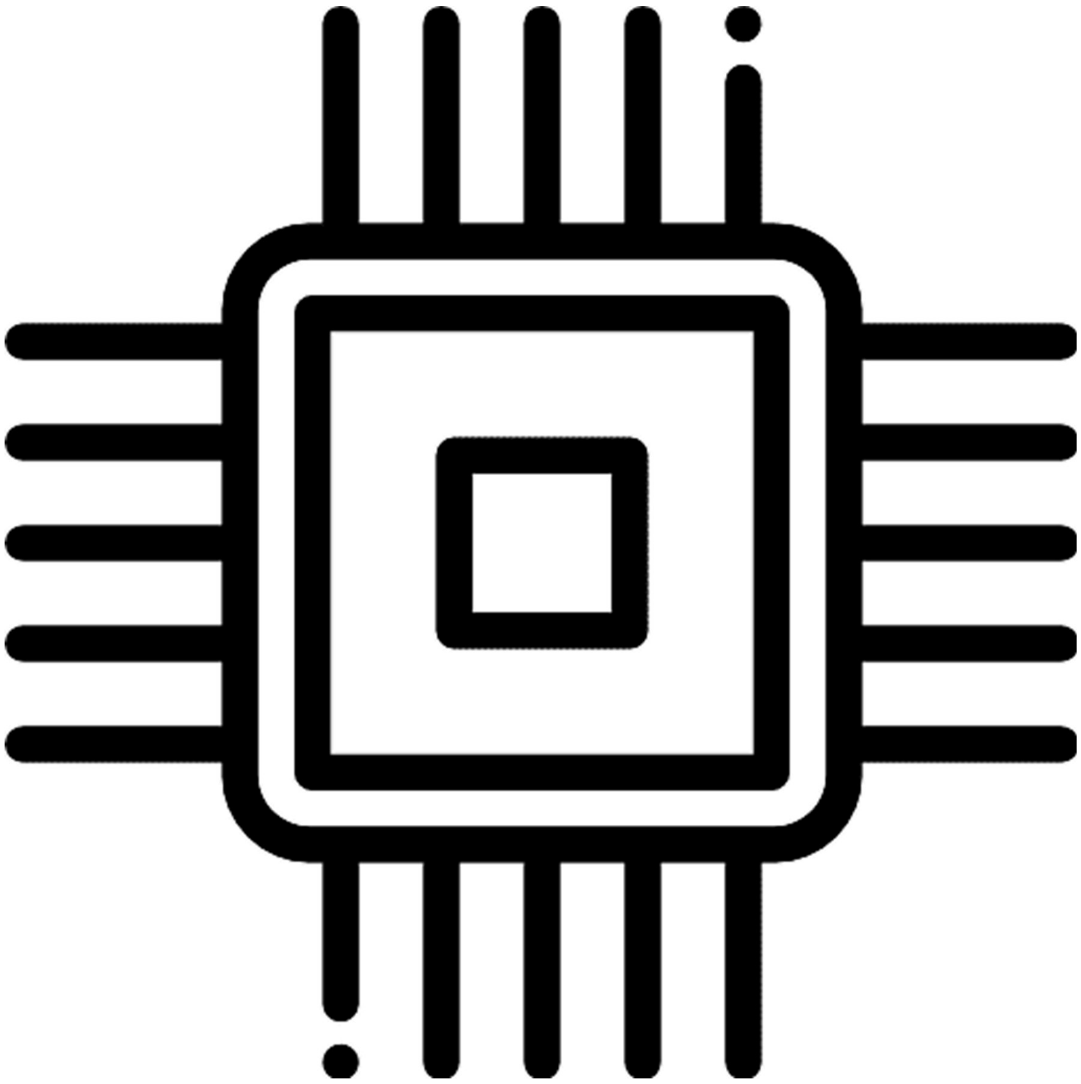
EMULATED PROGRAMMABLE CENTRAL PROCESSING UNIT



Saliaris Rafael 15 Mar 2025

Diploma student in Electrical and Computer Engineering
@ University of Patras

Contents

Prologue

The main premise of this project is to create a program capable of taking a file filled with bit-strings as input and to perform various actions such as: storing, displaying stored values, and performing mathematical and logical operations on stored values.

This is done in the Verilog programming language (a hardware description language a.k.a. HDL) as to fully encapsulate the process which an actual CPU goes through in order to perform such operations

Each directive is given as a string of 20bits. It has a total of 16 different directives, it operates on 8bit numbers [0:255], it has the ability to store up to 16 numbers in its registers, and has a ROM able to store 1024 distinct directives. It's also noteworthy that it has 3 flags, those being the **zero_flag** (last operation resulted in zero), the **jmp_flag** (last directive called for a jump), and the **overflow_flag** (last operation resulted in a memory overflow, meaning it resulted in an integer bigger than 255)

Basic functionality of a CPU

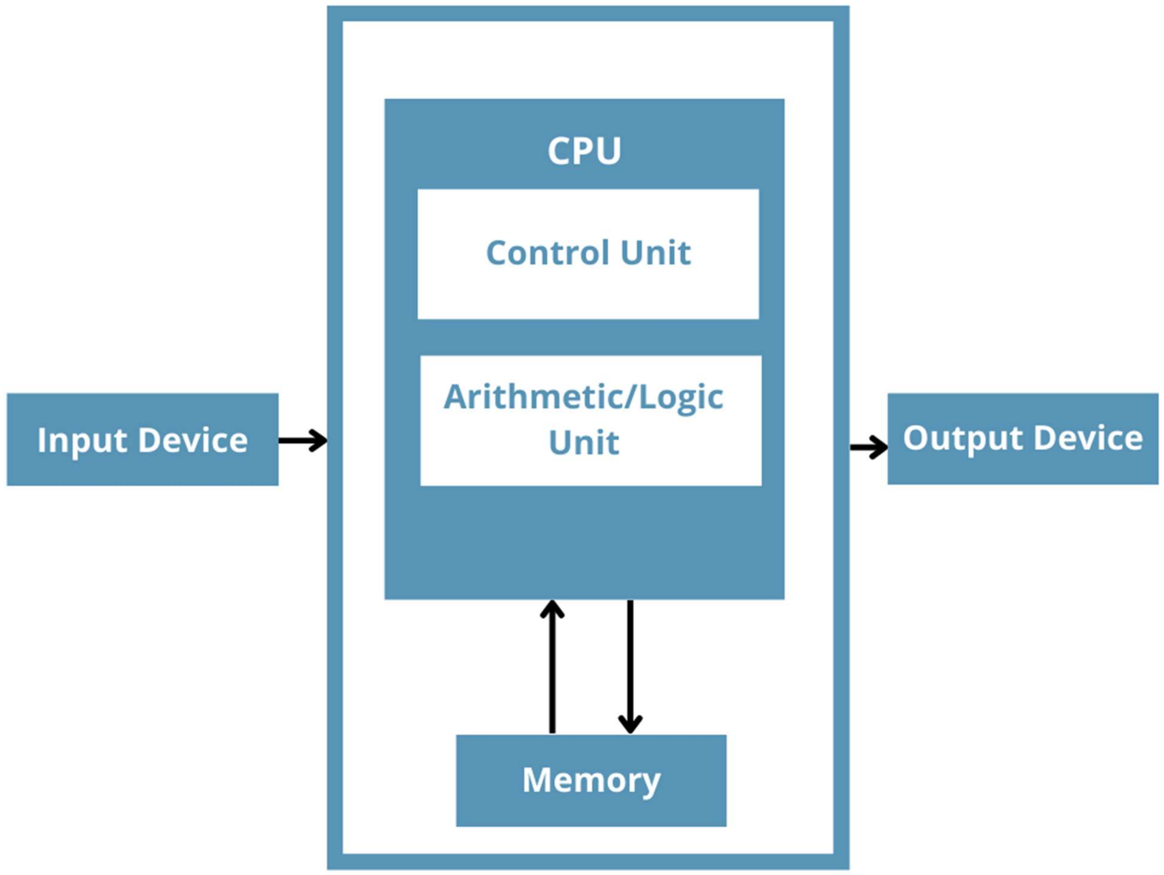
Directive Encoding

What exactly is a directive? It's a string of bits composed of 2 parts. The **OPCODE** (operation code) which signifies what action the CPU will take (store, add, compare, etc.), and the arguments which in turn can be split into parts. They usually are numeric inputs signifying integers or memory addresses.

These are what the Processing unit actually processes. They are taken by the Control Unit broken down based on the decoding protocol the specific processor uses and then executed based on the opcode and the various potential arguments.

Components

The average Processor is devised of 4 main components, a set of registers, a Read Only Memory (ROM) combined with a program counter (keeps track of the line the processor is reading), an Arithmetic Logic unit there to take care of the various mathematical and logical operations, and a Control Unit combined with a Clock in order to synchronize the components of the Processor so they work in harmony with one another.



1 Processor

Clock

The Clock is the most basic component inside the Processor alternating between 0 and 1 in a fixed rate, thus creating a frequency in which the computer works. Most things are made to function at the positive edge of the Clock (1), this allows the computer to take controlled steps towards completing an operation rather than chaotically *merge* all the values with one another, eliminating the possibly of using a value before its ready to be used.

Registers and ROM

The Registers and the ROM are where the values as well as directives are being stored and loaded from. The Registers are responsible for the management of *live* values, those being values that are being created during runtime. While the ROM is responsible for loading the inserted program into the Control Unit for further processing. Both of them load and store information on the positive edge of the clock.

Arithmetic Logic Unit (ALU)

The Arithmetic Logic Unit is responsible for the mathematical and logical operations that happen withing the Processor such as: addition, subtraction, logic AND, logic OR, logic XOR, logic NOT, left bit shifting, and right bit shifting. The unit by itself does not depend on the clock. Meaning that it will perform the operation that is selected no matter the state of the clock, though in my implementation the inputs of the ALU are clock dependent, meaning that even though the performance of the operation is independent of the clock, its performance of the current set of values depends on it, so essentially it still does stay synchronized with the clock.

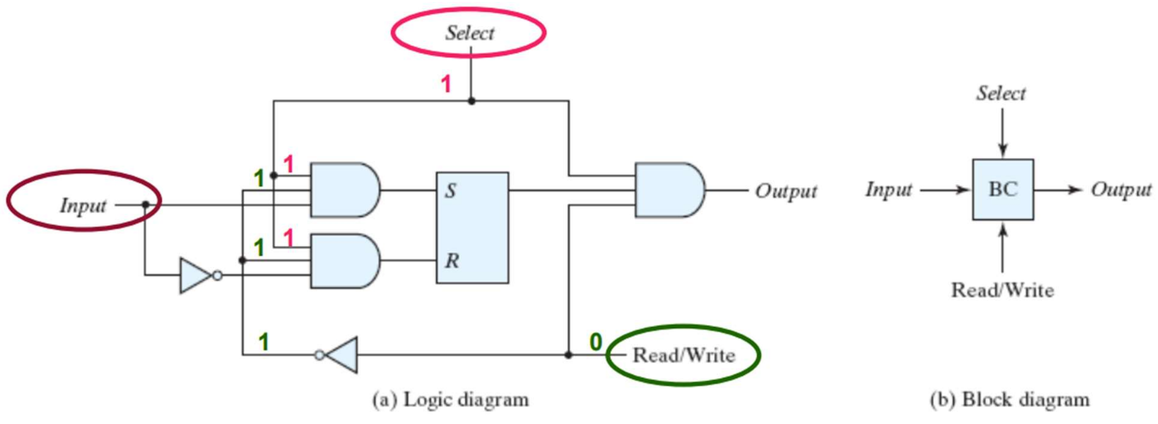
Control Unit

The Control Unit is the part of the Processor where the breaking down of the directives into its individual components takes place, also where the variety of the units are being synchronized in order for the directives to be correctly executed. Based on the **OPCODE** it choses what units to activate, in what order, and based on the arguments of the directive what their individual inputs should be. It is clock driven meaning that the next directive starts to execute one full clock cycle after the last directive ends.

In depth analysis of the functionality of memory modules

Memory Cell

A memory cell is the fundamental building block of any complicated Memory structure. It has the ability to store one bit, and later upon request read the store bit and output it.



2 Memory Cell

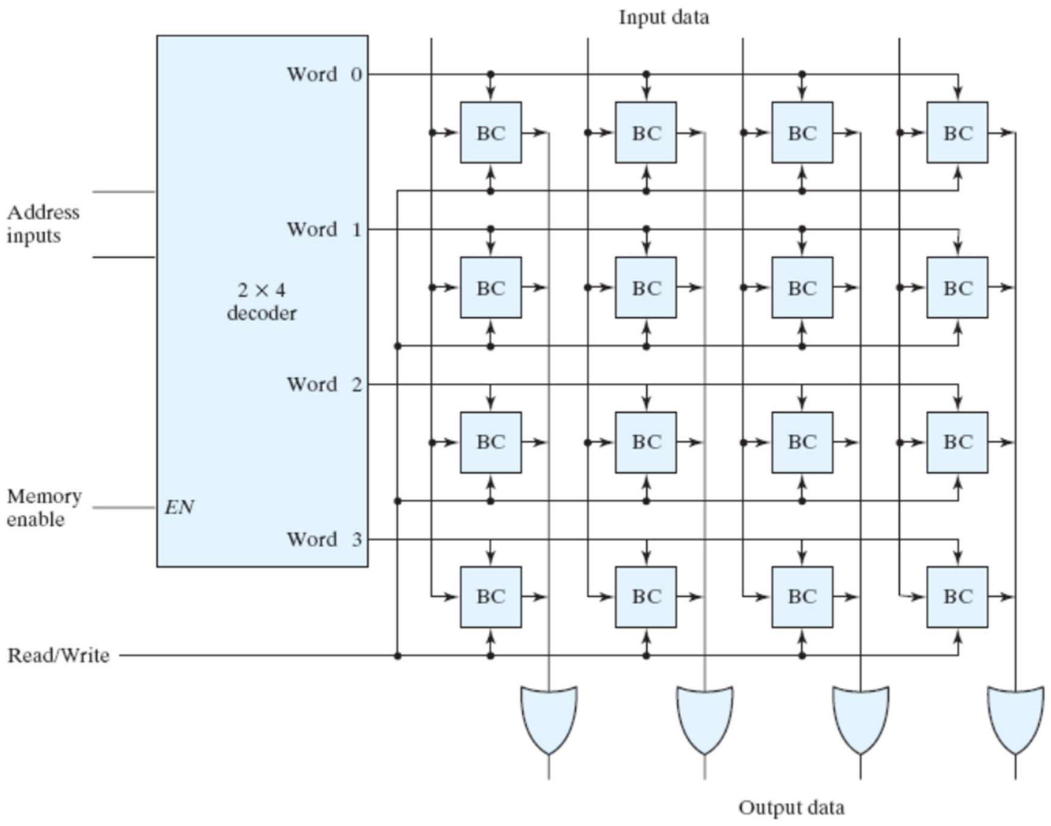
The actual storing is handled by an *SR flip flop*. *Select* activates the memory cell making it eligible for *Read/Write* operations, while *Select* is on low the *Output* is constantly 0. While *Select* is on high, if *Read/Write* is on low the *Output* of the *SR flip flop* is set to match the *Input*, if *Read/Write* is on high then the *Output* of the memory cell matches the values previously stored withing the *SR flip flop*.

Binary word in memory

The next step from a *Memory Cell* is to arrange them in such a way as to be able to save and load an entire string of bits in a memory. Simply done by having the desired number of cells sharing the same *Select* and *Read/Write* signals. And the *Input* and *Output* of each cell should be the bit it represents in the bit string (i.e. The first cell stores both the MSB of the *Input* and the *Output* and the last cell stores the LSB of both *Input* and *Output*).

Mass Memory Structure

If we take another step, we reach what is an *actual* computer Memory.



3 Mass Memory Structure

This is a (inefficient, yet the model I used for my implementation) 2-byte Memory (it stores 4 words 4 bits each). It works using a decoder as a *Select* signal. By inputting the address, while *Memory enable* is on high, it knows to select the corresponding word and perform the desired *Read/Write* operation. There are n OR gates, where n is the number of bits in a Word, all the nth bits are connected to the nth OR gate (i.e. all MSBs are connected to the OR gate that corresponds to the MSB of the *Output* data Register). When on *read mode* all the non-selected Words will be outputting a n bit long 0 signal, so the *Output* data will be equal the data held within the Selected Word.

Taking a look at the code

Here I negotiate the pivotal points of the code and their functionality; some parts which, if they were to be hardware constructed, would be rather simple. But due to the digital nature of this project I had to think of some work-arounds.

Registers and ROM

The only noteworthy thing about the implementation of the *Memory Cell* is the positive clock edge condition for the *Output*. Where it sets the *Output* and the logical and between *Select*, *Read/write*, and the *SR-Output*.

```
always @(posedge clk)
begin

    out <= rw & select
& SR_out;

end
```

What should also be taken into account is the way the *Memory Cell* modules attach to one another; the entirety of the *Word* module is implemented within this one *generate-for-loop*.

```
genvar i;

generate

    for (i=0; i<N; i = i+1) begin

        MemoryCell memcell_i(.clk(clk), .in(in_word[i]),
.select(select), .rw(rw), .out(out_word[i]));

    end

endgenerate
```

As far as the larger memory units go, the implementation is the same, instead of *Memory Cells* though we use *Words*. The interesting part is how the *Output Word* is being Calculated.

```
always @(posedge clk) begin
: finalization

    integer j;

    if (memoryEN) begin

        out_word =
{N{1'b0}};

        for (j=0; j<16; j =
j + 1) begin

            out_word =
out_word|words[j];

        end

    end else begin

            out_word =
{N{1'b0}};

    end

end

endmodule
```

If the memory is not enabled then the output is 0 all across. If the memory is enabled it assumes an output (`out_word`) which initially is set to zero. And then it passes it through an OR gate with all the other *Word* outputs in the memory. This leads into `out_word` having the value of the *Selected Word*.

Arithmetic Logic Unit (ALU)

The outputs of the ALU are being handled by a case clause (a.k.a. switch in most programming languages) emulating a decoder. There are two ALU instances a 1bit ALU and a Nbit ALU comprised of N-1bit ALUs. There is a peculiar piece of code I wrote, which should handle the first ALU in the series of N ALUs, that make up the larger unit.

```
wire default_control_flag = (control_first_in_series_flag === 1'bz) ?
1'b1 : control_first_in_series_flag;

wire ccin = (default_control_flag) ? ((select==4'b0010) ? 1'b1 : 1'b0)
: cin;

FA adder(.A(A), .B((select==4'b0010) ? ~B : B), .cin(ccin),
.out(result), .cout(ccout));
```

The `control_first_in_series_flag` is an optional input that must be given as high to the first ALU unit of the ALU chain. The `default_control_flag` is there to format the individual units in a manner that makes sense. It states that if the `control_first_in_series_flag` is unprovided then the `default_control_flag` should be set to high else it should have the same value as the `control_first_in_series_flag`. Now if the `default_control_flag` is set to low (meaning this unit is not the first in the series) then the unit should settle with the provided *cin* (carry-in) as its main *cin* (*ccin*) otherwise it should be set to either 0 or 1 depending on the chosen operation (1 for subtraction, 0 for everything else). Then a *Full Adder* is being prompted with inputs A and B or the conjugate of B (if the selected operation is subtraction) along with the *carry ins* and the proper channels for the outputs and *carry outs*.

Control Unit

The Control Unit in my program is also responsible for loading the machine code into the ROM, something which is done with built-in Verilog functions. Once the program has been loaded its execution will begin. I will provide the way a value is loaded into the registers since every other directive is a variation of that.

```
ENABLE_MEMORY = 1;

#50;

RAM_RW = 0;

MEMORY_IN = ARG1;

address = ARG2;

#50;

ENABLE_MEMORY = 0;
```

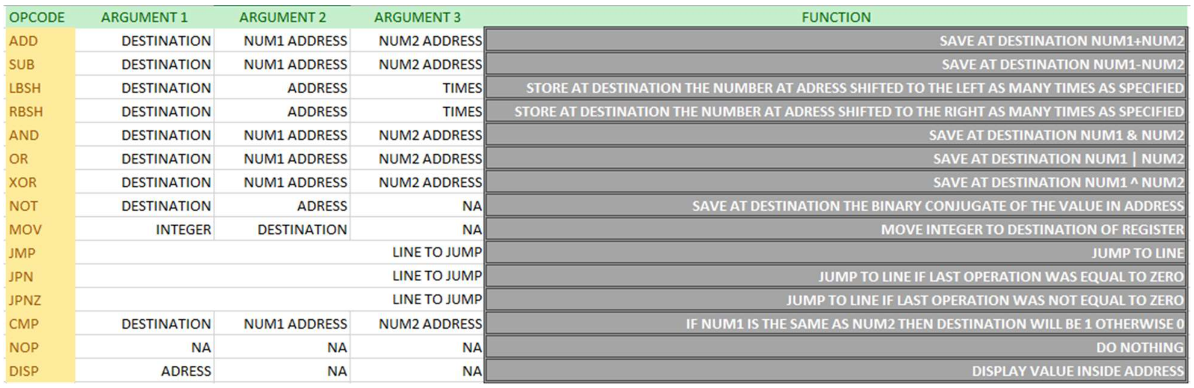
The Memory registers are being enabled, we wait for 50-time units (for safety) in order to have a full clock cycle go through. We set the registers to *Write* (meaning we will save a value to Memory), what we save (`MEMORY_IN`) is set to the first argument of our formatted directive and the address of where its stored is the second argument, we again wait for 50-time units for the data to go through and then we disable Memory so that it does perform unauthorized actions in the background.

Custom assembly and Assembler

Before I get into the assembler, I should to mention how the directives are actually set up: Each directive is a string of 20 bits, the first 4 bits are the **OPCODE** (meaning there is a total of 16 instructions), the next 8 bits is argument 1, usually used for integer input, the next 4 bits are argument 2, and the last 4 bits are argument 3. Argument 2 and 3 are usually used to specify location in memory, since the memory has 16 locations ([0:15]) only 4 bits are needed.

Custom assembly-like language rules and directives

There are 3 basic rules: Semicolon (;) at the end of every line, comments can go only after a semicolon and don't require a special initiator (they don't need symbols like # % // etc.), and no empty lines if you want to leave an empty like just type a semicolon (;) and the assembler will register it as the "do nothing" directive.



OPCODE	ARGUMENT 1	ARGUMENT 2	ARGUMENT 3	FUNCTION
ADD	DESTINATION	NUM1 ADDRESS	NUM2 ADDRESS	SAVE AT DESTINATION NUM1+NUM2
SUB	DESTINATION	NUM1 ADDRESS	NUM2 ADDRESS	SAVE AT DESTINATION NUM1-NUM2
LBSH	DESTINATION	ADDRESS	TIMES	STORE AT DESTINATION THE NUMBER AT ADRESS SHIFTED TO THE LEFT AS MANY TIMES AS SPECIFIED
RBSH	DESTINATION	ADDRESS	TIMES	STORE AT DESTINATION THE NUMBER AT ADRESS SHIFTED TO THE RIGHT AS MANY TIMES AS SPECIFIED
AND	DESTINATION	NUM1 ADDRESS	NUM2 ADDRESS	SAVE AT DESTINATION NUM1 & NUM2
OR	DESTINATION	NUM1 ADDRESS	NUM2 ADDRESS	SAVE AT DESTINATION NUM1 | NUM2
XOR	DESTINATION	NUM1 ADDRESS	NUM2 ADDRESS	SAVE AT DESTINATION NUM1 ^ NUM2
NOT	DESTINATION	ADRESS	NA	SAVE AT DESTINATION THE BINARY CONJUGATE OF THE VALUE IN ADDRESS
MOV	INTEGER	DESTINATION	NA	MOVE INTEGER TO DESTINATION OF REGISTER
JMP			LINE TO JUMP	JUMP TO LINE
JPN			LINE TO JUMP	JUMP TO LINE IF LAST OPERATION WAS EQUAL TO ZERO
JPNZ			LINE TO JUMP	JUMP TO LINE IF LAST OPERATION WAS NOT EQUAL TO ZERO
CMP	DESTINATION	NUM1 ADDRESS	NUM2 ADDRESS	IF NUM1 IS THE SAME AS NUM2 THEN DESTINATION WILL BE 1 OTHERWISE 0
NOP	NA	NA	NA	DO NOTHING
DISP	ADRESS	NA	NA	DISPLAY VALUE INSIDE ADDRESS

4 Custom assembly-like language rules and directives

These are all the directives, and how they are used (NA means No Attribute). For example, if I wanted to add the numbers stored in cell 5 and cell 7 and store the sum in cell 13, I would write "ADD 13 5 7;".

Also, there is the EOF command used to signify to the assembler where the end of the file is (written as "EOF;" in code).

Regarding the aforementioned flags: Whenever a numerical operation is being run, whenever it results in zero, the `zero_flag` is set to *high* allowing the use of JPNZ, whenever a jump directive is being used the `jmp_flag` is enabled which ensures the correct modification of the program counter, and whenever though a numerical operation the result is a number bigger than 255 the `overflow_flag` is set to high halting and finishing the program.

Assembler functionality

The assembler is a tool I built in the C programming language capable of compiling the human readable pseudo-assembly into the Processor's machine code. It does that by reading the `.asmm` file into an array of strings and excluding the semicolons (;) (they are there to tell the assembler when to stop reading). After that is done breaking the directives and encoding them is a matter of string operations.

Writing and running a program using the CPU emulator

The first step would be creating a file called specifically `program.asmm`, after that open the file with your desired text editor and write your code.

Example code for Fibonacci numbers:

```
MOV 1 0; current number of fib sequence

MOV 0 1; last number of the fib sequence

MOV 0 2; permanent zero as to move numbers with add or sum

MOV 0 3; cell to store previous fib numbers

;

ADD 3 0 2; move the current fib number to cell 3

SUB 4 0 3; sub current fib number from its storing cell, it can
only happen once per iteration

JPN 13;    do it so we may display it once, if sub is zero go to
line 13

ADD 0 0 1; time to find the next fib by adding the current and
previous

ADD 1 3 2; move current fib to place where the last fib number
is stored

JMP 6;    jump to line 6 in order to repeat the cycle

;

DISP 0;    display current fib

JMP 9;     jump back to line 9

EOF;
```

After having written the code in your `program.asmm` file, time to assemble it.

Do that my compiling your `assembler.c` file with your preferred C compiler and then run (on Windows) “**`./assembler 1 program.asmm`**” it will let you know if it was successful.

Afterwards you would also have to compile the ControlUnit.v file with your preferred Verilog Compiler and then drop the generated program.lc file in the same directory as the ControlUnit.out. using Icarus I compile it doing “**`iverilog -o ControlUnit.out ControlUnit.v`**” and then run it by doing “**`vvp ControlUnit.out`**”

With this specific program the desired output is the 13 first Fibonacci numbers.

Epilogue

In summary this project consists of a ROM, 16 Registers, an 8bit ALU, and a Control Unit. All simulated in the HDL Verilog working in unison as to emulate the real procedures a real CPU goes through in order to complete certain tasks and programs.

This serves little to no actual, and practical, purpose other than helping someone become better acquainted with both the workflow of a CPU and the use of assembly languages.