Implementation of Simple Calculators with ISA

차 호 현(#32224560), outcider112@dankook.ac.kr  
Undergraduate Student in Mobile System Engineering, Dankook University

Code Repository: <https://github.com/HOchacha/Computer-Architecture>

**Introduction**

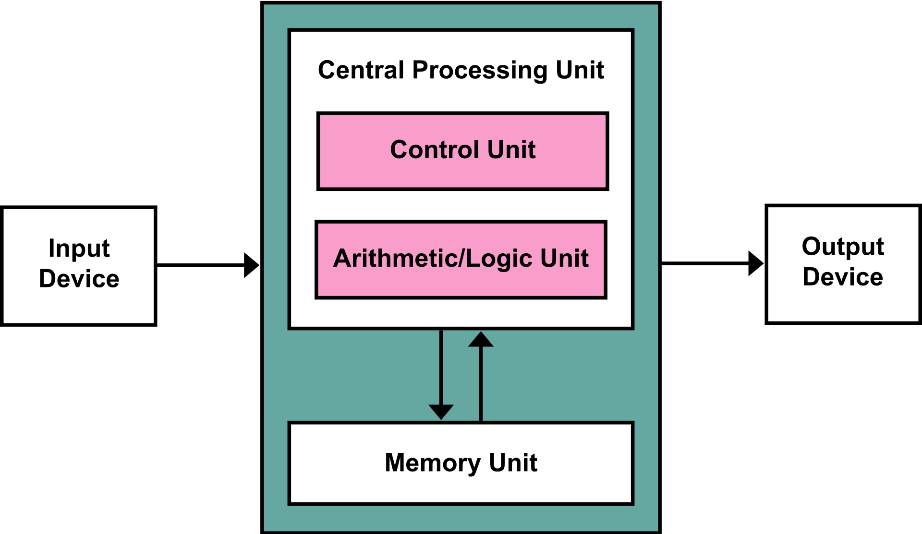
 The program, the calculator, is designed to follow instructions and execute them according to each specified operation. The calculator mimics the Von Neumann Architecture, consisting of Input & Output devices, a Central Processing Unit, and a Memory Unit.

Figure 1. Von Neumann Architecture Diagram

The calculator fetches the instruction from input devices and decodes the instruction. the decoder (CU) of the CPU gives a signal to an executor (ALU) for preparing the data and operation. At every fetch step, the special memory about the Program Counter is increased. This feature is about reading specific instructions by using the instruction address.

|  |  |  |
| --- | --- | --- |
| Operator | Instruction Format | Operation |
| +, -, \*, / | [OPCODE] [SOURCE] [TARGET]  [OPCODE] [SOURCE] [TARGET] | Arithmetic operation with source and target register. A result is stored in R0. During division operation, if the target register is set with Zero value, it triggers TRAP. |
| M | Move (Copy) the value of immediate value or register value. No result value made in this operation. |
| B | Jump to Specific instruction with Address. Address is notated in source. The target value must be specified with 0x0. |
| BEQ | If both source and R0 value are equal, jumps to specified address. The address is written in target. |
| GCD | Get the GCD value between source and target. If either source or target are not natural value, it triggers TRAP. |
| C | Set R0 as 1 if the source and target value are same. |
| H | Halt the calculator. Both operand field must be specified with 0x0. |

The instruction consists of three fields, an operator, and two operands. The operator specifies which operation is going to be executed. The operands are used to give values through immediate constants or registers. The format of the instruction is, [operator] [operand1] [operand2], fixed due to simplify the complexity of the architecture. The operator gives operation information by notating symbols such as “+”, “-“, “\*”, “/”, etc. Operand could be notated with prefixes e.g. immediate value for “0x” and address of register for “R”. Once the operation is executed, the result value must be stored in Register 0, which is one of the general-purpose registers. Each instruction finishes with a new line character “\n, and the fields are separated with white space.

**Implementation**

To implement the calculator, the program was coded and executed on Windows Visual Studio 2022 Community IDE and CLion in Windows 11 System. Also, the program can compile and run in Linux Ubuntu 22.04 Container and Virtual Machine Environment. The version of the Cmake is 3.22.1 in Linux Environment and 3.26.x in Windows.

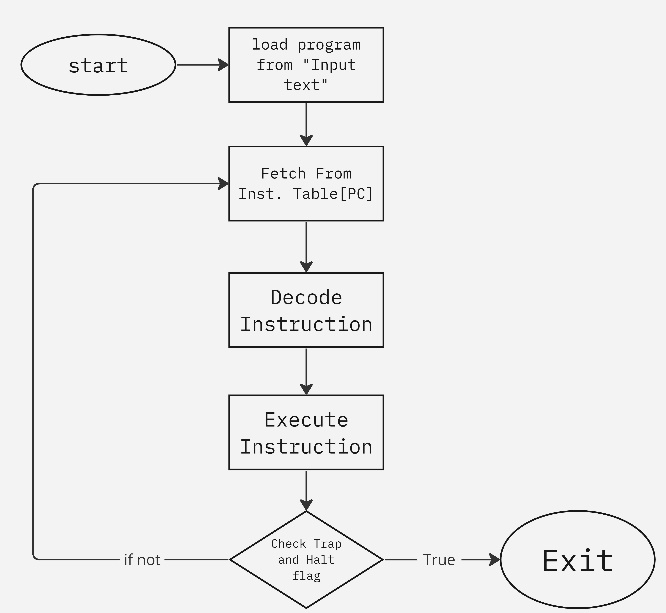


Figure 2. Calculator Flowchart

At the beginning of running the program, the calculator reads the specified text file and stores all the instructions.

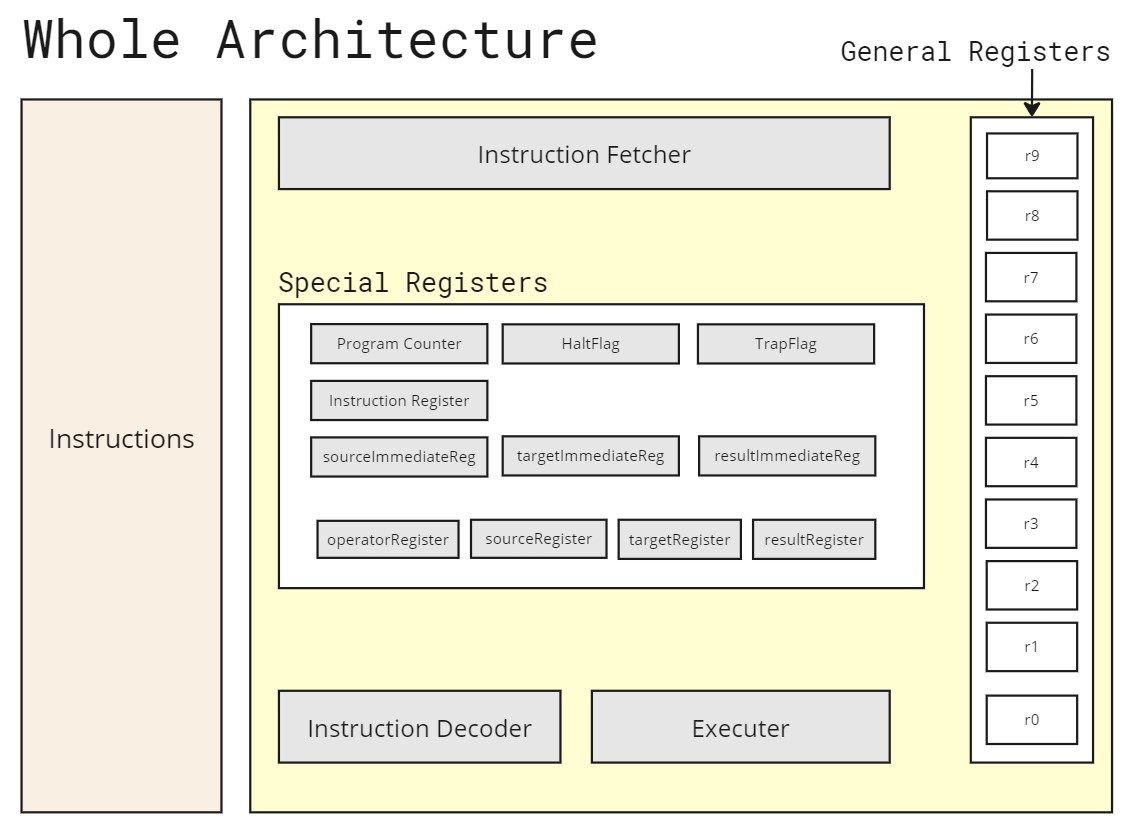


Figure 3. structure of the calculator

The calculator is implemented with an Instruction Fetcher, Instruction Decoder, Executer, and Registers. All the components have been abstracted from function to components, except for the register. There are two kinds of registers, special and general registers. All the registers are 32-bit sized variables except for some pointers. The general registers are used for storing some values. The general registers can be manipulated by instructions. The other registers, special registers have its rule. The program Counter gives the address of the next instruction. Three kinds of Immediate Register are used as buffers. The operator Register contains the operator values that specify execution.

There is some special trick about sourceRegister and targerRegister. That kind of registers are pointers for registers to address the complexity of the instruction format and simplify the whole program. In some previous implementations, there was a problem with how to manipulate the register. In most cases, we just use immediate value or that of register. We could implement this by copying the value from the register to the buffer. However, the Move operation is the problem. Move Operation works as copying the value from the target to the specified register. The former implementation wasn’t considered for implementing this kind of writing data on register specified in operand field. So, there is no way to give information and modify the register without making this step dirty. By using a pointer, we could use the actual register by making operand fields abstract. This feature makes the calculator simple.

Then, there would be some questions about using function parameters. However, there is the biggest problem with parameter locality and the static form of the function. First, a parameter is a local variable of a function. The C language hands over the value by Call by Value method, in which the changes of the parameter don’t affect to original variable in the caller function. To resolve the problem, there is only one way of using a pointer. Also, to handle the diverse cases of operand, such as Register-Immediate and Immediate-Immediate, and vice versa, the function prototype needs to be varied. However, C language doesn’t support function overloading. As a result, To give flexibility on the code work, we need to use a pointer.

**Fetch Instruction**

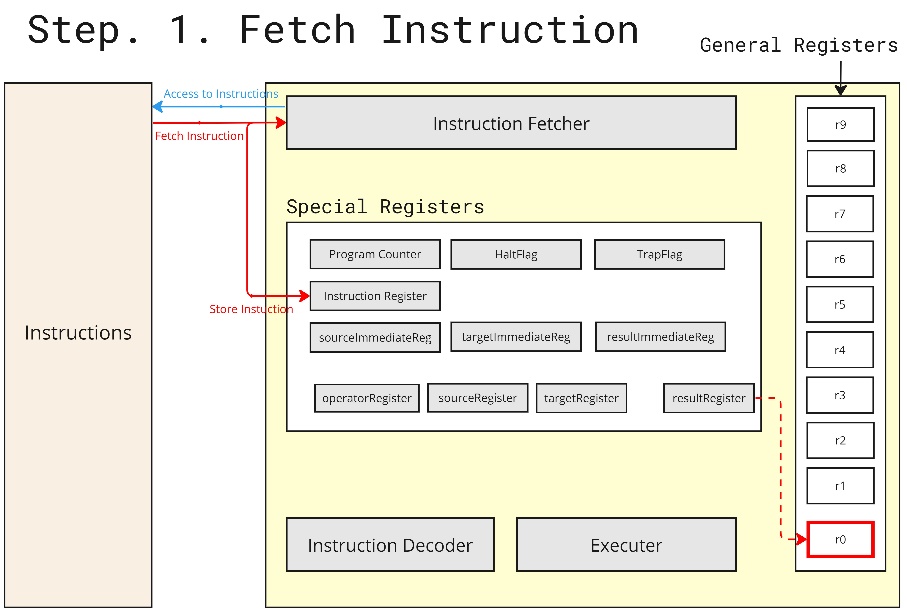


Figure 4. Fetch Instruction Step

In this step, Instruction Fetcher (function: fetchInstruction) fetches the instruction from the instruction table (pointer array). The fetched instruction is pointed by the instruction register.



Figure 5. Code Snippet about Instruction Fetcher

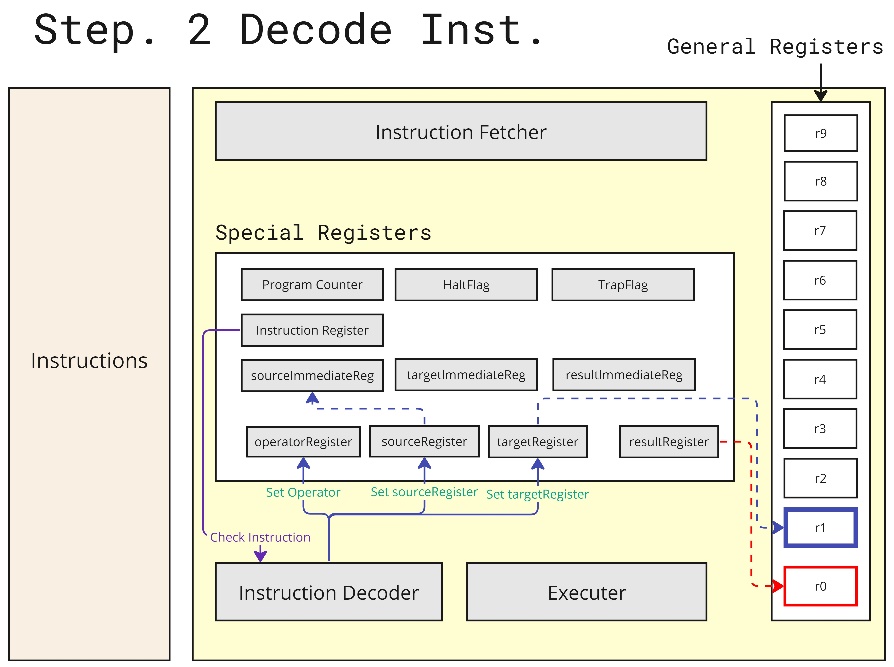


Figure 6. Decode Instruction Step

The instruction Decoder reads the instruction and sets the values to registers. Thanks to the simple and static instruction structure, the instruction parse rules could be simple, which allowed developers to focus on handling constants and registers. There is some logical problem with specifying the immediate value and register. To address the problem, the calculator uses a pointer which lets the calculator abstract with registers, even the operand contains immediate value. 

Figure 7. Code snippet about Setting Source Register Pointer

This structure lets the implementation of the instruction cycle easily separable between Execution and Decode by using the “**interface of the register**”.

**Execute Instruction**

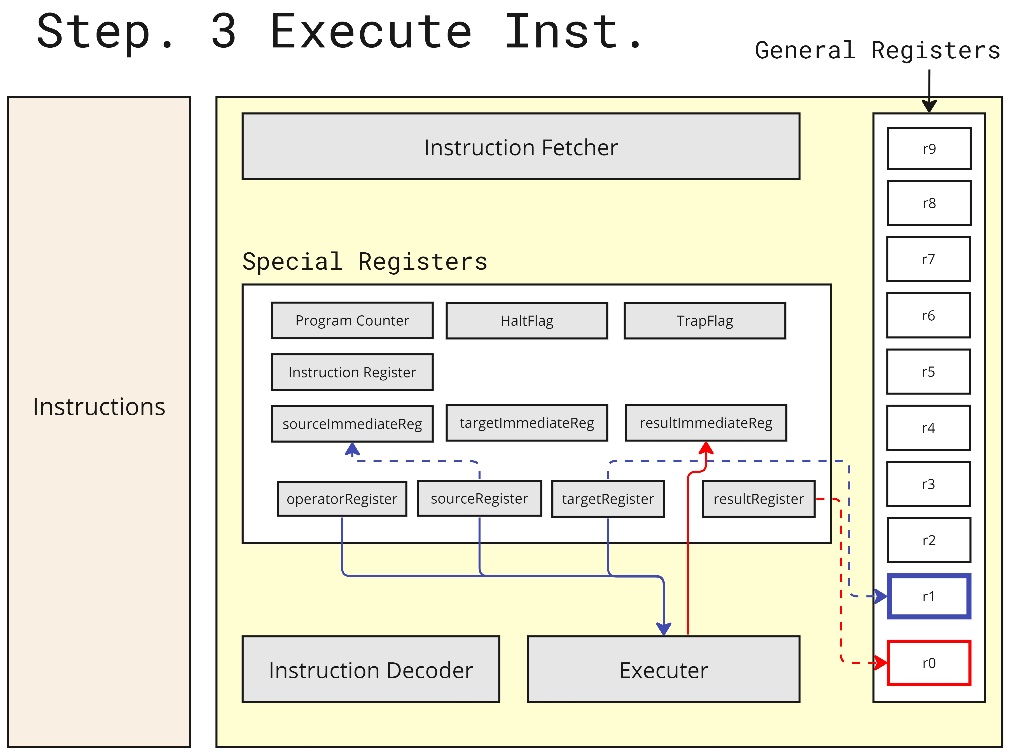


Figure 8. Execute Instruction Step

The only thing for the executer is just that calculates with two operand pointers, sourceReg, and targetReg. 

Figure 9. Code Snippets for Handling Specific Operations of Specific Operations

**Write Back Result**

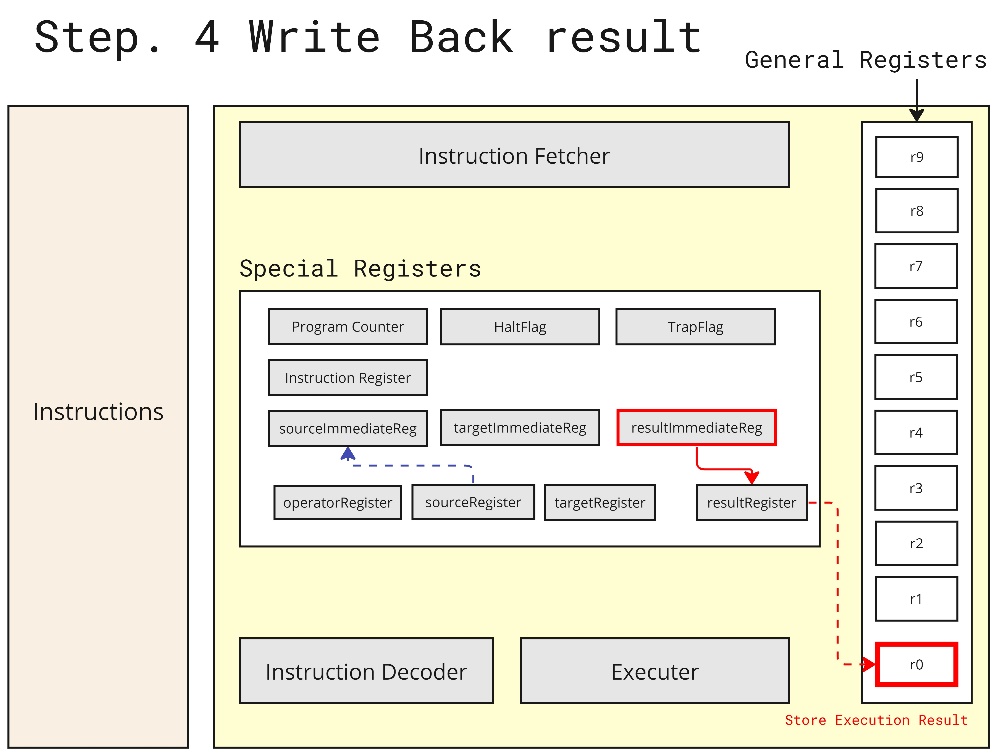
****

Figure 10. Write Back Result Step

Similarly, the result is stored in resultImmediateReg at the previous step, which works as a buffer before storing it in the actual general register. The value is stored in a specific register that is pointed by the result register.

Figure 11. Code Snippet about Write Back Result Step

**Running the Calculator Program in Linux and Window Environment**

To run the calculator, the Cmake package manager is needed. On the repository[[1]](#footnote-1), you can find the scripts to install and configure the linux environment. If you need more information about configuring this project, please refer to the README.txt file at the GitHub repository.

.

**텍스트, 스크린샷, 블랙, 폰트이(가) 표시된 사진

자동 생성된 설명**

Figure 12. Output of the Calculator Program

Once the calculator runs the instruction, you can find the log of each instruction and register about the execution. You can specify a macro “PRINT\_ALL\_REGISTER” to print the general-register values in every instruction cycle. 텍스트, 스크린샷, 블랙이(가) 표시된 사진

자동 생성된 설명

Figure 13. Output of the Calculator Program with Printing All Register Value

**Conclusion**

The primary significance of this project is emulating the patterns of CPU cycles to realize the fundamental concepts of CPU operations and implementing the program storage method of the Von Neumann Architecture in a calculator to understand how modern computers work. However, a limitation of this implementation is its operation based on string parsing rules, unlike actual computers operating at the bit level. Also, every step of processing an instruction is not implemented like MIPS. In MIPS, all components are directly connected to each other through lines for interaction, which marks a significant difference from the implementation of this project. while this project simulates the logical process of data transmission using a pointer, it’s significantly different from the complex physical structures, implemented with actual wires, marking a considerable deviation from real-world computer architectures.

**Reference**

폰 노이만 구조 , Wikipedia, Accessed on Mar. 24th , <https://ko.wikipedia.org/wiki/%ED%8F%B0_%EB%85%B8%EC%9D%B4%EB%A7%8C_%EA%B5%AC%EC%A1%B0#/media/%ED%8C%8C%EC%9D%BC:Von_Neumann_Architecture.svg>

1. https://github.com/HOchacha/Computer-Architecture [↑](#footnote-ref-1)