Computer Emulator Documentation

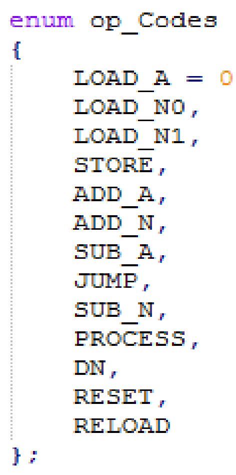
Samyuel Danyo

17.04.2017

ABSTRACT: The computer emulator implements the functionality of a Von Neumann machine, fetching and executing from memory, extended with a cache memory. The emulator is able to process simple programs through its instruction set, displaying the exact cause, place and data in the machine if an error occurs. Moreover, it allows the use of a debug mode, which captures the value of all internal signals, register, validations, functions and errors, presenting the opportunity of full analysis of the computer's functions.
The von Neumann architecture uses a single memory for data and program instructions, which lets the program to alter its own code. The system's flow is facilitated by the control unit- kind of a state machine, issuing signals to control the communication between registers, memory, ALU.
I started by making the instruction set, register topology and internal signals set. I built the emulator as a class in a namespace – I continued by designing a few functions, which are needed for creating a new computer, initialising it, initialising the RAM and cache, loading a program – including: entering instruction, checks for a valid instruction/data and displaying registers, signals and memory. Later, I designed the execution cycle functions for each operational code, building the sequencer. Last, I added more validation checks and implemented the debug mode.
At first the computer was loading and storing directly via the RAM. However, as an extension I added cache memory and optimised the loading/storing execution cycles.

1. Instruction Set design.

I used enumerated type to set the op-codes, their assembly codes can be seen in (Figure 1). The word length of mu CPU is 16 bits. If it is an instruction – the first byte is the op-code, the second is the data/address. The focus was on simple functions: LOAD_A – loading from an address, LOAD_N0 – loading a number in the more significant byte of the accumulator, LOAD_N1 – loading a number in the less significant byte of the accumulator, STORE – storing the value of the ACC in an address, ADD_A – adding the content of an address to the value of the ACC, ADD_N – adding a number, SUB_A – subtract from an address, SUB_N – subtract a number, JUMP – jump to an address, PROCESS – turns off the signal (processing) - shutting down, DN – doing nothing for a cycle, RESET – resets the registers (Not the RAM, CACHE) – starts the program from the beginning, RELOAD – reloads the program(initialising, RAM & CACHE).



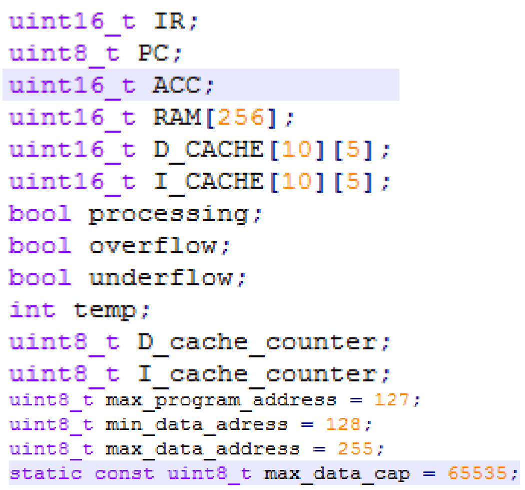
```
enum op_Codes
{
    LOAD_A = 0,
    LOAD_N0,
    LOAD_N1,
    STORE,
    ADD_A,
    ADD_N,
    SUB_A,
    JUMP,
    SUB_N,
    PROCESS,
    DN,
    RESET,
    RELOAD
};
```

Figure 1: Op codes set.

2. Register topology and internal signals.

I stuck to simple architecture, embodying: Instruction Register, Program Counter, Accumulator, Random Access Memory, one level CACHE with data (D_CACHE) and instruction (I_CACHE) sub-caches and a number of signals/registers facilitating the CPU function. All of these can be seen in (Figure 2).
The Instruction register (IR) holds the program instruction being executed in the cycle (in the second byte is the op-code, in the first byte is the number or address that needs to be accessed during execution). The Program Counter (PC) holds the address of the program instruction, being executed. The Accumulator (ACC) holds data from previous instruction or a number/data from an address being loaded in the cycle. The RAM is the main memory of the computer, storing the program instructions and the data. The CACHE is a secondary memory (much faster), which holds the most used instructions and data – speeding up the CPU's processing. I use associative mapping with FIFO replacement procedure. The cache_counters are holding the available line for storing new

information. The temporary register (temp) holds a temporary value that is used in a mathematical function or loading. The processing signal determines if the control unit will function, if it is true – the CPU is working, if – false, the CPU shuts down. Overflow is a signal going true if in addition function the data is too big to hold in the RAM – the data gets corrupted and the CPU stops working. Underflow is analogical to overflow but for subtracting. The RAM has 256 address of two-byte size. This shows the meaning of the max_data_cap of a single entry in the memory, the max_addresses are initialised to values, which assume full used of the memory and equal program memory and data size. Those are being changed when loading a particular program, according to its size and use of memory. The IR and ACC are both 2-byte sized, since they hold entries of the RAM (instructions or data). The PC is 1-byte sized, since the max address in the RAM is 255. Both sub-caches have ten lines, holding tag (the address of the memory block) and ten memory blocks, each comprised of four instructions/data entries), building up the CACHE to being able to store forty instructions and forty data entries. The cache_counters are 1-byte sized, since they hold the cache line address.



```
uint16_t IR;
uint8_t PC;
uint16_t ACC;
uint16_t RAM[256];
uint16_t D_CACHE[10][5];
uint16_t I_CACHE[10][5];
bool processing;
bool overflow;
bool underflow;
int temp;
uint8_t D_cache_counter;
uint8_t I_cache_counter;
uint8_t max_program_address = 127;
uint8_t min_data_adress = 128;
uint8_t max_data_address = 255;
static const uint8_t max_data_cap = 65535;
```

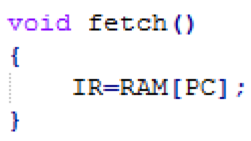
Figure 2: Register topology and signals.

3. Control Unit design.

The control unit's sequencer has three distinct stages. First – fetching the program instruction, second – decoding the instruction (matching the op-code and so, understanding what needs to be done in execution), and last – execution of the operation, which flow depends on the operation and incrementing the PC (setting the address for the next instruction). The last stage can me omitted or changed for different op-codes. After execution, the sequencer goes back to checking if processing is one, if yes, the cycle starts again. The sequencer loop can be seen in (Figure 5).

3.1. Fetch stage.

The fetch stage comprises of simply loading the IR with the RAM entry (instruction word) with address [PC]. The function can be seen in (Figure 3).



```
void fetch()
{
    IR=RAM[PC];
}
```

Figure 3: Fetch function.

3.2. Decode Stage.

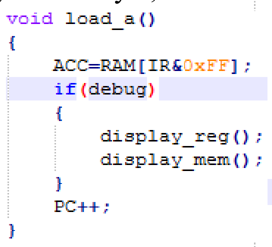
The decoding comprises of getting the op-code from the more significant byte of the IR and recognising which operation it holds. I accomplish that by masking the second byte of the IR and dividing by 256, which is equivalent to shifting one byte to the right, then I use a switch () to recognise the function. The process can be seen in (Figure 5).

3.3. Execution stage.

In this stage, the execution flow of the operation is followed. Each operation has its own execution flow, which is carried out by its execution function.

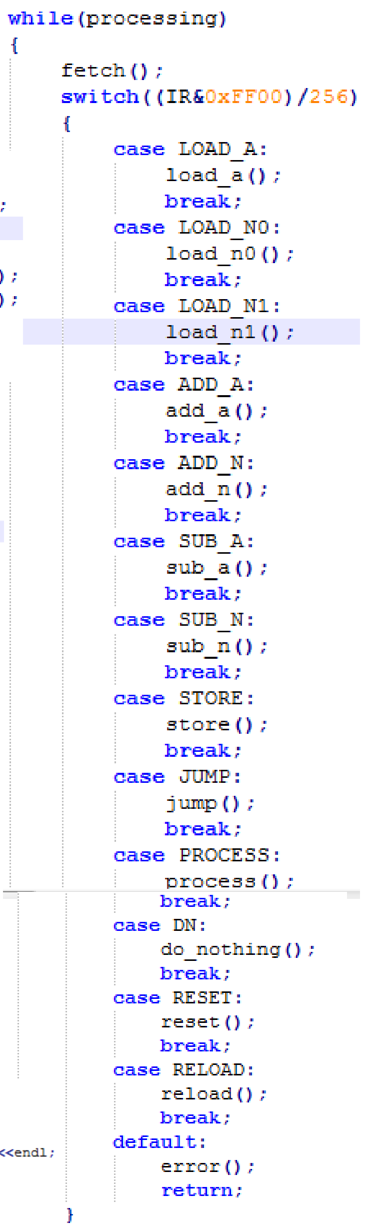
3.3.1. Load ACC from an address (LOAD_A) execution flow.

As seen in (Figure 4) the flow is: storing in ACC the RAM entry, which is on the address, held in the IR’s less significant byte, later the PC is incremented to the next instruction.



```
void load_a()
{
    ACC=RAM[IR&0xFF];
    if(debug)
    {
        display_reg();
        display_mem();
    }
    PC++;
}
```

Figure 4:
(LOAD_A) execution flow.

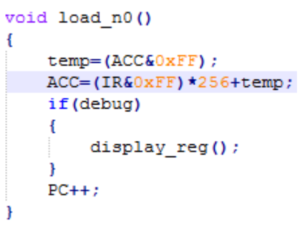


```
while(processing)
{
    fetch();
    switch((IR&0xFF00)/256)
    {
        case LOAD_A:
            load_a();
            break;
        case LOAD_N0:
            load_n0();
            break;
        case LOAD_N1:
            load_n1();
            break;
        case ADD_A:
            add_a();
            break;
        case ADD_N:
            add_n();
            break;
        case SUB_A:
            sub_a();
            break;
        case SUB_N:
            sub_n();
            break;
        case STORE:
            store();
            break;
        case JUMP:
            jump();
            break;
        case PROCESS:
            process();
           break;
        case DN:
           do_nothing();
           break;
        case RESET:
           reset();
           break;
        case RELOAD:
           reload();
           break;
        default:
           error();
           return;
    }
```

Figure 5:
CU loop.

3.3.2. Load ACC’s more sign. byte with a number (LOAD_N0) execution flow.

As seen in (Figure 6) the flow is: storing in temp the less sign. byte of the ACC, loading the ACC with value of temp added to the number in the less sign. byte of the IR (this is done so the first byte of the ACC is not lost), later the PC is incremented to the next instruction.



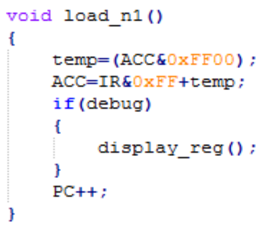
```
void load_n0()
{
    temp=(ACC&0xFF);
    ACC=(IR&0xFF)*256+temp;
    if(debug)
    {
        display_reg();
    }
    PC++;
}
```

Figure 6:
(LOAD_N0) execution flow.

3.3.3. Load ACC’s less sign. byte with a number (LOAD_N1) execution flow.

As seen in (Figure 7) the flow is: storing in temp the more sign. byte of the ACC, loading the ACC with value of temp added to the number in the less sign. byte of the IR (this is done so the second byte of the ACC is not lost), later the PC is

incremented to the next instruction.

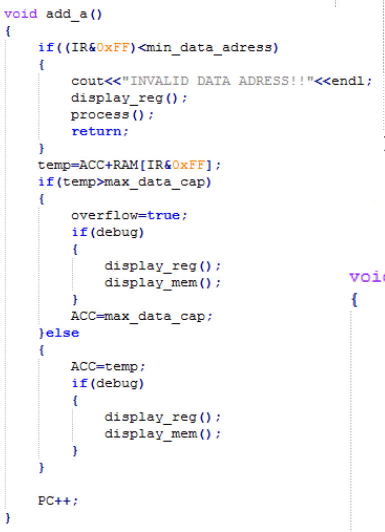


```
void load_n1()
{
    temp=(ACC&0xFF00);
    ACC=IR&0xFF+temp;
    if(debug)
    {
        display_reg();
    }
    PC++;
}
```

Figure 7:
(LOAD_N1) execution flow.

3.3.4. Add the data on an address to the ACC (ADD_A) execution flow.

As seen in (Figure 8) the flow is: storing in temp the addition of the ACC and data entry of the RAM on the address, held by the less sign. byte of the IR, storing in ACC the value of temp and incrementing the PC to the next instruction.

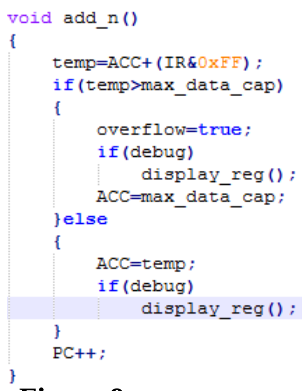


```
void add_a()
{
    if((IR&0xFF)<min_data_adress)
    {
        cout<<"INVALID DATA ADRESS!!"<<endl;
        display_reg();
        process();
        return;
    }
    temp=ACC+RAM[IR&0xFF];
    if(temp>max_data_cap)
    {
        overflow=true;
        if(debug)
        {
            display_reg();
            display_mem();
        }
        ACC=max_data_cap;
    }else
    {
        ACC=temp;
        if(debug)
        {
            display_reg();
            display_mem();
        }
    }

    PC++;
}
```

Figure 8:
(ADD_A) execution flow.

3.3.5. Add number to the ACC (ADD_N) execution flow.

As seen in (Figure 9) the flow is: storing in temp the addition of the ACC and the number, held by the less sign. byte of the IR, storing in ACC the value of temp and incrementing the PC to the next instruction.

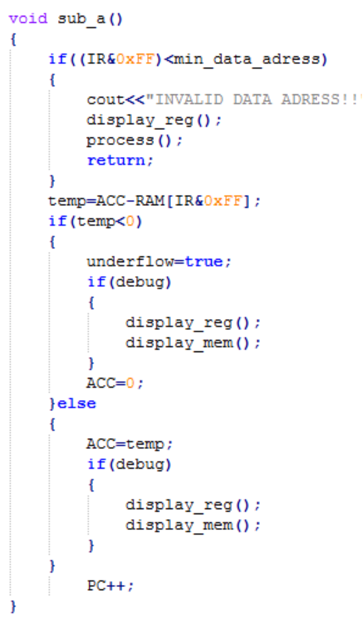


```
void add_n()
{
    temp=ACC+(IR&0xFF);
    if(temp>max_data_cap)
    {
        overflow=true;
        if(debug)
            display_reg();
        ACC=max_data_cap;
    }else
    {
        ACC=temp;
        if(debug)
            display_reg();
    }
    PC++;
}
```

Figure 9:
(ADD_N) execution flow.

3.3.6. Subtract the data on an address from the ACC (SUB_A) execution flow.

As seen in (Figure 10) the flow is: storing in temp the subtraction of the ACC and data entry of the RAM on the address, held by the less sign. byte of the IR, storing in ACC the value of temp and incrementing the PC to the next instruction.

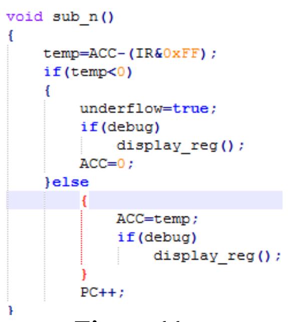


```
void sub_a()
{
    if((IR&0xFF)<min_data_adress)
    {
        cout<<"INVALID DATA ADRESS!!
        display_reg();
        process();
        return;
    }
    temp=ACC-RAM[IR&0xFF];
    if(temp<0)
    {
        underflow=true;
        if(debug)
        {
            display_reg();
            display_mem();
        }
        ACC=0;
    }else
    {
        ACC=temp;
        if(debug)
        {
            display_reg();
            display_mem();
        }
    }
        PC++;
}
```

Figure 10: (SUB_A) execution flow

3.3.7.Subtract number from the ACC (SUB_N) execution flow.

As seen in (Figure 11) the flow is: storing in temp the subtraction of the ACC and the number, held by the less sign. byte of the IR, storing in ACC the value of temp and incrementing the PC to the next instruction.

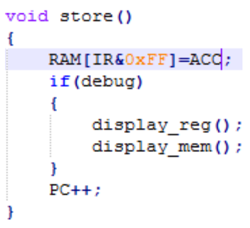


```
void sub_n()
{
    temp=ACC-(IR&0xFF);
    if(temp<0)
    {
        underflow=true;
        if(debug)
            display_reg();
        ACC=0;
    }else
        {
            ACC=temp;
            if(debug)
                display_reg();
        }
        PC++;
}
```

Figure 11: (SUB_N) execution flow.

3.3.8.Store the data from ACC in the RAM. (STORE) execution flow.

As seen in (Figure 12) the flow is: saving the value of the ACC in the RAM on an address held by the less sign. byte of the IR and incrementing the PC to the next instruction.

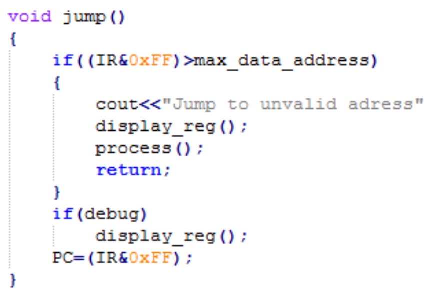


```
void store()
{
    RAM[IR&0xFF]=ACC;
    if(debug)
    {
        display_reg();
        display_mem();
    }
    PC++;
}
```

Figure 12: (STORE) execution flow.

3.3.9. Jump to an instruction (JUMP) execution flow.

As seen in (Figure 13) the flow is: updating the value of the PC with the address held by the less sign. byte of the IR and incrementing the PC to to the next instruction.

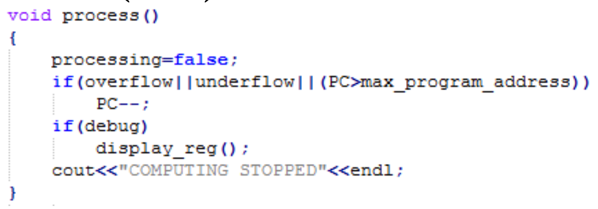


```
void jump()
{
    if((IR&0xFF)>max_data_address)
    {
        cout<<"Jump to unvalid adress"
        display_reg();
        process();
        return;
    }
    if(debug)
        display_reg();
    PC=(IR&0xFF);
}
```

Figure 13: (JUMP) execution flow.

3.3.10. Shut down the CPU (PROCESS) execution flow.

As seen in (Figure 14) the flow is: updating the value of processing to false, displays the shutdown, it also checks if the function process is called as an instruction or because of an error/EOP. If it is called because as an operation in an instruction it just updates the processing signal, if not it decrements the PC to the erroneous instruction.

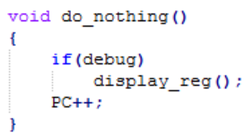


```
void process()
{
    processing=false;
    if(overflow||underflow||(PC>max_program_address))
        PC--;
    if(debug)
        display_reg();
    cout<<"COMPUTING STOPPED"<<endl;
}
```

Figure 14: (PROCESS) execution flow

3.3.11. Do nothing for a cycle (DN) execution flow.

As seen in (Figure 15) the flow is: just incrementing the PC to the next instruction.

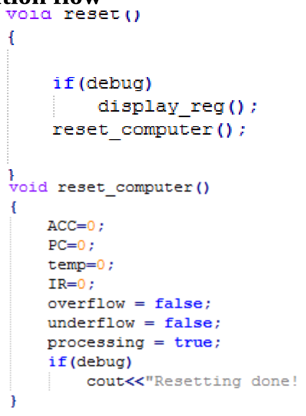


```
void do_nothing()
{
    if(debug)
        display_reg();
    PC++;
}
```

Figure 15: (DN) execution flow.

3.3.12. Start the program from beginning (RESET) execution flow.

As seen in (Figure 16) the flow is: initialising the registers to zero and the signals to operational values.

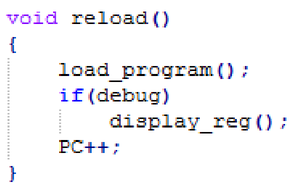


```
void reset()
{

    if(debug)
        display_reg();
    reset_computer();

}
void reset_computer()
{
    ACC=0;
    PC=0;
    temp=0;
    IR=0;
    overflow = false;
    underflow = false;
    processing = true;
    if(debug)
        cout<<"Resetting done!
}
```

Figure 16: (RESET) execution flow

3.3.13. Reload the program (RELOAD) execution flow.

As seen in (Figure 17) the flow is: calling the support func. load_program (Figure 21), which loads the program in the memory and updates the max/min_address/data values, then the PC is incremented to the next instruction.

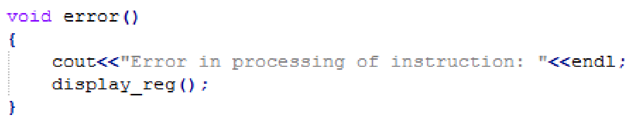


```
void reload()
{
    load_program();
    if(debug)
        display_reg();
    PC++;
}
```

Figure 17: (RELOAD) execution flow

3.3.14. Not recognised op-codes display error (default).

As seen in (Figure18) the flow is: display error message and dump the registers (Figure 22) so the user can see the faulty instruction.

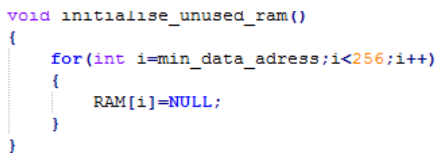


```
void error()
{
    cout<<"Error in processing of instruction: "<<endl;
    display_reg();
}
```

Figure 18: (op-code error) exe flow.

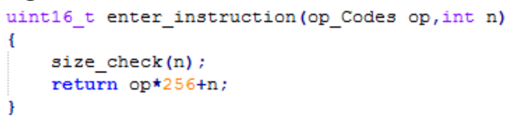
4. Support functions design.

I wrote a few support functions for the computer emulator class, two of them are just a constructor and deconstruct, while essential these are no-brainers. I wrote the reset_computer () function, which can be seen in (Figure 16), since it is used in the (RESET) operation. Its task is to zero-initialise the registers (not CHACHE or RAM) and initialise the internal signals to ready-to-process values. It is used to set the computer ready after declaring it. The next function is initialise_unused_ram (), seen in (Figure 19), the name speaks for its function. The next support function is enter_instruction (), it makes the storing of the program isntructions in the RAM easy, seen in (Figure 20). Next is the load_program () function. It loads one of the programs from the “hard drive” by user’s choice on the RAM, using enter_instruction (), it also sets the max/min_data/program_adress values and initialises the unused RAM, seen in (Figure 21). The next functions are for displaying: display_reg () – seen in (Figure 22), display_mem () – seen in (Figure 23), display_ram () – seen in (Figure 24).



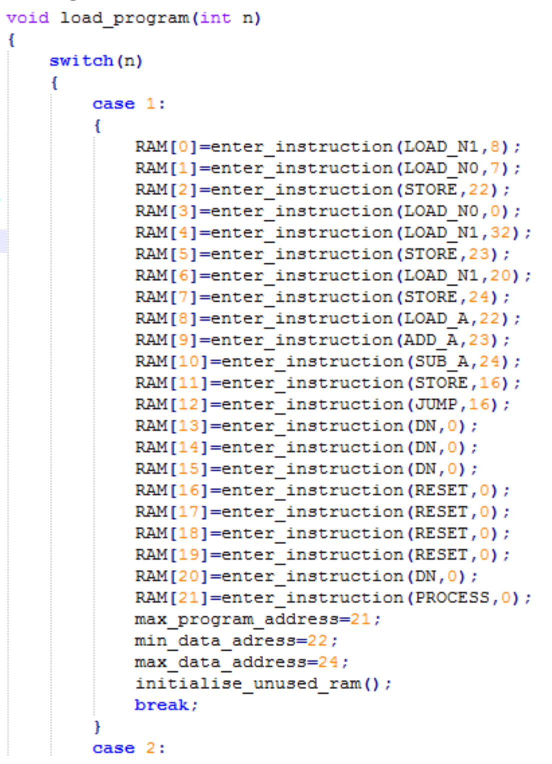
```
void initialise_unused_ram()
{
    for(int i=min_data_adress;i<256;i++)
    {
        RAM[i]=NULL;
    }
}
```

Figure 19: Init. unused RAM func.



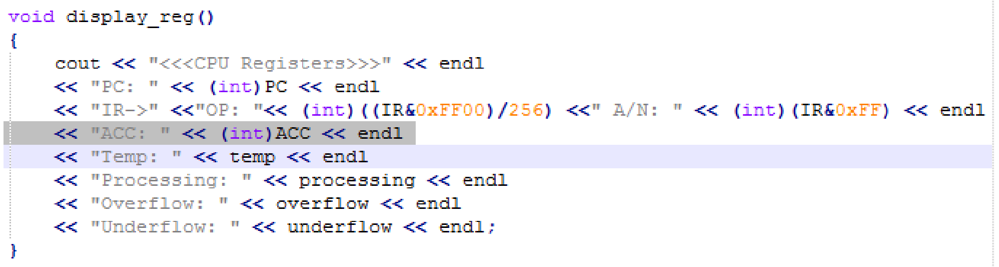
```
uint16_t enter_instruction(op_Codes op,int n)
{
    size_check(n);
    return op*256+n;
}
```

Figure 20: Enter_instruction func.



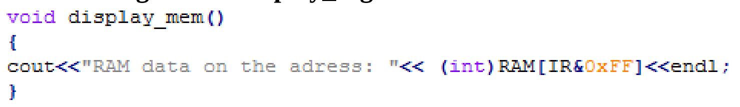
```
void load_program(int n)
{
    switch(n)
    {
        case 1:
        {
            RAM[0]=enter_instruction(LOAD_N1,8);
            RAM[1]=enter_instruction(LOAD_N0,7);
            RAM[2]=enter_instruction(STORE,22);
            RAM[3]=enter_instruction(LOAD_N0,0);
            RAM[4]=enter_instruction(LOAD_N1,32);
            RAM[5]=enter_instruction(STORE,23);
            RAM[6]=enter_instruction(LOAD_N1,20);
            RAM[7]=enter_instruction(STORE,24);
            RAM[8]=enter_instruction(LOAD_A,22);
            RAM[9]=enter_instruction(ADD_A,23);
            RAM[10]=enter_instruction(SUB_A,24);
            RAM[11]=enter_instruction(STORE,16);
            RAM[12]=enter_instruction(JUMP,16);
            RAM[13]=enter_instruction(DN,0);
            RAM[14]=enter_instruction(DN,0);
            RAM[15]=enter_instruction(DN,0);
            RAM[16]=enter_instruction(RESET,0);
            RAM[17]=enter_instruction(RESET,0);
            RAM[18]=enter_instruction(RESET,0);
            RAM[19]=enter_instruction(RESET,0);
            RAM[20]=enter_instruction(DN,0);
            RAM[21]=enter_instruction(PROCESS,0);
            max_program_address=21;
            min_data_adress=22;
            max_data_address=24;
            initialise_unused_ram();
            break;
        }
        case 2:
```

Figure 21: Load_program function (part of it).



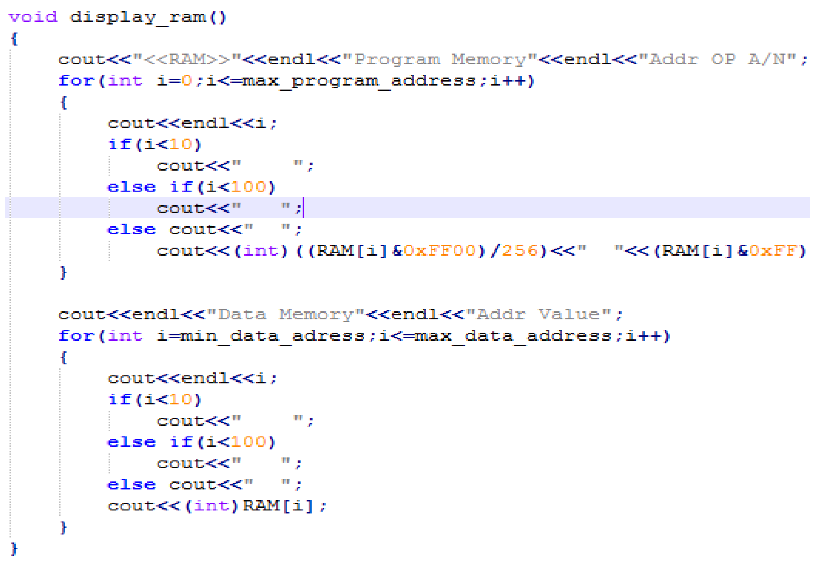
```
void display_reg()
{
    cout << "<<<CPU Registers>>>" << endl
    << "PC: " << (int)PC << endl
    << "IR->" <<"OP: "<< (int)((IR&0xFF00)/256) <<" A/N: " << (int)(IR&0xFF) << endl
    << "ACC: " << (int)ACC << endl
    << "Temp: " << temp << endl
    << "Processing: " << processing << endl
    << "Overflow: " << overflow << endl
    << "Underflow: " << underflow << endl;
}
```

Figure 22: Display_reg function.



```
void display_mem()
{
cout<<"RAM data on the adress: "<< (int)RAM[IR&0xFF]<<endl;
}
```

Figure 23: Display_mem function.

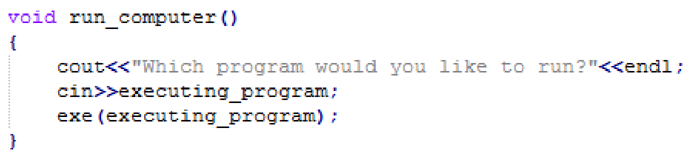


```
void display_ram()
{
    cout<<"<<RAM>>"<<endl<<"Program Memory"<<endl<<"Addr OP A/N";
    for(int i=0;i<=max_program_address;i++)
    {
        cout<<endl<<i;
        if(i<10)
            cout<<"    ";
        else if(i<100)
            cout<<"   ";
        else cout<<"  ";
            cout<<(int)((RAM[i]&0xFF00)/256)<<"  "<<(RAM[i]&0xFF)
    }

    cout<<endl<<"Data Memory"<<endl<<"Addr Value";
    for(int i=min_data_adress;i<=max_data_address;i++)
    {
        cout<<endl<<i;
        if(i<10)
            cout<<"    ";
        else if(i<100)
            cout<<"   ";
        else cout<<"  ";
        cout<<(int)RAM[i];
    }
}
```

Figure 24: Display_ram function.

The next support function is exe (), it turns on the CPU by reset_computer (), loads the program the user wants via load_program (), and goes into processing. The last support function is run_computer (), which does exactly what its name is, gets the program the user wants to run and executes, seen in (Figure 25).

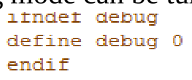
5. Validations, checks and debug mode.



```
void run_computer()
{
    cout<<"Which program would you like to run?"<<endl;
    cin>>executing_program;
    exe(executing_program);
}
```

Figure 25: Run_computer function

All of the points I will be talking about in this chapter can be seen in the figures above.
When adding or subtracting from an address, the first thing the execution flow goes through is a validation of the address (is the address in the instruction less or equal to the max_program_address), if not an error message pops and the processing stops. When adding, if the sum is larger than the max_data_cap, the ACC gets filled with the max_cap and overflow goes high. Subtracting checks are analogical to the adding ones, only the underflow goes high. On jumping, the function validates the address, which the program needs to jump to, if it is larger than the max_program_address, an error message pops, the registers are displayed so the user can see the fault in the IR and the computing is stopped. The next validation is used in enter_instruction (), if an address or a number is loaded in an instruction, they must fit in an unsigned byte, meaning 0-255, if the number/addr is not an error message pops and the processing stops. In load_program (), a check is implemented that pops an error message and does not let the processing to begin if the program number the user entered is wrong. Last, in exe (), after the execution stage of the cycle has finished I implemented a check for overflow, underflow or EOP, if one of these occurs a message is spit out and the CPU is shut down.
At the end of all execution flows, EOP, or any error (for every cycle the computer does, or doesn't do) if debug macro is defined the emulator will display the registers. Also, the debug mode displays memory on address when storing, loading, adding, subtracting from an address, nice messages for turning on of the computer, loading of the program, shutting down, as well as RAM at the end. The debug mode can be turned on with a compiler directive "-Ddebug" at the end of the compile command on the command prompt. If it is not defined on compile (Figure 26) ifdef sorts the debug mode to be off.

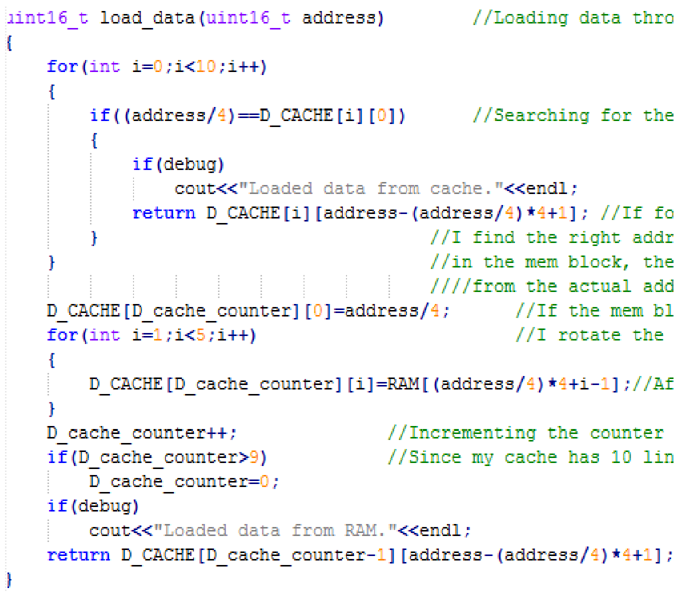


```
ifndef debug
define debug 0
endif
```

Figure 26: Debug mode off.

6. Extensions

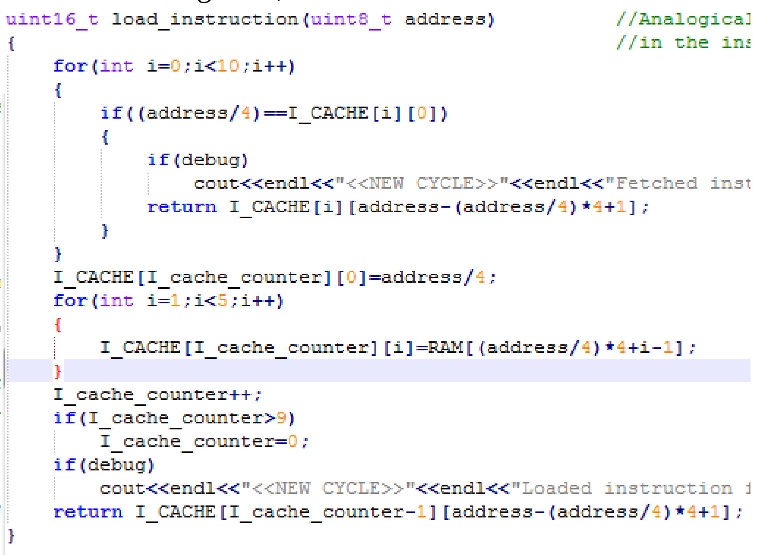
I have a lot of ideas how to make the computer better, as well as the emulator itself, in its user-friendliness and efficiency.
Two of the things I did further are: adding the executing_program value, which I needed to implement the loading of a particular program to happen from the prompt, while running the emulator, instead of entering programs by hand, or adding to code, the other thing is implementing cache memory.
To make the cache work I needed to change some of the functions. In the reset_computer (), I added initialising of the cache counters to zero. Another thing is, the initialise_unused_ram includes the initialising of the cache too. I designed two new functions load_data (Figure 27), and load_instruction (Figure 28). Load_data is used when loading any data from an address in the memory (LOAD_A, ADD_A, SUB_A) instead of just assigning value.



```
uint16_t load_data(uint16_t address)        //Loading data thro
{
    for(int i=0;i<10;i++)
    {
        if((address/4)==D_CACHE[i][0])      //Searching for the
        {
            if(debug)
                cout<<"Loaded data from cache."<<endl;
            return D_CACHE[i][address-(address/4)*4+1]; //If fo
        }                                             //I find the right addr
    }                                                 //in the mem block, the
                                                      ////from the actual add
    D_CACHE[D_cache_counter][0]=address/4;        //If the mem bl
    for(int i=1;i<5;i++)                          //I rotate the
    {
        D_CACHE[D_cache_counter][i]=RAM[(address/4)*4+i-1];//Af
    }
    D_cache_counter++;                   //Incrementing the counter
    if(D_cache_counter>9)                //Since my cache has 10 lin
        D_cache_counter=0;
    if(debug)
        cout<<"Loaded data from RAM."<<endl;
    return D_CACHE[D_cache_counter-1][address-(address/4)*4+1];
}
```

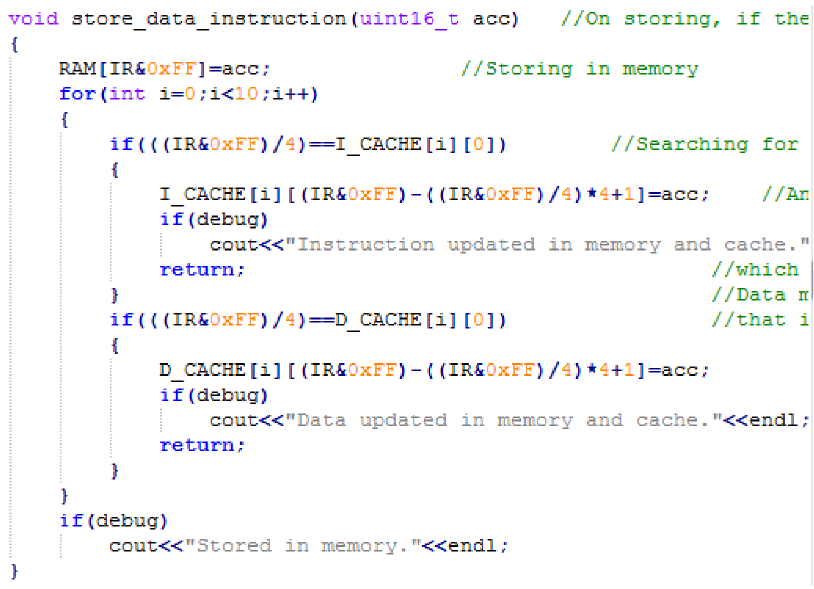
Figure 27: Load_data execution flow.

It takes the address and searches for its memory-block tag in the data cache. If it is there it returns the data on the address from the cache. If the memory block is not in the cache, it stores the memory-block tag of the address's block in the available cache's line, loads the block's addresses in the line, sorts out the next available cache line and returns the address's data. The Load_instruction function is analogical, but it is called when fetching new instruction and searches in the instruction cache. The last new function is store_data_instr (Figure 29). It is called when STORE operation is executed. It takes the ACC's value and stores it in RAM on the requested address, then it searches for the address's memory-block tag in both the data and instruction cache (most of the time data is stored, but the program can alter its own code, so the instruction cache needs to be checked too). If the address memory-block's tag is found the data/instruction is updated in the cache (this is very important because if the data is updated only in the RAM and it is saved on the cache, later when searching for it, the machine will load the outdated version. Of course, I added some nice display messages for the cache in debug mode.



```
uint16_t load_instruction(uint8_t address)                  //Analogical
{                                                            //in the ins
    for(int i=0;i<10;i++)
    {
        if((address/4)==I_CACHE[i][0])
        {
            if(debug)
                cout<<endl<<"<<NEW CYCLE>>"<<endl<<"Fetched inst
            return I_CACHE[i][address-(address/4)*4+1];
        }
    }
    I_CACHE[I_cache_counter][0]=address/4;
    for(int i=1;i<5;i++)
    {
        I_CACHE[I_cache_counter][i]=RAM[(address/4)*4+i-1];
    }
    I_cache_counter++;
    if(I_cache_counter>9)
        I_cache_counter=0;
    if(debug)
        cout<<endl<<"<<NEW CYCLE>>"<<endl<<"Loaded instruction i
    return I_CACHE[I_cache_counter-1][address-(address/4)*4+1];
}
```

Figure 28: Load_instruction execution flow

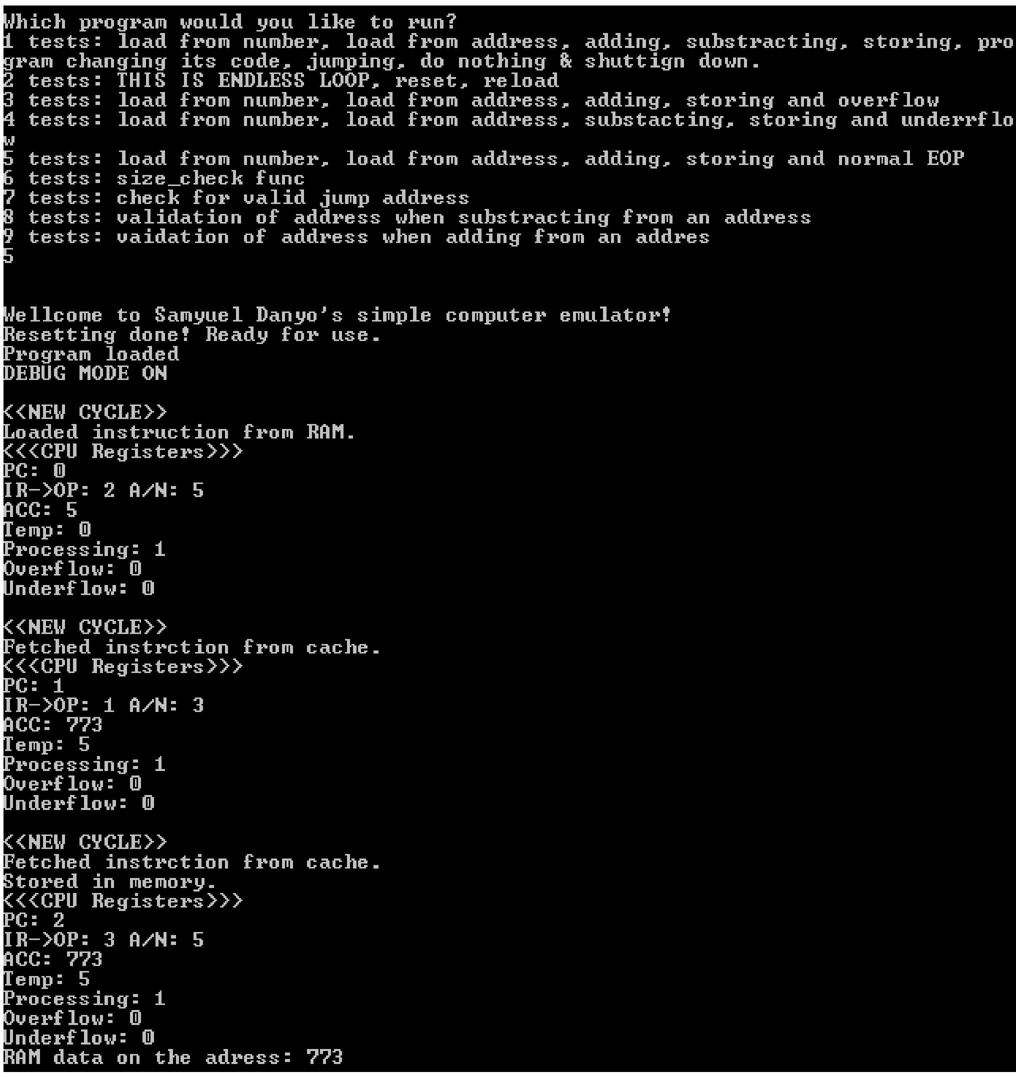


```
void store_data_instruction(uint16_t acc)   //On storing, if the
{
    RAM[IR&0xFF]=acc;                  //Storing in memory
    for(int i=0;i<10;i++)
    {
        if(((IR&0xFF)/4)==I_CACHE[i][0])        //Searching for
        {
            I_CACHE[i][(IR&0xFF)-((IR&0xFF)/4)*4+1]=acc;    //An
            if(debug)
                cout<<"Instruction updated in memory and cache."
            return;                                      //which
        }                                                //Data m
        if(((IR&0xFF)/4)==D_CACHE[i][0])                 //that i
        {
            D_CACHE[i][(IR&0xFF)-((IR&0xFF)/4)*4+1]=acc;
            if(debug)
                cout<<"Data updated in memory and cache."<<endl;
            return;
        }
    }
    if(debug)
        cout<<"Stored in memory."<<endl;
}
```

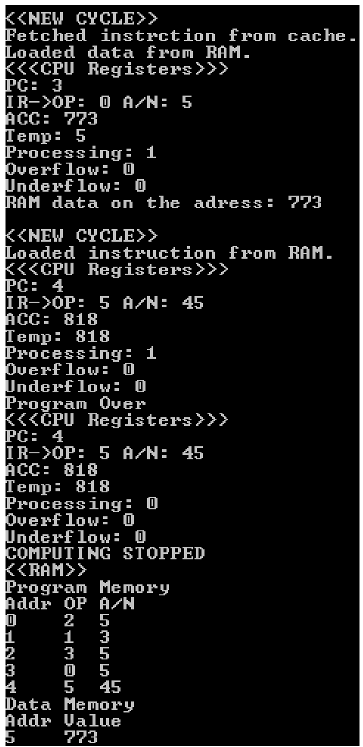
Figure 29: Store_data_instruction execution flow.

7. Testing.

I wrote 9 simple programs, which will be listed, when running the emulator. All aspects of the function of the computer are tested. When one runs the programs, use debug mode. I have made it very easy to follow each cycle and check the function of the CPU. Here are the results for one of the programs, where it can be seen the interface of the debug mode. →→→



```
Which program would you like to run?
1 tests: load from number, load from address, adding, substracting, storing, pro
gram changing its code, jumping, do nothing & shuttign down.
2 tests: THIS IS ENDLESS LOOP, reset, reload
3 tests: load from number, load from address, adding, storing and overflow
4 tests: load from number, load from address, substacting, storing and underrflo
w
5 tests: load from number, load from address, adding, storing and normal EOP
6 tests: size_check func
7 tests: check for valid jump address
8 tests: validation of address when substracting from an address
9 tests: vaidation of address when adding from an addres
5


Wellcome to Samyuel Danyo's simple computer emulator!
Resetting done! Ready for use.
Program loaded
DEBUG MODE ON

<<NEW CYCLE>>
Loaded instruction from RAM.
<<<CPU Registers>>>
PC: 0
IR->OP: 2 A/N: 5
ACC: 5
Temp: 0
Processing: 1
Overflow: 0
Underflow: 0

<<NEW CYCLE>>
Fetched instrction from cache.
<<<CPU Registers>>>
PC: 1
IR->OP: 1 A/N: 3
ACC: 773
Temp: 5
Processing: 1
Overflow: 0
Underflow: 0

<<NEW CYCLE>>
Fetched instrction from cache.
Stored in memory.
<<<CPU Registers>>>
PC: 2
IR->OP: 3 A/N: 5
ACC: 773
Temp: 5
Processing: 1
Overflow: 0
Underflow: 0
RAM data on the adress: 773
```



```
<<NEW CYCLE>>
Fetched instrction from cache.
Loaded data from RAM.
<<<CPU Registers>>>
PC: 3
IR->OP: 0 A/N: 5
ACC: 773
Temp: 5
Processing: 1
Overflow: 0
Underflow: 0
RAM data on the adress: 773

<<NEW CYCLE>>
Loaded instruction from RAM.
<<<CPU Registers>>>
PC: 4
IR->OP: 5 A/N: 45
ACC: 818
Temp: 818
Processing: 1
Overflow: 0
Underflow: 0
Program Over
<<<CPU Registers>>>
PC: 4
IR->OP: 5 A/N: 45
ACC: 818
Temp: 818
Processing: 0
Overflow: 0
Underflow: 0
COMPUTING STOPPED
<<RAM>>
Program Memory
Addr OP A/N
0     2  5
1     1  3
2     3  5
3     0  5
4     5  45
Data Memory
Addr Value
5     773
```