ECE 7063 Computer Engineering
ISA Report 1

Ceyda Kaya
Jude Urban
Parya Dolatyabi

September 2022

Abstract

Different computers throughout the history embraced different logical and physical architectures to decide how software will communicate with the processor. Every processor included a set of instructions to function and compute given tasks, and they all expected to have address memory spaces, set of registers available to programmers, a program counter register that operations are fetched through or assigned. This initial report aims to answer general architectural and design questions regarding registers, instructions, bit structures, addressing modes, and data formats following the structural checklist an ISA design should meet. Section 1 will discuss the literature background of ISA. Section 2 will include the fundamental components and properties of an ISA and address the design choices made for the project with a hardware schematic. Finally, Section 3 will be discussing the intended functionality of the architecture and a brief breakdown of the algorithm implementation.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Instruction Set Architecture

In this project it is aimed to design an Instruction Set Architecture (ISA) to make a CPU with enough functionality that any choice of program can be implemented in it. ISA is a set of commands forming the embedded programming language that a processor uses to execute a program. It acts as an interface between computational structure and physical implementation of the functions. Every processor design is based on an ISA which defines the functional capacity of the processing unit using set of instructions that processor decodes and executes.

The computer program consists of set of instructions which directs to the CPU to perform wanted operations. It can execute only low-level machine instructions in the binary format and therefore all high-level programs are first compiled in a compiler and translated into machine codes (Figure 1).

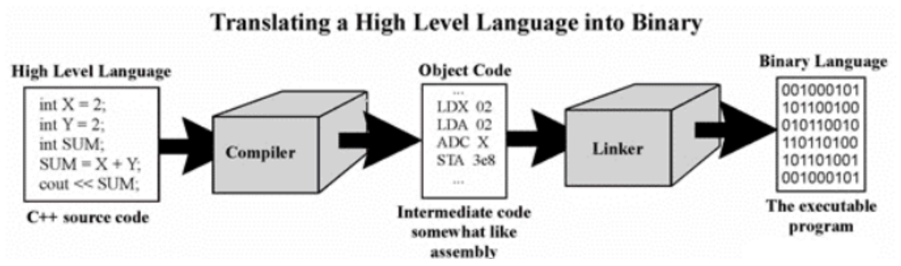


Figure 1: High-level to machine-level translation [1]

The instruction format defines the structure of the machine instruction in terms of operation code, operand, and address and this format can differ for each ISA. In general, all processors can perform the operations below and are designed according to von Neumann Architecture (Figure 2). In this architecture CPU performs basic computations by pulling data from memory, which is basically an array of bytes, and writing the results back into it. Memory unit also stores instructions for the program which controls CPU and tells what will be done with the data kept in registers.

- Basic instructions with ALU: add, subtract, multiply, divide, shift
- Comparisons: ==, ! =, >, >=, <, <=
- Managing data in different sizes and types: char, int, floating-point
- Divert the CPU to new instructions: branches, jumps, function calls

- Control, push/pop, and edit memory: load, store.

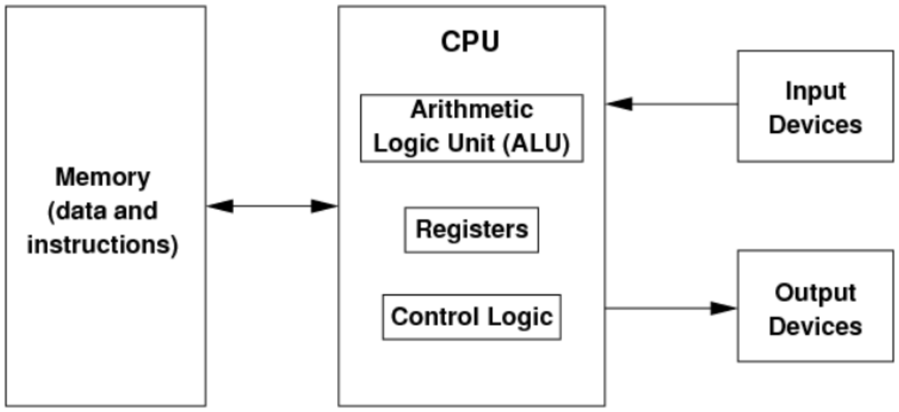


Figure 2: von Neumann Architecture [1]

Each cell has a numeric index and each index controls a different bus in the memory. In this design, 32 bit wide instructions were created and all memory read/write operations and operands were performed in groups of 1 byte. Therefore, instructions containing 32 bits were fetched from consecutive cells starting at address byte[0], byte[1], byte[2], and byte[3].

```
// load the entire instruction. all four bytes.
for(int i = 0 ; i < INSTRUCTION_SIZE ; i++)
{
    byte = machine_code_file.get();

    // end of file!
    if(byte == (uint8_t)-1)
        return;

    bytes[i] = byte;
}
```

To optimize the design and reach the goal to execute a program successfully, main components of an ISA were discussed in section 2.4.3 and design decisions were made accordingly.

1.2 Turing Completeness

In a system whether it is computational or theoretical, consistency, completeness, and decidability would be desired. If there are no contradictions in a given system, it means the system is consistent. Completeness means that a proof is required, that all true mathematical statements which exists in the system can be proven within the system. And finally, decidability is the idea that there should exist an effective approach to decide the truth or falsity of any statement. Today, an effective approach is known as algorithm which is step by step instructions to follow in order to reach the result.[2]

Turing states that any computation that can be carried out manually means that it can performed by a Turing complete digital processor. Therefore, a Turing complete machine is suggested to be able to perform any given set of tasks or computation that Universal Turing Machine [3] can, when fed with instructions and effective procedure, or algorithm as we call it.

2 Building the ISA

This section documents the final build plan and justifications into those decisions.

2.1 Foundational Language

This system will be developed in C++. It is chosen for its advantages with object-oriented programming and for its low-level memory capability. Specifically, C++ can manipulate single bytes - something a language like Python is not designed for. For these reasons, C++ is the language chosen for this system.

2.2 Readability

This ISA allows for comments using the "pound" character.

```
# code for making a for-loop!

# r1 contains the maximum iterator
# r2 contains the iterator
# r3 contains the incrementor
# r4 contains the running sum

loadi(r1, 5)        # maximum iterator limit
loadi(r2, 1)        # iterator
loadi(r3, 1)        # incrementor (++)
loadi(r4, 0)

for_loop_begin:

    # running_sum += iterator
    addi(r4, r4, r2)

    # iterator = iterator + 1
    addi(r2, r2, r3)

    # if iterator != max_iterator_limit,
    # go back to the label "for_loop_begin"
    bne(r1, r2, for_loop_begin)

# print the value out to the console!
console(r4)
      # running sum
```

2.3 Scalability

The code will be designed to support scaling for registers and instruction sizes, though desired alterations in instruction sizes will require additional work to configure.

```
#define NUMBER_OF_REGISTERS 16
#define INSTRUCTION_SIZE 4 // bytes
```

Based on the pre-compiler defines set by the user, the program will scale accordingly.

2.4 Instruction Architecture

2.4.1 Justification Into Using Four Byte Instructions

Because the ISA was created and simulated in software, there are different ways of approaching the problem - some more beneficial than others. In C++, there is no object that is less than 1 byte. Though a project on this scale may not require 4 bytes per instruction, additional computational power would be required to configure data of a smaller size. For example, all opcodes could fit within 6 bits, and, most likely, we won't have more than $2^4 = 16$ registers, so that could fit into 4 bits. However, configuring this smaller package of data would require additional CPU power to package and parse/interpret. Because of this, the ISA Architecture is as follows

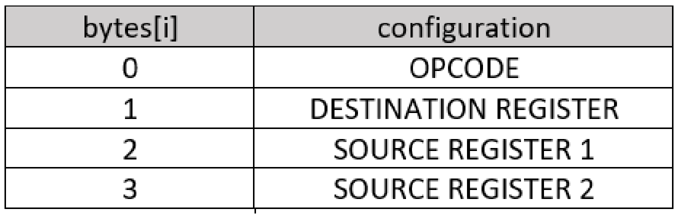
2.4.2 Size

As previously mentioned, each instruction uses four bytes to convey all information pertaining to it. Depending on the instruction type, some indices of the doubleword may or may not be used.

2.4.3 Type

R-type

R-type instructions are used to perform any arithmetic and logical operations given to the processor. A R-type instruction contains operation code, two source register operands, destination register which the result will be written into.R-Type general byte configuration would be:



bytes[i]	configuration
0	OPCODE
1	DESTINATION REGISTER
2	SOURCE REGISTER 1
3	SOURCE REGISTER 2

Table 1: R-Type, Instruction Breakdown

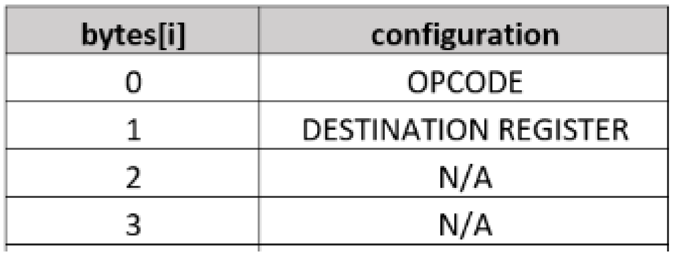
Operation Set Summary

- ADDI - add int values inside RS1 and RS2, store result in RD (OPCODE: 0x0)
- ADDF - add float values inside RS1 and RS2, store result in RD (OPCODE: 0x1)
- SUBI - subtract int values inside RS1 and RS2, store in RD (OPCODE: 0x2)
- SUBF - subtract float values inside RS1 and RS2, store in RD (OPCODE: 0x3)
- MULI - multiply int values inside RS1 and RS2, store in RD (OPCODE: 0x4)
- MULF - multiply float values inside RS1 and RS2, store in RD (OPCODE: 0x5)
- DIVI - divide int values inside RS1 and RS2, store in RD (OPCODE: 0x6)
- DIVF - divide float values inside RS1 and RS2, store in RD (OPCODE: 0x7)
- SLT - move the bits in an operand left by the specified number of bits
- NOT - returns true(1) if operand is false(0) and vice versa, store Boolean result in RD (OPCODE: 0x9)
- AND - returns true(1) if both operands are true(1), store Boolean result in RD (OPCODE: 0xa)
- NAND - return false(0) if both operands are true(1), store Boolean result in RD (OPCODE: 0xb)
- OR - return true(1) if one of the operands are true(1), store Boolean result in RD (OPCODE: 0xc)

- NOR - return false(0) if one of the operands are false(0), store Boolean result in RD (OPCODE: 0xd)
- XOR - return true(1) if and only if the values RS1 and RS2 are the same (OPCODE: 0xe)

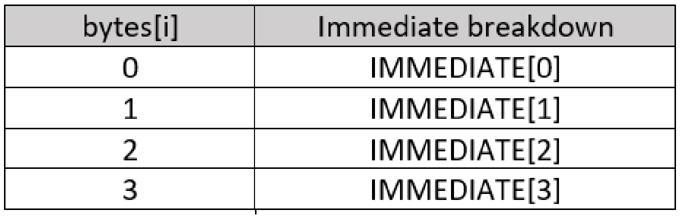
I-type

I-type instructions perform load/store instructions from/to objects and memory. Similar to R-Type, an I-type instruction contains operation code and destination register operands, but no source registers. The immediate operands for this type are stored in the machine code itself i in a 4-byte configuration, appended to the end of the instruction. To read the appended four byte immediate, a FLAG will need to be implemented that will read the following doubleword as an extension of the previous instruction, not a new instruction itself. The byte configuration for loading a float into a register would be:



bytes[i]	configuration
0	OPCODE
1	DESTINATION REGISTER
2	N/A
3	N/A

Table 2: I-Type, Instruction Breakdown



bytes[i]	Immediate breakdown
0	IMMEDIATE[0]
1	IMMEDIATE[1]
2	IMMEDIATE[2]
3	IMMEDIATE[3]

Table 3: I-Type, Immediate

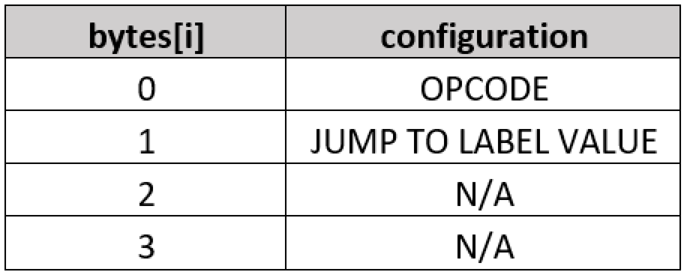
Operation Set Summary

- LOADI - Load int operand (OPCODE: 0x12)
- LOADF - Load float operand (OPCODE: 0x13)

- STORE - Store the result (OPCODE: 0x14)
- CONSOLE - Print the value stored in RS1 (Source Register 1) (OPCODE: 0x15)

J-type

J-Type instructions are used to jumps between different instructions, thus contain operation code and enumerated-label's target address to jump. By pairing this operation with logic, iterative pieces of code (loops) can be achieved. 2.2 contains an example. J-Type general byte configuration would be:



bytes[i]	configuration
0	OPCODE
1	JUMP TO LABEL VALUE
2	N/A
3	N/A

Table 4: J-Type, Instruction Breakdown

Operation Set Summary

- JUMP - jump to label 'my_label' (OPCODE: 0xf)
- BEQ - branching operation (OPCODE: 0x7)
- BNE - Jump to the label "my_label" if the data inside the two registers is identical (OPCODE: 0x11)

L-Type

"L-Types" provide a means of identification to labels. Each label that is read is incremented by one. Assuming the label already exists, its enumerated equivalent will be referenced. L-Types contain:

bytes[i]	configuration
0	OPCODE
1	LABEL VALUE
2	N/A
3	N/A

Table 5: L-Type, Instruction Breakdown

2.5 Registers

- Register_1 → 8 bits (1 byte) - Source Register
- Register_2 → 8 bits (1 byte) - Source Register
- Register_3 → 8 bits (1 byte) - Destination Register

bytes[0]	bytes[1]	bytes[2]	bytes[3]
OPCODE	**SOURCE REGISTER 1**	**SOURCE REGISTER 2**	**DESTINATION REGISTER**

Table 6: Core Registers

Processor Design Schematics

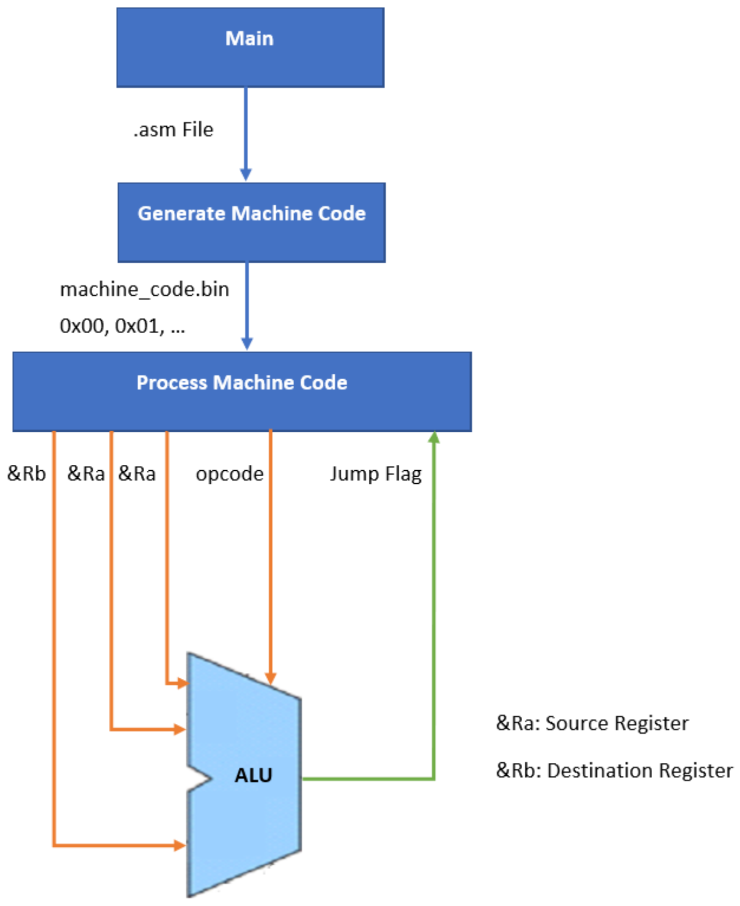


Figure 3: Effective Process Breakdown

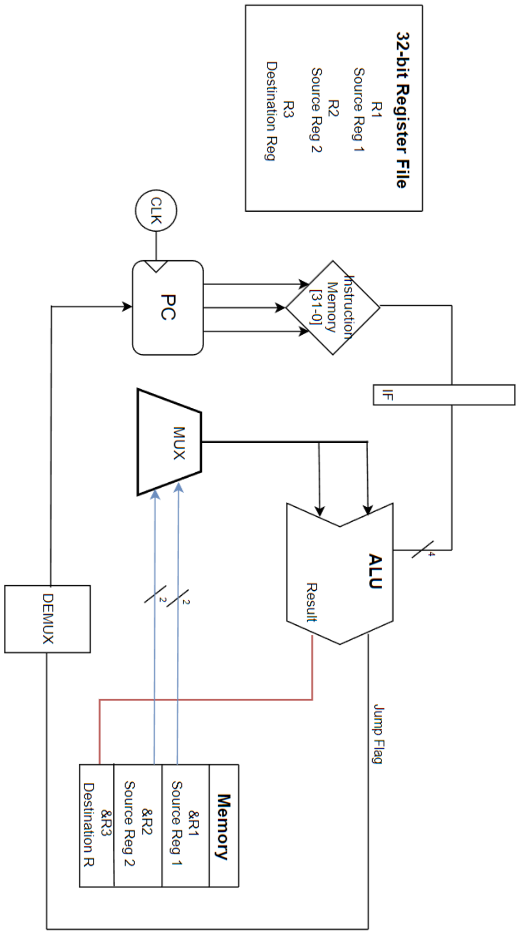


Figure 4: Operational Architecture

3 Pseudo-code for Calculating Pi

Parsing, compiling, and processing will be done though pi.asm.

```
# code for calculating pi!

# pseudo-code to calculate the ratio of pi:
# diameter = 5.0
# circumference = 15.7079632679
# pi = circumference / diameter

# r1 contains the circumference
# r2 contains the diameter
# r3 contains the result of pi

# load the float values into registers 1 and 2
loadf(r1, 15.7079632679)         # circumference
loadf(r2, 5.0)                   # diameter

divf(r3, r1, r2)                 # calculate pi
console(r3)                      # print the hex value to the console
```

If implemented correctly, this will output **"Register 0x41506c contains the value 0x40490fda"** which, converted to IEEE-754 floating point algorithm is 3.14159!

3.1 The Effective Process - Algorithm

This sub-section covers the computational structure and file hierarchy that will provide the system functionality.

1. *.asm : Instructions needed to perform a desired computation (user code).
2. main.cpp : Filename (*.asm) access from console, function calls to read and generate instructions from the specified file.
3. main.h : Size definitions, R/J/I/L-type instruction labels, Opcodes.
4. generateMachineCode.hpp : Functionality to read, parse, interpret, and write ISA components including opcode, source and destination registers, label enumerators, and immediates as machine code.
5. ALU.hpp : Logical and mathematical functions of created instruction

Example Syntax:

```
void DIVISION(RD, RS1, RS2)
{
    void DIVISION(address_RS1, RS1, sizeof(float));
    memcpy(address_RS2, RS2, sizeof(float));
    // check a divide by zero here!
    rd_f =  r1_f / r2_f;
    memcpy(address_RD, RD, sizeof(float));
}
```

6. processMachineCode.hpp : Machine code processing, 4-byte instruction breakdown and fetch loop. Functionality to read, parse, interpret, and execute ISA components written by generateMachineCode.hpp.

7. machinecode.bin : Binary machine code.

4 Conclusion

In this report, an ISA design from scratch was proposed and the planning stages as well as the structural integrity of the system was underlined and supported with visuals. Further steps for the project will include development of the software, algorithmic breakdown, and full schematic of the system.

References

[1] Scott, A., 2022. How High Level Languages are Converted to Machine Code - Wide Info. [online] Wide Info. Available at: ¡https://wideinfo.org/how-high-level-languages-are-converted-to-machine-code/¿ [Accessed 24 September 2022].

[2] Cs.odu.edu. 2022. Turing Completeness. [online] Available at: ¡https://www.cs.odu.edu/ zeil/cs390/latest/Public/turing-complete/index.html¿.

[3] Dawson, J.W., 2001. Martin Davis. The universal computer. The road from Leibniz to Turing. WW Norton Company, New York and London 2000, 257 pp. Bulletin of Symbolic Logic, 7(1), pp.65-66.