ECE 7063 Computer Engineering ISA Report 2

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.
With the high level plan and design finished for the ISA, this second report aims to answer the intricate functionality of the system.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Instruction Set Architecture

This project 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).

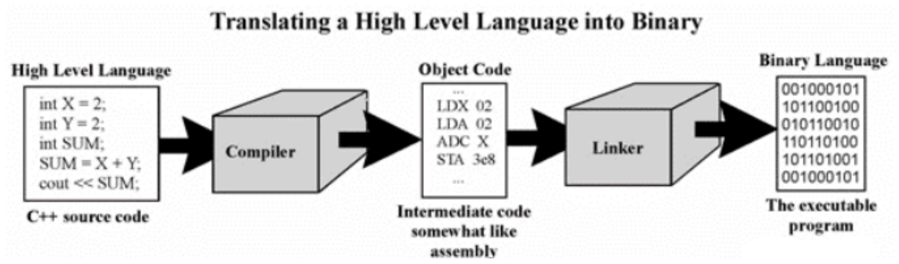


Figure 1: High-level to machine-level translation [1]

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.

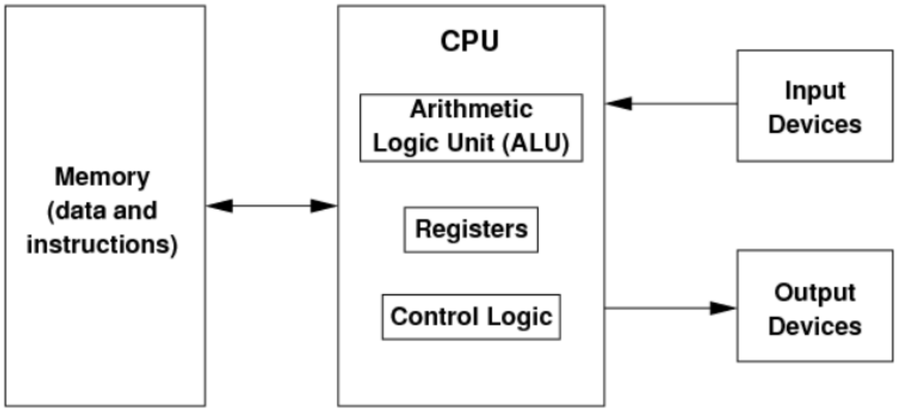


Figure 2: von Neumann Architecture [1]

In this design, 32 bit long instructions were created and all memory read/write operations and operands were performed in groups of 4 bytes (32 bits). Instructions containing 32 bits were fetched from consecutive cells starting at byte[0], byte[1], byte[2], and byte[3]. Below is pseudo-code to read the bytes from system memory.

```
// 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 5.2 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. A requirement for this ISA implementation was Turing Completeness.

2 Building the ISA

This section documents the high-level final build decisions and justifications into those decisions. All of the topics discussed were a desired requirement for the final implementation.

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 # character.

```
# ISA example code:

loadi(r1, 5)            # load the integer 5 into register 1 (r1)
loadf(r2, 3.14)         # load the float 3.14 into register 2 (r2)

# add the values inside registers 0 and 1 and store it into register 2
addi(r2, r0, r1)

console(r2)             # print the value out to the console
```

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 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

2.5 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.

3 File Hierarchy

This section covers the computational structure and file hierarchy that provides 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
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 Processor Design Schematics

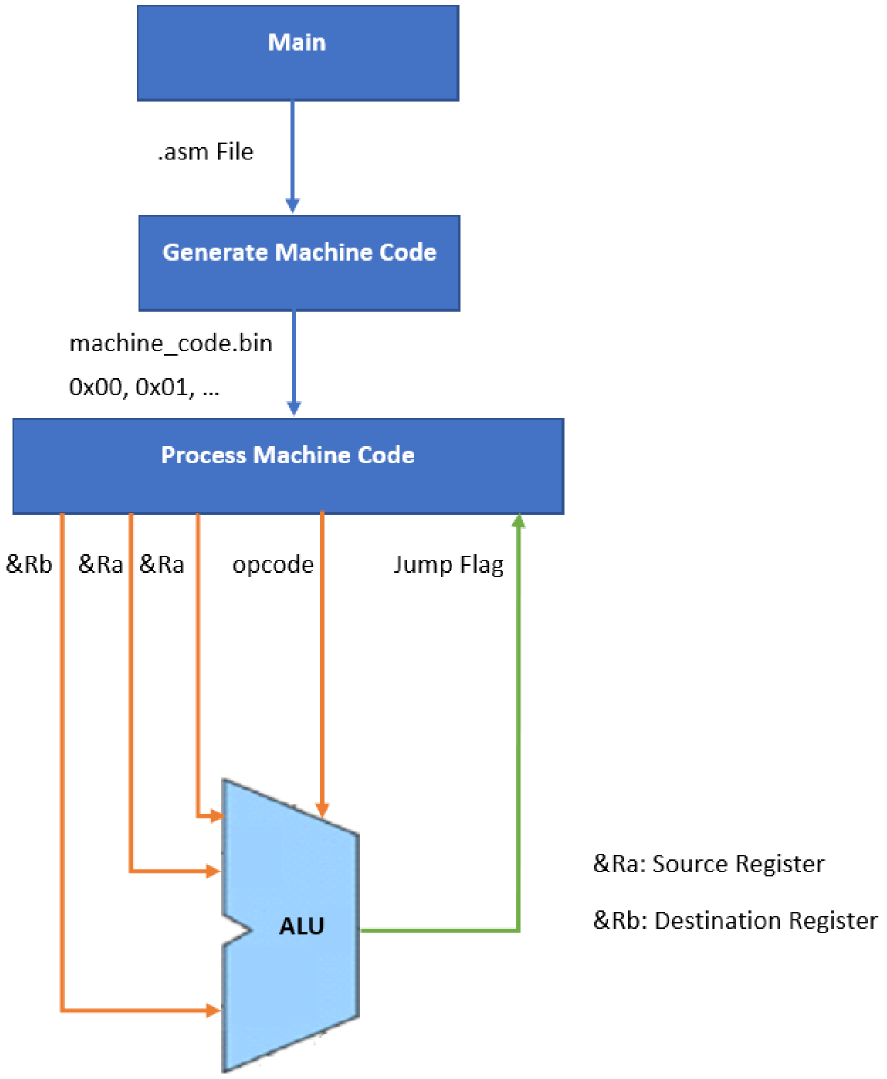


Figure 3: Effective Process Breakdown

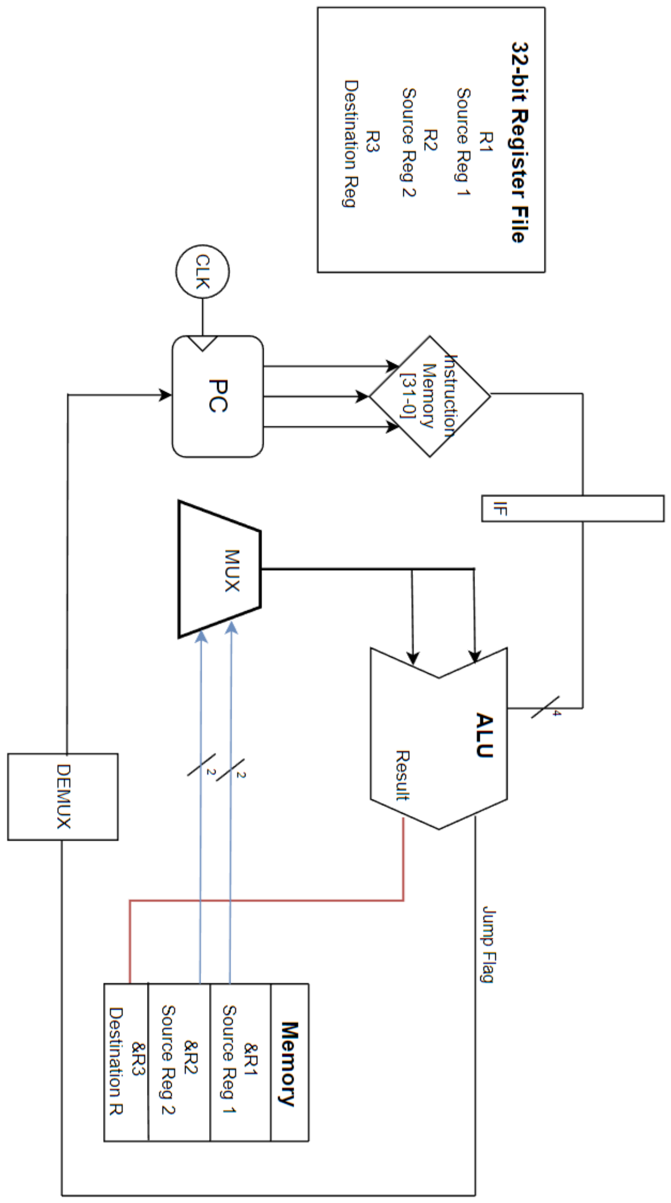


Figure 4: Operational Architecture

5 Instructions

5.1 Definitions

To assist in the development of writing redundant C++ code, a text file was created with all definitions of the instructions. It had the following format:

```
addi
addf
...
```

Two python scripts were created to use these definitions to quickly generate C++ code. Using this definition structure, both writing and processing is made easy. This code is used both the header definitions and and formulation of array of function pointers to ALU functions. This is defined in 7.2.2.

5.1.1 Header Key-Value Pairs

The first of two parts of the C++ code automatically generated by Python is the Key-Value definitions of ALU operations. Each operation is associated with a string representation and an integer representation. This is used for both parsing (writing machine code) and indexing (processing machine code). A small excerpt of these definitions is shown below.

```
#define ADDI_S "addi"
#define ADDI_V 0x0
#define ADDF_S "addf"
#define ADDF_V 0x1
...
```

5.1.2 Array of Function Pointers

The second of two parts of the C++ generated by Python is the array of function pointers. Using the text file of definitions, the following excerpt of code was automatically generated.

```
RTypeFunctionPtrs[ADDI_V] = ADDI; /* address of ADDI() */
RTypeFunctionPtrs[ADDF_V] = ADDF; /* address of ADDF() */
// and so on ...
```

Where

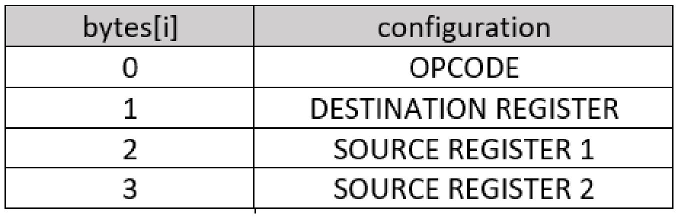
- ADDI_V - *is the integer representation of the operation, and*
- ADDI - *is the address of the function itself. (Inside ALU.hpp).*

The indexing, dereferencing, call of each function is shown
below in processRType().

5.2 Types

5.2.1 R-Type

R-type (Register 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. The data structure for R-type instructions is shown in Table 1. The opcodes and descriptions for R-Type instructions are shown in Table 2.



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

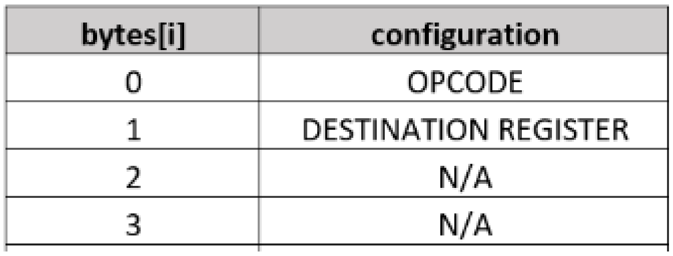
Table 1: R-Type, Instruction Breakdown

ADDI	0x0	Performs integer addition operation on two source registers and stores the value in the destination register.
ADDF	0x1	Performs floating point addition operation on two source registers and stores the value in the destination register.
SUBI	0x2	Performs integer subtraction operation on two source registers and stores the value in the destination register.
SUBF	0x3	Performs floating point subtraction operation on two source registers and stores the value in the destination register.
MULI	0x4	Performs integer multiplication operation on two source registers and stores the value in the destination register.
MULF	0x5	Performs floating point multiplication operation on two source registers and stores the value in the destination register.
DIVI	0x6	Performs integer division operation on two source registers and stores the value in the destination register.
DIVF	0x7	Performs floating point division operation on two source registers and stores the value in the destination register.
NOT	0x8	Performs NOT operation on one destination register.
AND	0x9	Performs AND operation on two source registers and stores the value in the destination register.
NAND	0xa	Performs NAND operation on two source registers and stores the value in the destination register.
OR	0xb	Performs OR operation on two source registers and stores the value in the destination register.
NOR	0xc	Performs NOR operation on two source registers and stores the value in the destination register.
XOR	0xd	Performs XOR operation on two source registers and stores the value in the destination register.
XNOR	0xe	Performs XNOR operation on two source registers and stores the value in the destination register.

Table 2: R-Type Opcodes and Descriptions

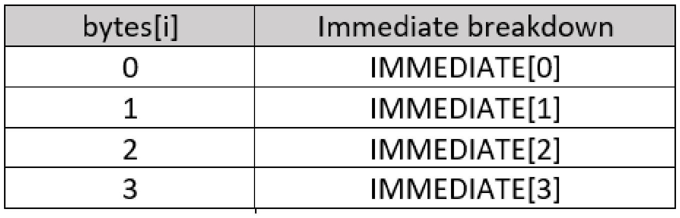
5.2.2 I-Type

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. Unlike R-Type, I-type contains no source registers. The immediate operands for this type are stored in the machine code itself i in a 4-byte configuration. It is appended to the end of the instruction. To read the appended four byte immediate, a flag was implemented that will read the following doubleword as an extension of the previous instruction, not a new instruction itself. The memory breakdown is shown in Tables 3 and 4. The opcodes and descriptions for R-Type instructions are shown in Table 5.



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

Table 3: I-Type, Instruction Breakdown



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

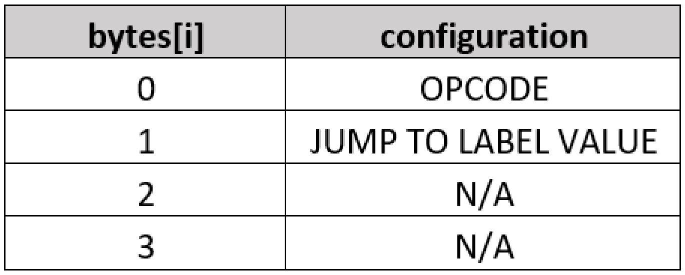
Table 4: I-Type Immediate

LOADI	0x12	Loads an integer into a destination register.
LOADF	0x13	Loads a floating point value into a destination register.
CONSOLE	0x14	Performs console operation on one destination register, printing the hexadecimal value out to the console.

Table 5: I-Type Opcodes and Descriptions

5.2.3 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. The J-Type byte configuration is shown in Figure 6. The opcodes and descriptions of each J-type is shown in Table 7.



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

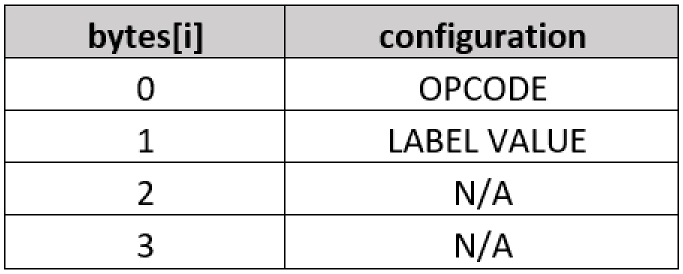
Table 6: J-Type, Instruction Breakdown

JUMP	0xf	Unconditionally jumps to a label “my_label”
BEQ	0x10	Jumps to the label “my_label” if the data inside the two source registers are not identical.
BNE	0x11	Jumps to the label “my_label” if the data inside the two source registers are identical.

Table 7: J-Type Opcodes and Descriptions

5.2.4 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. The memory configuration of L-Type instructions is shown in Table 8. The opcode and description for an L-Type instruction is shown in Table 9.



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

Table 8: L-Type, Instruction Breakdown

:	0x15	Assigns an enumerated value to a label, ending with the character ’:’

Table 9: L-Type Opcodes and Descriptions

6 Building Machine Code

One half of the system is compiling user-written commands into machine code. This section describes the process by which user-commands are compiled into machine code to be interpreted later on.

6.1 Interpreting

A vital part of compilation is the parsing of specific commands defined by the user. Parsing required a few components:

1. Ignoring comments
2. Associating opcode commands (in string format) with their corresponding integer equivalent
3. Interpreting and casting immediates accordingly.
4. Physically writing the machine code

6.1.1 Ignoring Comments

As previously mentioned, comments begin with the character #. Using this information, a simple "stall" algorithm was developed:

```
if (character == '#')
{
  // stall until the end of the line
  while((character = instr_file.get()) != '\n' && character != '\r' && character > 0)
  {}
}
```

This allows the user to make any comments he or she would like without interfering the system's processes.

6.1.2 Delimiters

To write machine code, a string (like "addi", for example) must be associated with the hex value of 0x0 (its corresponding op-code). To accomplish this, the use of delimiters was instantiated:

- `OPEN_INSTRUCTION_DELIMITER = '('`
- `CLOSE_INSTRUCTION_DELIMITER = ')'`
- `REGISTER_IDENTIFIER = 'r'`
- `REGISTER_DELIMITER = ','`

The strings, which are stored as a vector of strings, are parsed using these predefined delimiters. Using these delimiters, sub-strings associated with operations. And because every operation string is associated with an opcode, the associated opcode can be found as well.

6.2 Writing

6.2.1 Breakdown

The strings are parsed using the delimiters. This is stored into a vector of strings for easy iteration through the instructions. While iterating, each command is cross-referenced with its corresponding op-code. Depending on the type of instruction commanded by the user, the machine code will differ because some operations simply pad zeros to reach 32 bits. The machine code for an "$addi(r3, r1, r2)$" operation is shown in Figure 5. The colored columns show each component's corresponding byte machine code.

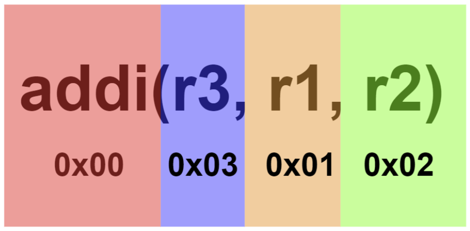


Figure 5: Instruction Breakdown

Once the instruction has been parsed and broken up into its components, writing the machine code is fairly simple. To begin, the opcode is always written first.

6.2.2 Opcode

```
// write the opcode to the file
machine_code_file.write(reinterpret_cast<const char*>(&opcode_byte), sizeof(uint8_t));
```

6.2.3 R-Type

```
// each of the register indices are found using a loop...
// cast and write the sub-string sub_str as an integer
register_byte = (uint8_t)std::stoi(sub_str.c_str());
machine_code_file.write(reinterpret_cast<const char*>(&register_byte), sizeof(uint8_t));
```

6.2.4 I-Type

```
// write the register location as machine code
register_byte = (uint8_t)std::stoi(sub_str.c_str());
machine_code_file.write(reinterpret_cast<const char*>(&register_byte), sizeof(uint8_t));

// futher on, for an integer
// write the float value as machine code
```

```
immediate_float = std::stof(sub_str.c_str());
machine_code_file.write(reinterpret_cast<const char*>(&immediate_float), sizeof(float));
```

6.2.5 J-Type

With J-Type, the label enumerator must first be recalled. After this value has been found, it can be written into machine code.

```
// find the enumerator
if((label_enumerator = findEnumFromLabel(instruction_string)) != COMPILER_SUCCESS)

// write the enumerator
machine_code_file.write(reinterpret_cast<const char*>(&label_enumerator),
sizeof(uint8_t));
```

6.2.6 Labeling

```
machine_code_file.write(reinterpret_cast<const char*>(&opcode_byte),
sizeof(uint8_t));
machine_code_file.write(reinterpret_cast<const char*>(&label_enumerator),
sizeof(uint8_t));

// pad with zeros
machine_code_file.write(reinterpret_cast<const char*>(0),
sizeof(uint8_t));
machine_code_file.write(reinterpret_cast<const char*>(0),
sizeof(uint8_t));
label_enumerator += 1;
```

7 Processing Machine Code

This section covers reading, storing, processing, and executing instructions written into machine code.

7.1 Reading

To begin, *processMachineCode.hpp* reads four bytes at a time. It checks the first byte to see if an enumerated label object should be assembled. This is performed by *readMachineCode*(). This builds an std::vector of uint32_t pointers, called "instructions".

```
// 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 COMPILER_SUCCESS;

    bytes[i] = byte;

    if(bytes[0] == L_V)
    {
        label* l = new label;
        l->enumerator = (uint8_t)bytes[1];
        l->instruction = instruction_ptr;
        machineLabels.push_back(l);
    }
}
```

7.2 Processing

To process the machine code, *processMachineCode*() reads the std::vector of "instructions". A *while* loop processes the instructions. You might be asking, "Yo dawg, what's in the program counter?" The answer is the address of the next instruction to be executed.

```
// start on the FIRST instruction, initialize the programCounter
instruction_idx = 0;
programCounter = instructions.at(instruction_idx);

while(programCounter != NULL)
{
    // update the program counter!
    programCounter = instructions.at(instruction_idx);
    instruction = *programCounter;
    instruction_idx += 1;
    ...
}
```

7.2.1 Immediate

Storing immediates into memory follows the following format:

1. Initialize an immediate flag as false. When true, the four appended bytes are read.
2. If the processor encounters a *loadi* or *loadf* instruction, raise the immediate flag to read the following four appended bytes.
3. Read the four bytes, then execute the previous instruction with the previous instruction.

```
// 1)
bool immediateFLAG = false;

...

// 3) Read an immediate instead of an instruction
if (immediateFLAG)
{
    immediateFLAG = false;
    // load immediate into memory
    ...
    // execute previous instruction
    continue;
}

// 2)
if(opcode == LOADF_V || opcode == LOADI_V)
{
    // flag true so that the next four bytes read is the FLOAT value
    immediateFLAG = true;
}
```

7.2.2 Array of Function Pointers

The array of function pointers is used when processing R-Type instructions. The other instruction types have custom implementations in $processMachineCode()$.

```
void processRType(
    uint8_t opcode,
    uint32_t* destination_register,
    uint32_t* source_register1,
    uint32_t* source_register2)
{
    // make the function call with the corresponding opcode!
    (*RTypeFunctionPtrs[opcode])(destination_register,
                                source_register1,
                                source_register2);
}
```

8 Printing

Use the *console*(rX) instruction to print the value contained inside a register.

8.1 Options

There are two printing options for this ISA. You can specify the printing style by commanding "0x" or "0b" as a header, or first line of code. A comment(s) can come before if desired.

1. Hexadecimal printing - "0x"
2. 32 bit printing- "0b"

9 Source Code

You can access all code at the "computerEngineering" GitHub repository. The source code for this particular project is in the **ISA** directory.

10 Code for Calculating Pi

This code along with several other examples can be found in the GitHub repository above.

```
# specify hex print option
0x

# 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 status values into the registers
loadf(r1, 15.7079632679)        # circumference
loadf(r2, 5.0)                  # diameter

divf(r3, r1, r2)                # calculate pi
console(r3)

# this outputs 0x40490fda which, converted to IEEE-754
# floating point algorithm is 3.14159
```

This outputs **"Register 0x41506c contains the value 0x40490fda"** which, converted to IEEE-754 floating point algorithm is 3.14159! You can convert hex to floats and vice versa using this online Hex-Float Converter.

11 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.