signal is 1 if the output of the instruction is written back to register, and 0 otherwise.

E. Arithmetic Logic Unit (ALU):

This is the component that conducts instructed arithmetic and logical operations to obtain the output of the instruction.

To communicate between the ALU and the Controller block, another block called ALU Control takes in the opcode, funct3, funct7 from instruction memory and output the operation needed to be done in the ALU as an integer.

The ALU takes in the operation it has to conduct, two integer operands then executes its operation and outputs an ALU struct. This struct consists of (1) a Zero signal for jump types instructions to signifies a jump or not and (2) the result of the operation to be written to the Registers or the Data memory or the PC register to jump to a calculated instruction memory address.

F. Data memory:

Data memory is used to store most of the data not frequently used in the operations. In a cycle where load instruction is specified, data is read from memory and will be transferred to register memory, await to be involved in future operations. When some data is done being processed, it is stored back to memory.

Data memory is implemented similar to that of instruction memory where it is an array of Cell structs, each of which has an address field and a data field to store the actual data. The data memory in this program is 256 x 64-bit wide.

G. **Program Counter (PC):**
Program Counter keeps track of where in the process the processor is in, and it indicates the next instruction to be executed. As an instruction is fetched from instruction memory, the controller reads the opcode and decides whether the process will go into a branch or not. If there is no branch in the process, the PC will be incremented by 4, and the next instruction will be loaded from instruction memory. Otherwise, when the process is redirected by a branch, PC will be re-calculated so that the process will start at the beginning of the branch.
In our simulator, the Branch signal from Controller controls the value of PC. If this signal is 1, PC will either have immediate value as the next PC value (jalr function) or increase by the value of immediate, depending on the jump/branch function; otherwise, PC will increment by 4.

III. Linking components

To most accurately simulate the single-cycle RISC-V microarchitecture, each component is treated as an object where it has some data to be stored in a struct (named after the component's name) and a few functions implementing what the logic in that component does.

The Main.c code only needs to do 2 tasks: (1) parse then load the instructions from assembly file to instruction memory and (2) run the Tick function, which simulated everything that happens within a clock cycle.

The Tick function is defined in Core.c. In this function, step by step of a processing cycle is done.

Step 1: Getting instruction from instruction memory (at the address pointed to by current PC) and decode it by splitting the binary instruction into chunks of bits and put them on appropriate buses

Step 2: Obtain control signals (Branch, MemWrite, MemRead, etc.) from the Control block by processing the opcode

Step 3: Assigning sources and destinations to registers. More explicitly, get the pointers WriteRegister (rd), Read data 1 (rs1) and Read data 2 (rs2) of the register block to point to the register location in used.

Step 4: Generate the immediate: getting the 32-bit instruction and based on the instruction type, assemble the integer from the bits representing it. It is not necessary that this is done after step 3, as it only requires the binary instruction as an input, but has to be done before step 5

Step 5: Operate the ALU Control and the ALU. Assigns either Read data 2 pointer of the register block or the immediate generated in step 4 to the second input operand of ALU, depending on the signal ALUSrc.

Step 6: Get the next value of PC. But do not set it to PC yet

Step 7: Write back the data to data or memory. The data written to register can be PC+4 (for jal or jalr instructions), or the ALU operation result, or data from Memory depending on MemToReg, RegWrite and whether there is a jump or not.

Step 8: Increment PC or write to PC the address to jump to. If there is no instructions left, tickFunc returns, signifies the end of the simulation.

All the above steps are commented in the Core.c file

Verification/Testing

To test the Parser, we wrote a printBinary function that prints the instruction decimal in binary format, so that we can check each parts of the binary (funct3, funct7, opcode, etc.) against the expectation.

Running the program with example_cpu_trace, setting x25 and x10 to 4, x22 to 1 and data memory [0, 1, 2, 3] to {16, 128, 8, 4} , **the result from x9 is 128 and x11 is 16.** Checking this against what we run through on paper, it is correct.

In addition to testing with example_cpu_trace, we wrote an assembly script to multiply a 4x4 matrix and a 4x1 vector.

The 4x4 matrix and the 4x1 vector is initialized in the data memory as below:

0 1 2 3 1

4 5 6 7 2

8 9 10 11 4

12 13 14 15 8

Each element in the matrix is stored in a 64-bit cell in memory, contiguous to each other, so matrix at index (i, j) is at Memory[i*n + j] with n as the length of each row. The result we got back is:

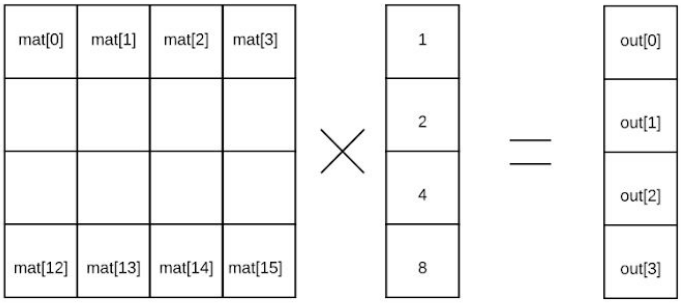
6

44

152

432

Which is NOT the right value when doing normal matrix multiplication on paper. However, the result is true to the implementation of the C code given in the extra credit part of the project description, which is:




```c
int shift (int val, int scale)
{
        return (val << scale);
}

void matrix_opr (int *mat, int *out)
{
        for (int i = 0; i < 4; i++)
        {
                out[i] = 0;

                for (int j = 0; j < 4; j++)
                {
                        out[i] += shift(mat[i * 4 + j], i);
                }
        }
}
```

From the above C functions, we derived our assembly trace, included in the .zip submission as **extra_credit_code** and use it as an input to our Main.c file to obtain the aforementioned result.

Evaluation

I. Achievements

Through this project, we learnt a lot about computer architecture and improved our developing skills as well. In this project, we got a thorough understanding of RISC-V architecture and computer architecture in general, as we simulated all of five steps in the process of a single-cycle RISC-V CPU. Implementing our understanding on a software program also taught us a lot, as we had to define and build necessary structures for the architecture to function as expected, and connecting all those structures was very challenging to achieve. However, going through all of those difficulties trained us how to connect a lot of C files and Header files and how to debug our program step by step. In

addition, we also taught ourselves to use GitHub to manage the project, do version control and to write a holistic yet detailed report like this.

II. Limitations/Future Work

There is a small flaw in designing the immediate generator for SB type instruction that we have not yet figured out how to fix. As in SS type instruction, the immediate in the binary instruction only represents bit 1:12 of the immediate text value in the trace file, which left out bit 0, causing the CPU inability to differentiate between an odd value of line number to jump to and an even value smaller than that odd value by 1. For example, bne x27, x28 -15 will be the same as bne x27, x28 -16. A quick fix we did was making our assembly code do 1 more line of nothing (assigning 0 to x0) just to round down our immediate value for SB type instructions, which wasted 1 more clock cycle.

Another limitation is, as highlighted in the Solution_design/Linking components part, the core does every step in serial, and we have only one core to execute all of the steps. Therefore, this simulation will only be limited to simulating non-parallel programs. A fix to this is to “parallelize” the simulator by, for example, use more threads to execute different tasks concurrently, pipelining the processes each component has to deal with. More explicitly, while the first thread is executing step 7 (next PC is known), another thread can begin fetching the next instruction.

Summary

Creating a simulator for RISC-V single cycle CPU has been a challenging and practical project yet leaving much room for creativity in design. The deliverables have been met with a simulator working properly over many different instructions, without skipping over the requirement to stay true to the hardware implementation.