LBS™ LJUD & BILDSKOLAN KREATIVA GYMNASIET

Logic Simulator CPU Programmering

Neo Sellberg SU23B

2026-02-12

Handledare: Mats Tauholm

Teknikprogrammet
Gymnasiearbete 100p
HT25/VT26

Abstract

This project aims at creating a CPU from scratch using the program Logisim and creating programming languages for the CPU using python 3. Components described in the Background chapter are used to make the CPU, as detailed in the implementation chapter. Later the programming languages Digital Assembly and NeoLang were created to make programming the CPU easier. After that a program can be easily made in NeoLang, which in this case was a calculator program, as described in the Result chapter.

However in the Analysis chapter it was noted that a major limitation was the Logisim program itself since the only viable output medium is a text console component, making not all programs creatable in Logisim. It was then surmised that it would be possible to create any program creatable in a regular computer in the digital CPU if the CPU architecture designing program contains more features than Logisim.

Table of Contents

1. Introduction

This project is aimed at making a working CPU inside a simulation, and then using this CPU to create some sort of software. The CPU will only be loosely based on existing CPU architecture and for the most part the CPU will be designed for the purpose of the development of this specific software. Additionally the software will be programmed using a programming language specifically designed for this CPU, which will also be a significant part of this project.

The CPU architecture will be designed in Logisim. The code later written and then imported to Logisim for the software, along with the programming language used, will be programmed in Visual Studio Code, abbreviated as VSC, using Python 3.

2. Background

In order to understand the method by which the CPU is designed, understanding the circuits used is very important. Therefore, the first subsection of this chapter is dedicated to explaining the different circuits used in this project's CPU architecture.

While this project is mostly made without referencing the design of other CPU, we will use the "Designing a CPU" lecture from Princeton University as a loose guide, which explains the architecture of a TOY CPU, a CPU architecture designed for learning CPU architecture[2].

Circuts

RAM

RAM stands for Random Access Memory and is a type of storage circuit used by the CPU to store information. It stores many different pieces of data at a time at specific locations within the RAM called addresses. The data can be accessed or modified using this address. In this project the RAM will be able to store 65536 different 16 bit values.

Registers

Registers are used to store information for immediate use in the CPU. Unlike the RAM they usually are faster to access. This project's CPU will use 4 normal registers along with 2 special registers (see REGARRAY).

ROM

ROM stands for read only memory. In traditional computer architecture it is used to store instructions vital to the CPU because the ROM circuit can retain the information stored even when power is lost, unlike the RAM circuit. In this CPU it will be used to store the code for the program.

ALU

The ALU, or the Arithmetic Logic Unit, is a circuit that contains several mathematical and comparative functions to be used by the program.

Logisim

This project's CPU will be made and designed using Logisim, a program made for designing CPU architecture in an efficient way. The program uses a grid based pattern for all

components and wires and therefore makes understanding it easier allowing for easier refactoring if needed.

The program also includes many prebuilt circuits, allowing you to start with the actual CPU architecture faster, while not being bogged down trying to make all components yourself.

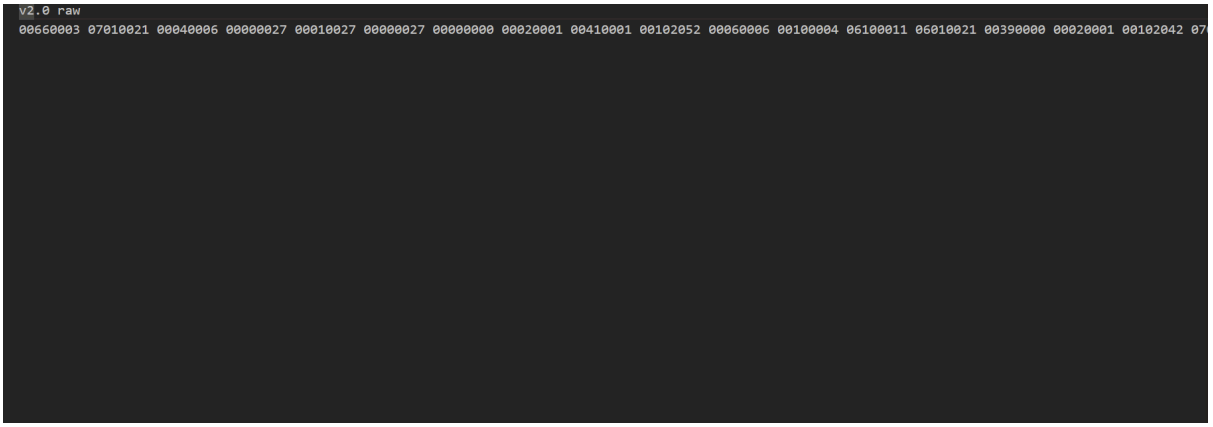
Assembly

A big inspiration for this project is the assembly language, which is the language that is directly compiled to binary. The programming language that is compiled to Logisim will have a similar format to assembly language and borrow some of its instructions such as MOV or JMP. A particularly useful feature coming from assembly is how it implements functions, or “macros” as they’re called in assembly, however they won’t be directly implemented in this project’s assembly language, but will instead be artificially created by a higher level language.

3. Potential Issues

Before starting the project it is important to establish potential issues that can arise and potential solutions to those issues if there are any.

One of the first potential issues that need to be solved is “how do we transfer the compiled code from VSC to Logisim?” The answer lies in the included ROM component. If you right click the ROM and choose the option “Save Image” you will get a file that looks something like this:



This represents the raw byte data of every line in the form of hexadecimal. We can thus compile our code in this format and simply click right click again on the ROM component and instead select “Load Image” and select our compiled file.

In order to create practical programs we will need some way to create conditional code, meaning code that only runs given a certain condition is met. In order to do this we can take inspiration from the assembly programming language.

In assembly you can create macros to repeat certain sets of instructions or create conditional code. You can use conditional jumps to jump to the point in the code where the macro is located and then use the call stack to return to the original point in the code and this will be how our program will create conditional statements. (see DA instructions JMI, CALL, and RET)

Another potential issue could be limitations in the program Logisim. For example Logisim only allows for 32 bits in any of its components, meaning we cannot physically create a 64-bit CPU and all the information from a single instruction has to fit within 32 bits. In addition to this Logisim also contains bugs which may hinder the program or the creation of a program, for example certain components do not work as they should and sometimes the execution of the program can just cease and Logisim has to be restarted

Since our CPU will be very big and complex Logisim may have a hard time running it at a certain speed, although Logisim allows for up to 4100 clock cycles per second it is not certain that this speed can be achieved.

3.1 Question Statement

How can we create a CPU architecture digitally and use it to create a program with the use of programming languages?

4. Method

First off we need a programmable CPU. In order to run a line of code we need a way to easily tell which type of instruction the current line is. In our case the part of the line that tells you what the instruction is, is stored in a nibble, meaning 16 different states and therefore 16 possible instructions.

Next we can link the different values from the instructions to registers, RAM, the ALU, or the console to effectively handle these different instructions.

These will be programmed into the language Digital Assembly, a programming language made for this project inspired by the assembly language. Since this is hard and inconvenient to program in, a higher level programming language will be created to be interpreted to DA called NeoLang.

Digital Assembly

DA is the name given to the programming language used for this project. It uses an compiler built in python 3.13.7. It is based on the assembly language, a low level language used by most modern, mainstream programming languages. It includes an instruction followed by a varying number of parameters. It is fully acceptable to write instructions, as well as parameters, in either uppercase or lowercase or a combination of the two.

All information, including parameters, of a single instruction must be containable inside a 32bit value and in the following instruction set the format for any instruction will be shown using this representation in little-endian(least significant nibble first) byte order:

IXXX VVVV
I = Instruction
V = Value to write

This is the data format for the instruction IMM. Each letter represents one nibble, 4 bits, of data. The letters written under the format show what each letter means, for example the nibble indicating what instruction it is is represented using the first nibble, I, and the last 4 nibbles, written with Vs, represent what 16 bit value to use for the instruction, meanwhile each X shows that that specific nibble is not used in this instruction.

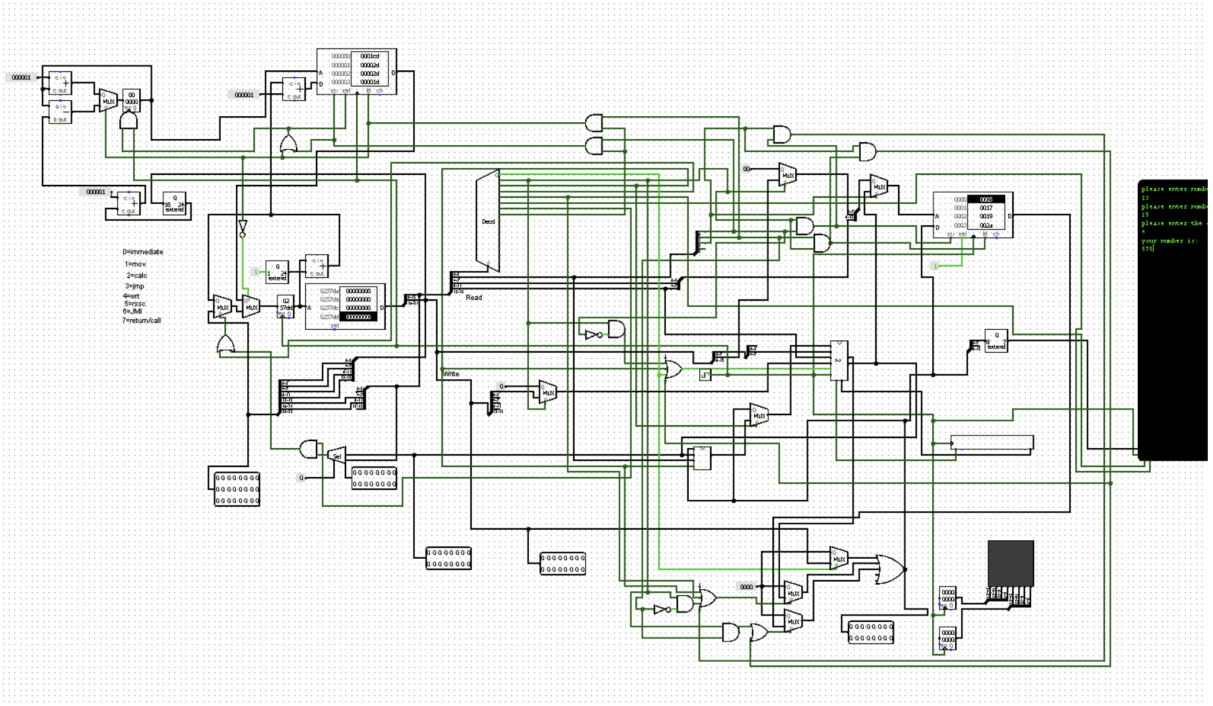
In a lot of the instructions a data storage address is needed. This can most often be designated as a specific register and in that case something like “REG0” should be written with the number after “REG” signifying the index of the register. For some instructions “RM” can be written instead of “REG” to instead use RAM, with the number after being used as the address.

NeoLang

NeoLang is a higher level language which is interpreted into DA, making it less tedious to do most things and makes loops and conditional statements possible. NeoLang mostly takes inspiration from Python in the form of its structure. It is higher level than DA but still a lot lower than something like the C language. NeoLang implements additional features such as functions, variables and if statements.

5. Implementation

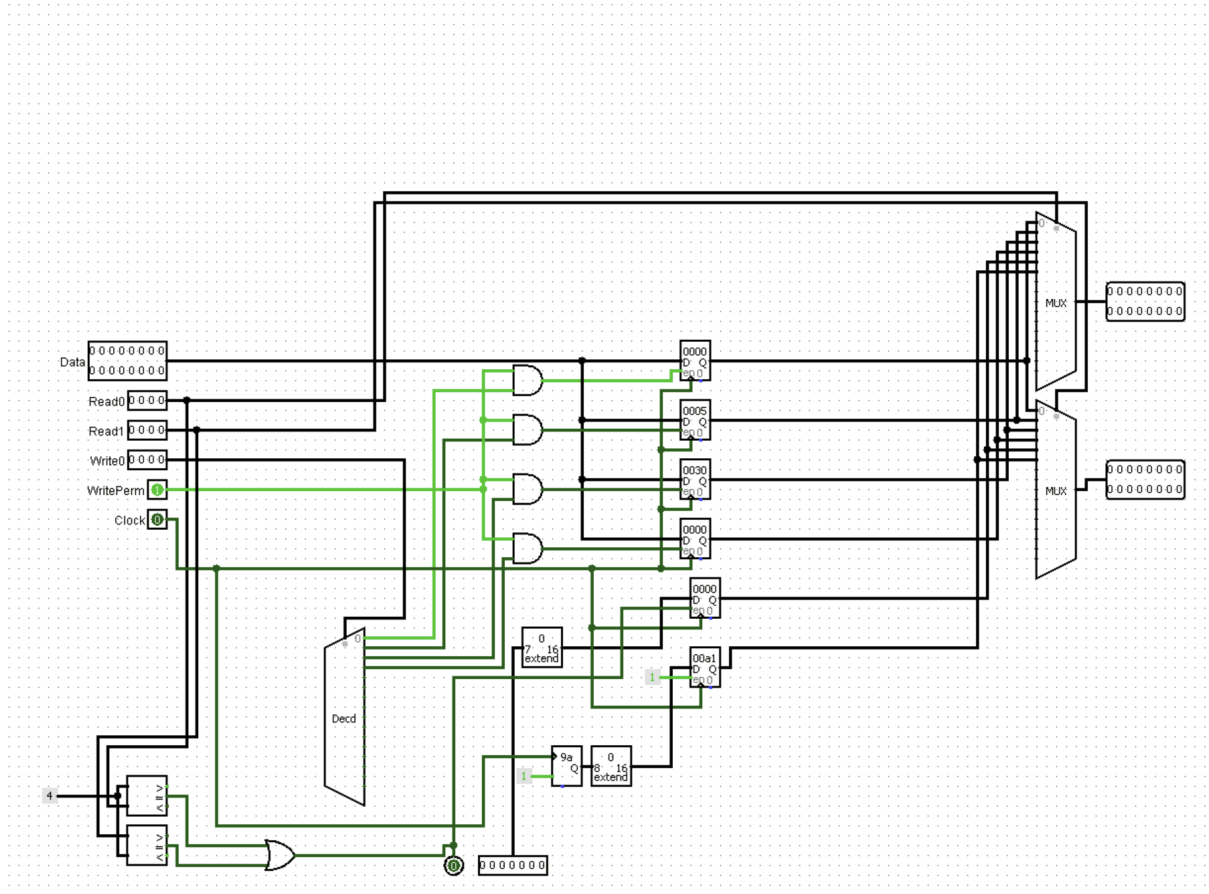
CPU Architecture



An image of the entire CPU

REGARRAY

The first component we need to make is the REGARRAY which contains all registers and provides storage which can be easily used by other parts of the CPU.



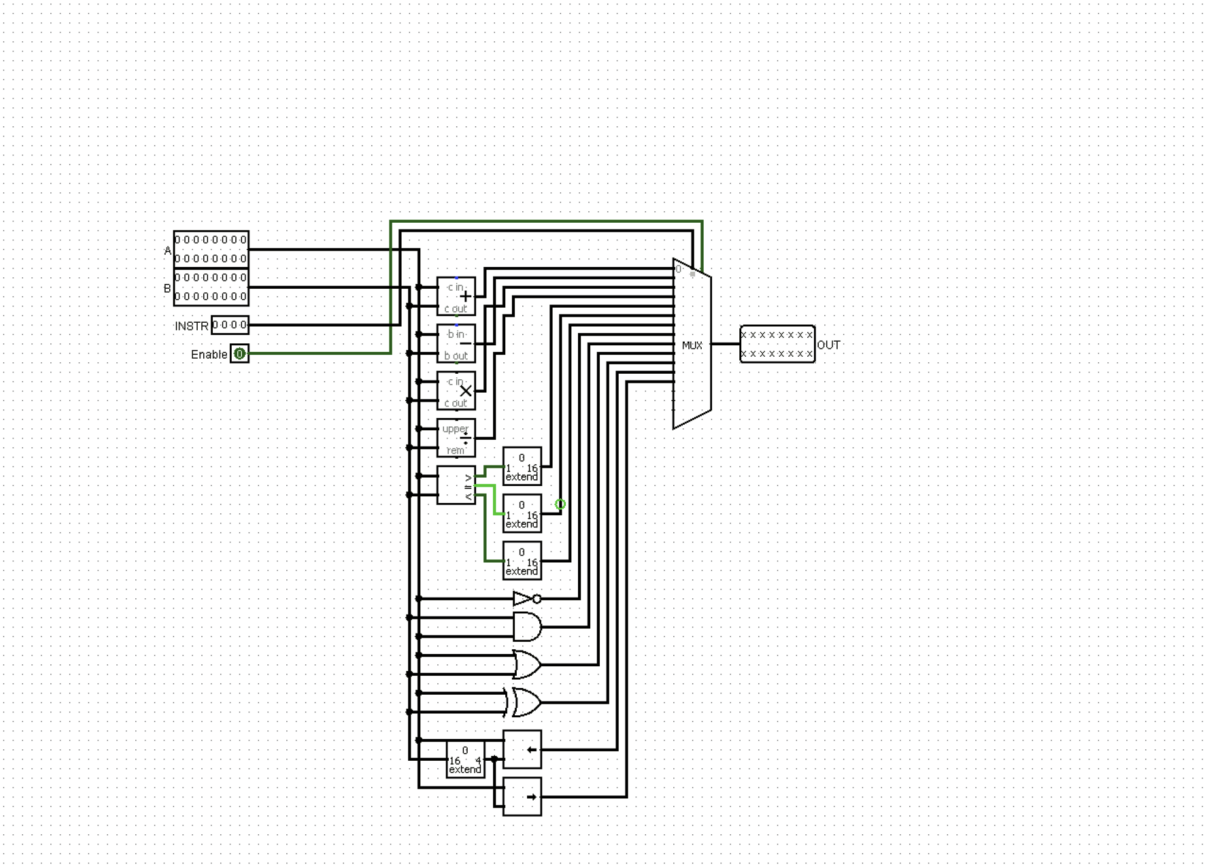
The REGARRAY contains two outputs, the main output, and the secondary output. A secondary output is needed for calculation instructions that require two values. The output registers are determined by the two read nibbles in the input, namely read0 and read1, which correspond to the main output and secondary output respectively.

The REGARRAY has the three additional inputs, Data, write0, and WritePerm. Data is the data that will be written if writing to a register. Write0 determines which register will be written to and WritePerm determines if the REGARRAY has permission to overwrite the value of a register. If WritePerm did not exist then it would overwrite the value of REG0 almost every clock cycle.

The REGARRAY contains 4 normal registers which can be written to and read from freely, however the last two registers are special. The first special register is the keyboard register. It stores the last pressed key and gets the next key pressed after that whenever it is accessed and the next special register is the random number generator which generates a new random number every clock cycle using Logisim's built in random number generator.

ALU

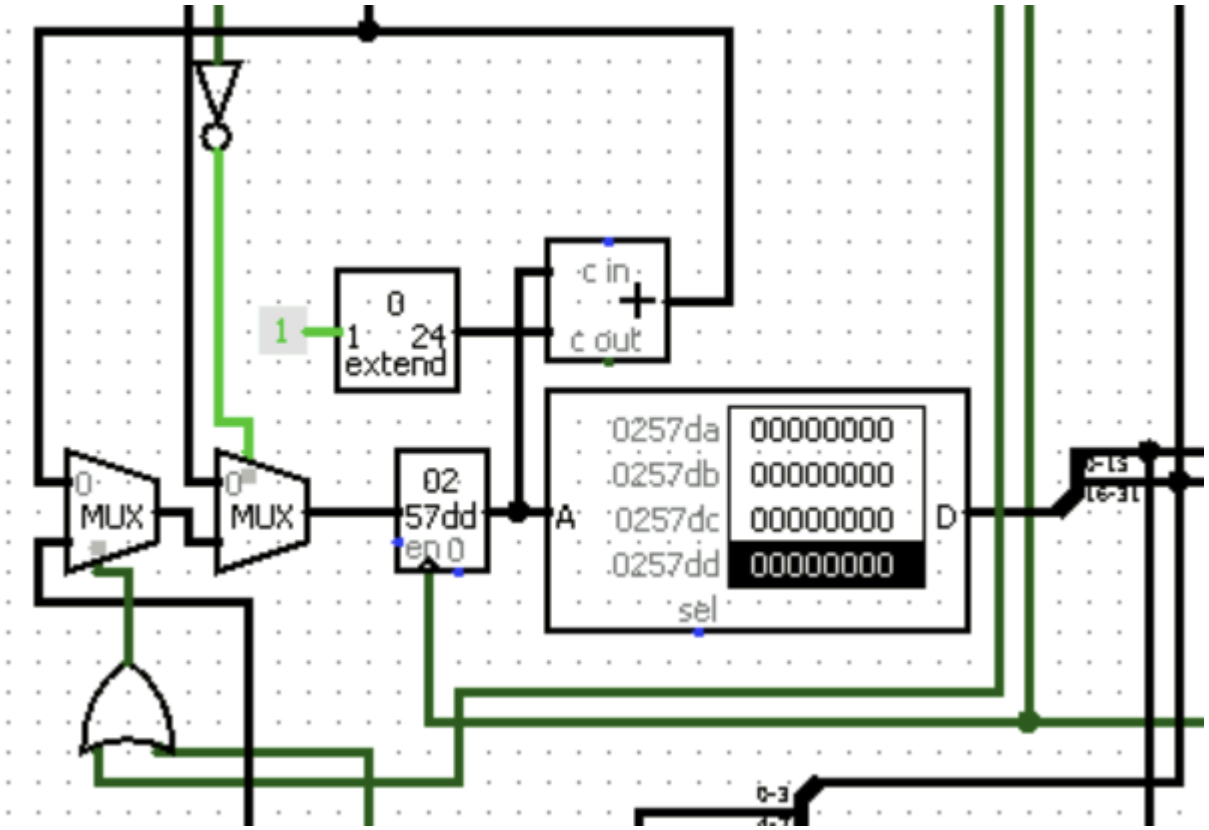
The next component to make is the ALU (Arithmetic Logic Unit) which enables the CPU to make calculations and use arithmetic, logic, and comparisons.



Here we make all calculations simultaneously and simply select the one we want using the MUX (Multiplexer) which chooses one of the values based on the INSTR input.

The INSTR input therefore determines the operation used and the A and B inputs are the values used in the operation, with A always coming first (like A-B or A/B). If the operation only requires one value, like the NOT operation, then it uses only A.

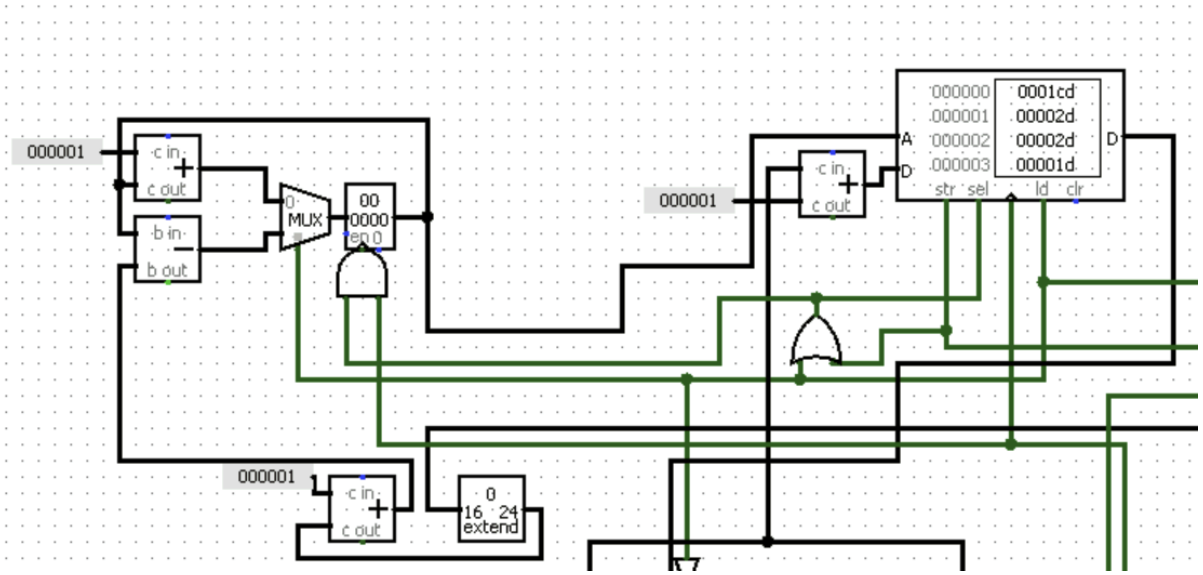
ROM and Code



This part here is not a component in itself but instead a part of the larger CPU component (see the image at the top of the Implementation tab). The big storage array here is the ROM, which stores all the compiled code and instructions to be used in the program. The components preceding it are used to figure out which line to read. Normally it just jumps to the next line, but in special cases we need to give it another line.

The first of these scenarios is if we are using a jump instruction (an instruction to jump to a specific line). Then we should get a value from the instruction and use that to update the ROM. Another scenario is if it is trying to get a line from the call stack, in which case we get the line from the call stack. The first MUX handles whether or not a jump instruction is used and the second MUX whether or not a line is coming from the call stack.

Call Stack



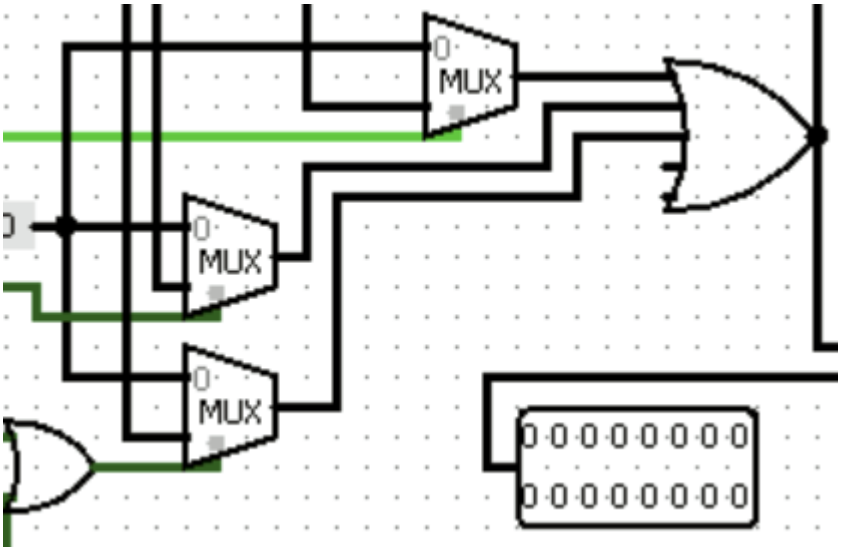
Here we have the call stack part of the CPU. This uses a 24 bit RAM component for the call stack meaning the call stack has a size of 2^24 or approximately 16.8 million, which is more than enough.

We want to do one of two things with the call stack:

1. Add a line to the call stack.
2. Get a line x addresses away from the top of the call stack (x > 0)

To do this we keep track of where the top of the call stack is using a register. We can then add one to it if adding a line or subtract by x if we want to get a line. x, in this case, comes from the instruction data.

Main Bus



Here we have the start of the main bus. A bus is a connection carrying certain data across the entire component. We use different multiplexers to determine which one to send onto the main bus. Only one of the multiplexers will ever be active meaning that when they later move into the big OR gate only the value that we let through will appear in the main bus. This data can then be used by the REGARRAY, ALU, console, or RAM.

DA Instruction Set

In order to handle the different instructions we use a dictionary to determine what function to use to parse them. Every instruction has an associated instruction class which contains their keyword, the amount of parameters they have, and the function to parse them and turn them into binary code.

Python

```
INSTRSET = {"IMM":Instruction("IMM", 1, InterpretIMM),
            "MOV":Instruction("MOV", 2, InterpretMOV),
            "CAL":Instruction("CAL", 3, InterpretCAL),
            "RSSC":Instruction("RSSC", 0, InterpretRSSC),
            "JMP":Instruction("JMP", 1, InterpretJMP),
            "WRT":Instruction("WRT", 1, InterpretWRT),
            "JMI":Instruction("JMI", 2, InterpretJMI),
            "MOA":Instruction("MOA", 3, InterpretMOA),
            "CALL":Instruction("CALL", 0, InterpretCALL),
            "RET":Instruction("RET", 1, InterpretRET)
            }
```

IMM

IMM stands for immediate. Its function is to input values in the CPUs registers. It only inputs to the first register, namely REG0. The value must be an integer, but can also be written using RGB values using the prefix RGB then the format RRR/GGG/BBB.

Example instruction(s):

```
None
IMM 255
IMM RGB255/255/255
```

Data format:
IXXX VVVV
I = Instruction
V = Value to write

Code:

```python
def InterpretIMM(args) -> list: #Input the number into REG0
    val = 0
    if (args[0].upper().startswith("RGB")): #example
RGB255/255/255
        rgb = args[0].upper().removeprefix("RGB").split("/")
        val = (round(15*int(rgb[0])/255.0)<<8) +
(round(15*int(rgb[1])/255.0)<<4) + round(15*int(rgb[2])/255.0)
    elif (args[0].upper().startswith("HEX")):
        s = args[0].upper().removeprefix("HEX")
        rgb = [s[i:i+2] for i in range(0, len(s), 2)]
        val = (round(15*int(rgb[0], 16)/255.0)<<8) +
(round(15*int(rgb[1], 16)/255.0)<<4) + round(15*int(rgb[2],
16)/255.0)
    else:
        val = int(args[0])
    return [0, val]
```

MOV

MOV stands for move. It is used to move a value from one memory location to another. It can use both RAM and registers as parameters, however both of the locations cannot be RAM

addresses. The first parameter will be what location to write the value to and the second parameter will be what location to copy the value from.

Example instruction(s):

```
None
MOV REG1 REG0 #stores the value of REG0 to REG1
MOV RM0 REG0
```

Data format:
ISA(A/R2) WRAA
I = Instructiona
S = Settings (Including whether to write to or read from RAM and whether to read address from a register. See MOA)
A = RAM adress (first A is the highest bit)
W = Which register to write to (if writing to a register)
R = Which register to read from (if reading from a register)
R2 = Register to read address from if reading from a stored address as specified in the settings

Code:

```python
def InterpretMOV(args): #Move from one memory location to another
    val = [1, 0]
    write = InterpretReadWriteMain(args[0], False)
    val[0] += write[0]
    val[1] += write[1]

    read = InterpretReadWriteMain(args[1], True)
    val[0] += read[0]
    val[1] += read[1]
    if (args[2] == 23):
        print(args[1])
    return val
```

```python
def InterpretReadWriteMain(arg: str, r: bool): #interpret where to read and write from on the main bus
```

```
    val = [0, 0]

    if r:
        if (arg.upper().startswith("RM")): #read from RAM
            val[0] += 32 +
((int(arg.upper().removeprefix("RM"))&0xff00))
            val[1] +=
(int(arg.upper().removeprefix("RM"))&0xff)<<8
        else:
            val[1] += REGISTERS[arg.upper()] << 4
    else:
        if (arg.upper().startswith("RM")): #write to RAM
            val[0] += 16 +
((int(arg.upper().removeprefix("RM"))&0xff00))
            val[1] +=
(int(arg.upper().removeprefix("RM"))&0xff)<<8
        else:

            val[1] += REGISTERS[arg.upper()]
    return val
```

CAL

CAL stands for calculate. Its function is to do a calculation or comparison between two values and output the result into REG0. The two memory location parameters must be registers and may not be REG0, and the last parameter will signify the operation. Currently the only operations are ADD (addition), SUB (subtraction), MUL (multiplication), DIV (division), GT (greater than), EQ (equals), and LT (less than), NOT (bitwise not of the first value), AND (bitwise and), OR (bitwise or), XOR (bitwise xor), LSFT (bitwise left shift), RSFT(right shift).

Example instruction(s):

```
None
CAL REG1 REG2 ADD # calculate REG1 + REG2 and store into REG0
```

Data format:
IOXB XAXX
I = Instruction
O = Operation
A = Value 1
B = Value 2

Code:

```
Python
def InterpretCAL(args): #Calculate and move into REG0. First arg is A second is B
    assert(args[1].upper() != "REG0" and args[0].upper() != "REG0")
    return [2 + (REGISTERS[args[1].upper()]<<12) + (CALCINSTR[args[2].upper()]<<4), (REGISTERS[args[0].upper()]<<4)]
```

RSSC

Will reset the console screen.

Example instruction(s):

```
None
RSSC
```

Data format:
IXXX XXXX

Code:

```python
def InterpretRSSC(args): #Reset Screen
    return [5, 0]
```

WRT

WRT stands for write. It will write the ASCII value of the value in a certain register to the console.

Example instruction(s):

```
WRT REG0
```

Data format:
IXXX XRXX
I = Instruction
R = Register to read from
S = if first bit is 1 reset the screen instead of writing

Code:

```python
def InterpretWRT(args): #Draw to Screen ex: WRT REG0
    val = [(4), (REGISTERS[args[0]]<<4)]
    return val
```

JMP

JMP stands for jump. Its function is to jump from one line to another in the code in order to be able to create loops.

Example instruction(s):

```
None

JMP 5
```

Data format:

IL(L/R) LLLL

I = Instruction

L = Line to jump to

S = Settings

R = Address to read line from if that is enabled

Code:

```python
def InterpretJMP(args): #Jump to line
    args[0] = FindJumpLocation(args[0], args[1])
    return [3 + ((args[0]&16711680)>>4),
((args[0]&3840)<<4)+(args[0]&255)]
```

```python
def FindJumpLocation(arg: str, linenum):

    if arg.upper() in Macros:
        return Macros[arg.upper()]
    elif arg.startswith("+"):
        return linenum + int(arg.removeprefix("+"))
    elif arg.startswith("-"):
        return linenum - int(arg.removeprefix("-"))
    else:
        return int(arg)
```

JMI

JMI stands for jump if. It works like JMP but with an additional argument for a register. If this register's first bit is 1 it will proceed with the jump, otherwise it will continue execution.

Example instruction(s):

```
None
JMI 5 REG0
```

Data format:

ILLR LLLL

I = Instruction

R = Register to read from

L = Line to jump to

Code:

```python
def InterpretJMI(args):
    args[0] = FindJumpLocation(args[0], args[2])
    return [6 + ((args[0]&0xff0000)>>12), 0xffff&args[0]]
```

MOA

This instruction will move the value from the address stored in a specific register to a specified register. The first parameter will specify whether you're writing to the address or if you’re reading from it, 0 if the former, 1 if the latter. The second parameter is the write location and the third is the read location. Uses the same instruction nibble as MOV and therefore the same data format.

Example instruction(s):

```
None
MOA 0 REG1 REG0 #reading from the address in REG0 and writing to REG1
MOA 1 REG0 REG1 #taking the value from REG1 and writing it to the address on
REG0
```

Data format:
ISA(A/R2) WRAA
I = Instruction
S = Settings (Including whether to write to or read from RAM and whether to read address from a register)
A = RAM adress (first A is the highest bit)
W = Which register to write to (if writing to a register)
R = Which register to read from (if reading from a register)
R2 = Register to read adress from if reading from a stored address as specified in the settings
Code:

```
Python
def InterpretMOA(args):
    val = [65, 0]
    if (args[0] == 0): # read from the adress on the second
register and write it to the first
        #adress is on the second nibble of the second segment
while write adress is the first of the second segment
        val[0] += 32
        val[1] += REGISTERS[args[2]]<<4
        val[1] += REGISTERS[args[1]]
    else: # read from the second register and write it to the
adress of the first
        #adress is on the second nibble of the second segment
while the read value is on the last of the first segment
        #MOA REG0(write to the adress of this) REG1(read the
value from this)
        val[0] += 16
        val[0] += REGISTERS[args[2]]<<12
        val[1] += REGISTERS[args[1]]<<4
```

```
    return val
```

CALL

This instruction will store the current instruction index + 2 into the call stack.

Example instruction(s):

```
None
CALL
```

Data format:
ISXX LLLL
I = Instruction
S = Settings (whether to write to the call stack, when the instruction is CALL, or read from it, when the instruction is RET)
L = how far down the call stack to go if calling RET

Code:

```python
def InterpretCALL(args):
    return [23, 0]
```

RET

This instruction will read the value x amount of addresses from the top of the call stack and make that the new current line.

Example instruction(s):

```
None
RET 2 #loads the value 2 from the top of the call stack and returns to that
line
```

Data format:
ISXX LLLL
I = Instruction
S = Settings (whether to write to the call stack, when the instruction is CALL, or read from it, when the instruction is RET

Code:

```python
def InterpretRET(args):
    return [39, int(args[0])]
```

NeoLang

Since DA is hard to program in, an additional programming language will be created to serve the purpose of trivialising certain steps, like writing to the console. It's here where conditional statements will be created.

Before compiling the NeoLang code a pre-compile loop is run where we check for information we need before compiling the rest of the code. Among these things are constants, variable declarations, and function declarations

```
Python

```

Variables

variables can be used to handle data and use it between function calls. You define a variable using the def keyword and it is then allocated an address in RAM. Note that variables can

only be 16 bits and you cannot use floating point numbers. You can also use & before the variable to get the address as an int.

We use the same dictionary data structure to store the different instructions as in DA:

```python
INSTRSET = {
    "write":Instruction("write", 1, InterpretWrite),
    "resetscreen":Instruction("resetscreen", 0,
InterpretResetScreen),
    "alloc":Instruction("alloc", 2, Allocate),
    "calc":Instruction("calc", 4, InterpretCalculate),
    "ptrval":Instruction("ptrval", 3, InterpretPtrVal),
    "chrinput":Instruction("chrinput", 1, InterpretInput),
    "return":Instruction("return", 1, InterpretReturn)}
```

Example instruction:

```
def x
write(x) #writes the ASCII value of x
```

Constants

constants were added in order to give some way of creating an immutable variable. You define a variable by using the keyword const and then writing the name of your constant separated by a space then the value of the constant. Note that constants can only be 16 bits and you cannot use floating point numbers.

Example implementation:

```
const PI 3
```

write()

write was one of the first instructions created in order to easily write a large string instead of having to manually write out every character.

Example implementation:

```
None
write(“Hello World!”)
```

Code:

```python
def InterpretWrite(args):
    lines = []
    if type(args[0]) == Variable:
        lines.append(f"MOV REG0 RM{args[0].addr}")
        lines.append("WRT REG0")
    else:
        for c in args[0]:
            lines.append("IMM " + str(ord(c)))
            lines.append("WRT REG0")
    return lines



def WriteToPos(pos: str, x) -> list:

    if type(x) == Variable:
        if (pos.startswith("RM")):
            return [f"MOV REG0 RM{x.addr}", f"MOV {pos} REG0"]
        else:
            return [f"MOV {pos} RM{x.addr}"]
    elif type(x) == int:
```

```
        if (pos != "REG0"):
            return [f"IMM {x}", f"MOV {pos} REG0"]
        else:
            return [f"IMM {x}"]
    elif type(x) == str:
        if len(x) == 1:
            return [f"IMM {ord(x[0])}", f"MOV {pos} REG0"]
        else:
            raise Exception("string too long to write")
    else:
        raise Exception(f"Cannot write value {x} to position
{pos}")
```

alloc()

alloc stands for allocate and will allocate memory to a variable if it does not already exist and will then assign a value to that variable. The first parameter is the variable (note that if the variable is not already defined you have to enter the variable name in the form of a string) and the second parameter is the value to assign.

Example implementation:

```
None
def a

alloc(“b”, 2)
alloc(a, b)
```

Code:

Python

```python
def Allocate(args: list):
    lines = []
    hasvarb = False
    foundvarb = None
    if type(args[0]) == Variable:
        hasvarb = True
        foundvarb = args[0]
    else:
        for varb in varbs:
            if (varb.kword == args[0]):
                hasvarb = True
                foundvarb = varb
                break

    writeval = ""
    if type(args[1]) == str:
        if len(args[1]) != 1:
            raise Exception("Invalid allocation parameter")
        else:
            writeval = str(ord(args[1][0]))
    elif type(args[1]) == Variable:
        if not hasvarb:
            foundvarb = Variable(args[0], FindFreeAddr())
            varbs.append(foundvarb)
        lines.append(f"MOV REG1 RM{args[1].addr}")
        lines.append(f"MOV RM{foundvarb.addr} REG1")
        return lines
    else:
        writeval = str(args[1])
    if (hasvarb):
        lines.append("IMM " + writeval)
        lines.append(f"MOV RM{foundvarb.addr} REG0")
    else:
        varbs.append(Variable(args[0], FindFreeAddr()))
        lines.append("IMM " + writeval)
```

```
        lines.append(f"MOV RM{varbs[len(varbs)-1].addr} REG0")
    return lines
```

calc()

calc stands for calculate. It is used to do arithmetic, bitwise operations and comparisons. The first parameter takes the variable to store the result into. The second and third parameter defines the values to use for the operation and the fourth parameter defines the operation to use (see DA CAL instruction for different operations).

Example implementation:

```
None
def x

alloc(“a”, 23)
alloc(“b”, 23)

calc(x, a, b, “eq”) #writes 1 to x since a and b are equal
```

Code:

```
Python
def InterpretCalculate(args): #arg0: variable to store result
into, arg1: number 1, arg2: number 2, arg3; operation
    lines = []
    lines.extend(WriteToPos("REG1", args[1]))
    lines.extend(WriteToPos("REG2", args[2]))
    lines.append("CAL REG1 REG2 " + args[3])
    lines.append(f"MOV RM{args[0].addr} REG0")
    return lines
```

ptrval()

ptrval is primarily used to enable pointers because it is able to write or read from a specific address in RAM. The first parameter defines the variable to store to if reading or to read from if writing to the address. The second parameter determines the address in memory to access and the third parameter is a string that can be either “r” or “w” which determines whether to read or write.

Example implementation:

```
None
def addr

alloc(“x”, 2)
ptrval(x, &addr, “w”) #writes the value 2 to addr
```

Code:

```python
def InterpretPtrVal(args): #arg0 = var to write/read from, arg1 = ram adress, arg2 = "r" or "w"
    lines = []
    if type(args[1]) == int:
        lines.append(f"IMM {args[1]}")
        lines.append(f"MOV REG1 REG0")
    elif type(args[1]) == Variable:
        lines.append(f"MOV REG1 RM{args[1].addr}")

    if (args[2] == "r"):
        lines.append("MOA 0 REG0 REG1")
        lines.append(f"MOV RM{args[0].addr} REG0")
```

```
    else:
        lines.append(f"MOV REG0 RM{args[0].addr}")
        lines.append("MOA 1 REG1 REG0")
    return lines
```

chrinput()

chrinput takes one character from the keyboard and writes the ASCII value to a variable. The first parameter takes the variable to write to. chr was added as a prefix to emphasise that this function only gets one character of input.
(see ASCII values here: https://www.ascii-code.com/)

Example implementation:

```
None
def x
chrinput(x)
write(x)
```

Code:

```
Python
def InterpretInput(args):
    lines = []
    lines.append("IMM 0")
    lines.append("MOV REG2 REG0")
    lines.append("MOV REG1 KBOARD")
    lines.append("CAL REG1 REG2 EQ")
    lines.append("JMI -4 REG0")
    lines.append("WRT REG1")
    lines.append(f"MOV RM{args[0].addr} REG1")
```

```
    return lines
```

return()

return will use the call stack to return to a point in the call stack. The first parameter takes how far in the call stack you want to go back, 0 meaning you go back 1. Note that you do not need to write return at the end of a function call since the interpreter automatically does that for you.

Example implementation:

```
None
def x

func Hello:
write(“Hello”)
chrinput(x)
if x:
return(1)
endif
write(“World”)
endfunc

Hello()
```

Code:

```
Python
def InterpretReturn(args):
    return [f"RET {int(args[0])}"]
```

Functions

functions can be used to use code several times or can be used for conditional statements such as if or while. To make a function you need to write func, then the function name and then a colon and then at the end of the function you need to write endfunc. To call the function, write the function name and a pair of parentheses. This will call the function and then add the current location to the call stack.

Example implementation

```
None
func Hello:
write(“Hello world”)
endfunc
Hello()
```

If statements

if statements are essentially extensions of functions. They take a variable and if the first bit of the variable is 1 then it will execute the code. Note that since they are an extension of functions they also add to the call stack.

Example implementation:

```
None
def x
alloc(x, 1)
if x:
write(“passed”)
endif
```

Import

The import keyword can be used to import external .neo files and enables easier management of large code bases. When using import do not write .neo at the end. It is automatically added

for you. It is important to note that import simply inserts the code of the imported file where the import was written.

Example Implementation:

```
None
Import Float
import IntInput
```

Code:

```python
if line.startswith("import"):
        importlines = []
        lines.pop(i)
        with open(f"{line.split(" ")[1]}.neo", "r") as f:
            importlines = f.readlines()
            importlines.reverse()
        for l in importlines:
            lines.insert(i, l.removesuffix("\n"))
        print(f"[Imported {line.split(" ")[1]}.neo]")

        continue
```

6. Result

This project has created a functioning CPU architecture that can be interacted with using code compiled by custom made programming languages, Digital Assembly and NeoLang. These were then used to create a calculator program.

Calculator Program

None

```
def tempnum
def a
def b
def op
def out
def temp
def inp
def bool
def bool2

func inputing:

chrinput(inp)
calc(bool, inp, 57, "GT")
calc(temp, inp, 48, "LT")
calc(bool, temp, bool, "OR")

#if input is not a number
if bool:
return(1)
endif

calc(tempnum, tempnum, 10, "MUL")
calc(inp, inp, 48, "SUB")
calc(tempnum, tempnum, inp, "ADD")
inputing()
endfunc

#input handling
write("please enter number 1\n")
inputing()
alloc(a, tempnum)
alloc(tempnum, 0)
write("please enter number 2\n")
inputing()
alloc(b, tempnum)
write("please enter the opertation\n")
chrinput(op)
write("\n")

calc(bool, op, "+", "EQ")
if bool:
calc(tempnum, a, b, "ADD")
endif

calc(bool, op, "-", "EQ")
if bool:
```

```
calc(tempnum, a, b, "SUB")
endif

calc(bool, op, "x", "EQ")
calc(temp, op, "*", "EQ")
calc(bool, bool, temp, "OR")
if bool:
calc(tempnum, a, b, "MUL")
endif

calc(bool, op, "/", "EQ")
if bool:
calc(tempnum, a, b, "DIV")
endif

write("your number is:\n")

calc(temp, tempnum, 10000, "DIV")
calc(out, temp, 48, "ADD")
calc(bool, temp, 0, "EQ")
calc(bool, bool, 0, "NOT")

if bool:
write(out)
endif

calc(temp, temp, 10000, "MUL")
calc(tempnum, tempnum, temp, "SUB")


calc(temp, tempnum, 1000, "DIV")
calc(out, temp, 48, "ADD")
calc(bool2, temp, 0, "EQ")
calc(bool2, bool2, 0, "NOT")
calc(bool, bool, bool2, "OR")

if bool:
write(out)
endif

calc(temp, temp, 1000, "MUL")
calc(tempnum, tempnum, temp, "SUB")


calc(temp, tempnum, 100, "DIV")
calc(out, temp, 48, "ADD")
calc(bool2, temp, 0, "EQ")
calc(bool2, bool2, 0, "NOT")
```

```
calc(bool, bool, bool2, "OR")

if bool:
write(out)
endif

calc(temp, temp, 100, "MUL")
calc(tempnum, tempnum, temp, "SUB")


calc(temp, tempnum, 10, "DIV")
calc(out, temp, 48, "ADD")
calc(bool2, temp, 0, "EQ")
calc(bool2, bool2, 0, "NOT")
calc(bool, bool, bool2, "OR")

if bool:
write(out)
endif

calc(temp, temp, 10, "MUL")
calc(tempnum, tempnum, temp, "SUB")


calc(out, tempnum, 48, "ADD")
write(out)
```

At the first part of this code we use the Inputing function to recursively look for a multi digit integer by checking if the character inputted is between the ASCII characters 48 and 57 (including 48 and 57), namely '0' and '9'[1]. After we confirm that we multiply the stored number by ten and then add the value of the input. So if we want to input the number 67:

We input the first character namely 6, which has the ASCII value 54 which is between 48 and 57. Then we multiply the current number, which has the value 0, by 10. Then we subtract 48 from the ASCII value to get 6 and add that to the number. For the 7 we do the same thing, multiply the number by 10, giving us 60 and then add 7 giving us 67. After that you simply need to input any character besides numbers

next we get the operation using a simple chrinput, not checking if the input is invalid, since it does not need to be a multi character string, and then we do the corresponding mathematical operation.

After that we need to write the number to the console. The problem is that if we simply use the write() function it will try to write the ASCII value of our calculated number, which will

be called x. So in order to fix this we will need to get each of the digits one by one. To accomplish this we can use the fact that a 16 bit integer can have a max value of 65,535 and we can also use integer division to our advantage.

Lets say x is 65,535. When we divide x by 10,000 we get the number 6, namely the first digit, due to the properties of integer division. We can then convert this to an ASCII number to print by simply adding 48. What we then need to do is to subtract the calculated number by 60,000 giving us 5,535. After that we repeat this process using 1,000, 100, and 10 as dividers until we reach 1 and can simply use the raw number.

However we run into a problem if we have a number like 24, because this would print out: 00024. To remove the 0s at the start of the number we add a boolean that if set to true will always print the number, otherwise it will not print the number, and we set it to true the first time the digit is not 0 and then using the bitwise OR operation on the bool on the following processes.

6.1 Analysis

This result can be considered a success, since we successfully created a programmable CPU and shows that it is indeed possible to digitally simulate a computer, which was the expected result. Something that could be possible given more effort is digitally simulating a CPU inside the digitally simulated CPU and so on. Although the simulated computer will always be slower and have less resources than the physical computer it is located on and therefore it has no practical uses. There are however several improvements that could be made to all levels of the project to make it faster and easier to work with.

As for the CPU, it seems to have few flaws, but a couple potential improvements as the CPU contains all components shown in the TOY-lite CPU in Princeton University's "Designing a CPU" lecture[2]. One of the most obvious improvements is to enable the CPU to use floating point numbers, since it currently contains no components to handle floating point numbers. In the current state of the project it is possible to artificially create floating point numbers using code, but this would be slower than having the required components on the CPU, which would be done by adding floating point mathematical operation components to the ALU.

Most potential improvements exist in the NeoLang language. Something that would be useful would be adding arrays to allow for larger size variables, this would then enable the addition of classes since the variables in a class can be made into a single array of variable, additionally this would make it possible to store an entire string into a variable instead of only one character per variable.

Another improvement that could be made would be not treating if statements the same as functions and thus allowing for easier handling of the call stack, since if statements also add to the call stack, since the way it is coded is that if you have an if statement inside a function

it will not return from the function but will instead return to the function from the if statment. A decision was made to make you able to return multiple lines back in the call stack to fix this but a better way would be to not use the call stack in if statements.

Also making local variables and functions would allow for easier handling and remembering of variables and functions, since in the current state of the language all variables and functions are globally callable after they are defined.

Along with other countless numbers of improvements that can be made to make the language more similar to traditional programming languages, these call for an even higher level language that compiles into NeoLang, something that could be named NL++. This language would preferably mimic the format of the C programming language to make it more recognizable to other programmers.

A major limitation however is that the only viable output medium is the text console, therefore to be able to make more types of programs you would need to create your own program, which includes more outputs, like sound or a screen, and therefore it would be theoretically possible to create all programs in a digital CPU given that the program, in which the CPU is made, has a sufficiently large amount of features.

6.2 Conclusions

The beginning of this report contains a question statement, which is “How can we create a CPU architecture digitally and use it to create a program with the use of programming languages?”

This can be answered as such:

First, create a CPU with all the components required in a TOY-lite CPU.

Second, create a programming language that can be more easily used than raw bytes.

Third, create a higher level programming language that makes certain things easier to implement.

After that you can continue creating higher level programming languages or you can create a program to compile to the CPU.

However it is important to note that if you are using Logisim you would not be able to create all forms of programs and would thus need to create your own program to build the CPU in before doing anything else.

Sources

1. ascii-code.com
2. “Designing a CPU” by Princeton University (cs.princeton.edu/courses/archive/fall09/cos126/lectures/22CPU-2x2.pdf)

Attachments

Open source GitHub repository: https://github.com/Neosell08/DLSASM