# ELEC2204 Coursework – Computer Emulation

Rishin Amin, Student #: 426725479

*ra8g14@soton.ac.uk*

*Personal Tutor: Dr Alex Weddell*

## 1 Introduction

This work outlines the process of designing, implementing and testing an emulator for a simplistic computer system with Von Neumann architecture. Test results are provided along with conclusions on the functionality of the emulator.

## 2 Emulated System Design

This section outlines the design of the emulated system, detailing its specification, instruction set and how it works.

### 2.1 Memory

The emulated system has 256 bytes of memory broken down into 128 cells; each cell consists of 2 bytes. This design was chosen to accommodate the way each instruction is broken down, see figure 1.

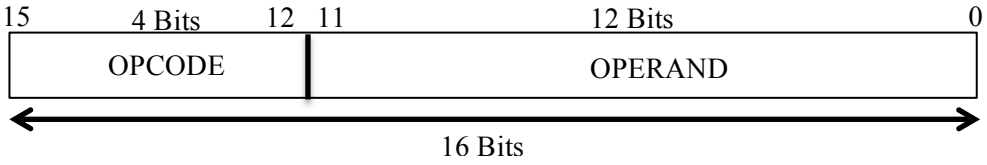


Figure 1 - Showing the way instructions are broken down in the emulated system

Each block of memory can store one instruction and as specified by the von Neumann architecture each memory location contains instructions and data. Resultantly, up to $2^{12} = 4096$ memory locations can be referenced giving the system much room for expansion. Only 128 were chosen in this case, as it is a more manageable number and the test programs do not require too much memory. This instruction design was chosen, as one byte per instruction would have been too small to allow enough memory to be referenced. The next logical instruction length is 2 bytes as it is standard practice to deal in bytes. The opcodes only required 4 bits due to the limited instruction set. Using two operands split into 6 bits each (one for data and one for referencing memory) seemed unnecessary due to the nature of the system's design; the system allows for user input data. The instruction topology dictated the size of the registers in CPU.

### 2.2 CPU

Figure 2 shows the high level schematic of the emulated CPU. The CPU consists of 5 registers: 'IR' (instruction register), 'ACC' (Accumulator), 'PC' (Program counter) 'INPUT' and 'OUTPUT'. IR stores the 'fetched' instruction from system memory. ACC performs basic arithmetic operations on data stored in it, such as add and subtract and stores the result. PC points to the next instruction in memory. The INPUT and OUTPUT registers are used for user input and program output respectively. This allows the basic programs to have some form of user interaction.

The CPU has 3 system flags: ‘overflow’, ‘underflow’ and ‘halt’. The overflow flag goes high if the contents of ACC are larger that 255 i.e. the register has does not contain enough bits to store the result. The underflow flag goes high if the contents of ACC are less than 0 i.e. the calculation in the register results in a negative number. The ‘Halt’ flag goes high when the program ends, telling the CPU to stop fetching and decoding instructions.

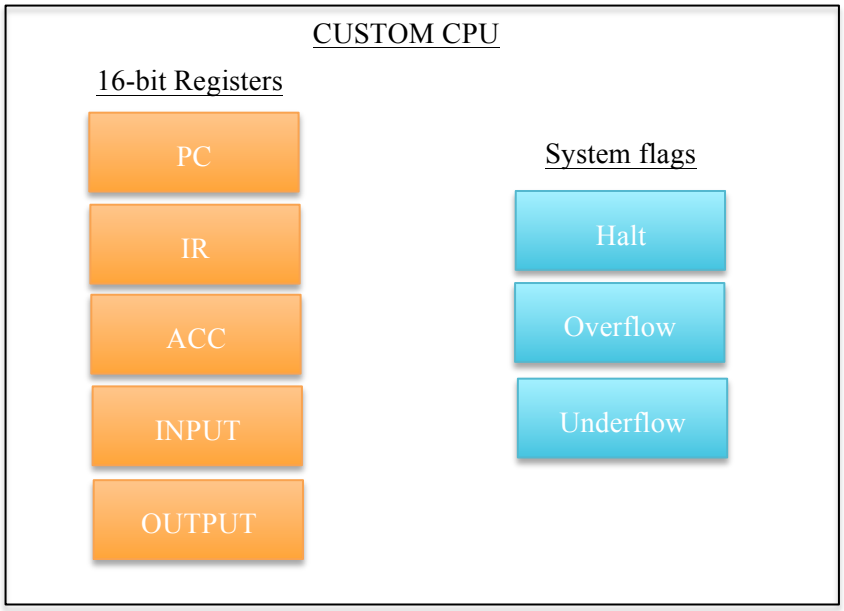


Figure 2 - Top-level view of the CPU schematic

### 2.3 Instruction set

A limited instruction set has been implemented for the CPU, the instructions, along with their descriptions are shown in the table below.

| Machine Code | Assembly Instruction | Description |
|---|---|---|
| 0x0111 | HLT | End the program by setting the ‘halt’ flag high |
| 0x1??? | ADD | Add contents of mem[operand] to contents of accumulator |
| 0x2??? | SUB | Sub contents of mem[operand] from contents of accumulator |
| 0x3??? | STA | Store contents of accumulator in mem[operand] |
| 0x4??? | LDA | Load contents of mem[operand] to the accumulator |
| 0x5??? | BRA | Set the program counter to operand |
| 0x6??? | BRZ | If contents of accumulator = 0 and overflow/ underflow flags are 0, set program counter to operand |
| 0x7??? | BRP | If the underflow flag is zero or the contents of the accumulator = 0, set the program counter to operand |
| 0x8??? | INP | Set the contents of the accumulator to the contents of the user input register |

| | | |
|---|---|---|
| 0x9??? | OUT | Output contents of the accumulator to the console |

Although limited these instructions can make up quite complex programs, allowing multiplication and division to be performed by repeatedly using ‘ADD’ and ‘SUB’ respectively. Each instruction is ‘#defined’ in the system’s class header therefore, programs can be written in assembly or machine code.

# 3 Emulator design

This section details the design of the emulator program and its capabilities.

## 3.1 Structure & Functionality

The emulator uses a class that encompasses the structure and running of the system. The class has four methods fundamental to the emulation of the system: fetch(), decode_execute(), load_program() and run().‘fetch()’ retrieves the instructions stored in the system memory and increments the program counter to point to the next instruction. ‘decode_execute()’ decodes the instruction in the instruction register, by isolating the opcode and the operand and executing the correct instruction using the correct memory location if necessary. ‘load_program()’ loads the program to main memory starting at the address ‘0x0000’. ‘run()’ runs the emulator until the halt flag is set high, by repeatedly calling ‘fetch()’ then ‘decode_execute()’.

## 3.2 Debug mode

A debug mode called ‘DEBUGGER’ has been implemented using pre compiler directives. If ‘DEBUGGER’ is defined in the emulator class the emulator will step through each instruction displaying contents of the registers involved in each instruction. Options to show the status of all the registers and flags or reveal the contents of all the memory locations are also provided after each instruction is executed. This debug made makes stepping through the program after making a test case much easier.

## 3.3 Operation

Operating the emulator program is very simply. A menu is displayed in the console, giving the users 3 programs to choose from ‘a’, ‘b’ or ‘c’. Descriptions for these programs can be accessed from the menu. When the user selects the run option, the emulator will load the selected program to memory then run the emulated system.

# 4 Testing

This section outlined the testing programmes ‘b’ and ‘c’ in the emulator using test cases and stepping through the program, one instruction at a time. The formal testing procedure of only these two programs are detailed as together they make use of the full instruction set.

## 4.1 Testing Program ‘b’

Program ‘b’ takes in a user input and outputs 1 if the number is even and 0 if the number is odd. Figure 3 shows the assembly code for this program when it is hardcoded into the memory.  When running in normal

```
mem[0] = INP;
mem[1] = STA | 16;
mem[2] = LDA | 16;
mem[3] = SUB | 15;
mem[4] = BRZ | 10;
mem[5] = STA | 16;
mem[6] = BRP | 2;
mem[7] = LDA | 13;
mem[8] = OUT;
mem[9] = BRA | 12;
mem[10] = LDA | 14;
mem[11] = OUT;
mem[12] = HLT;
mem[13] = 0;
mem[14] = 1;
```

Figure 3 - assembly code for program 'b'

mode the program gives the expected output as shown in figure 4.

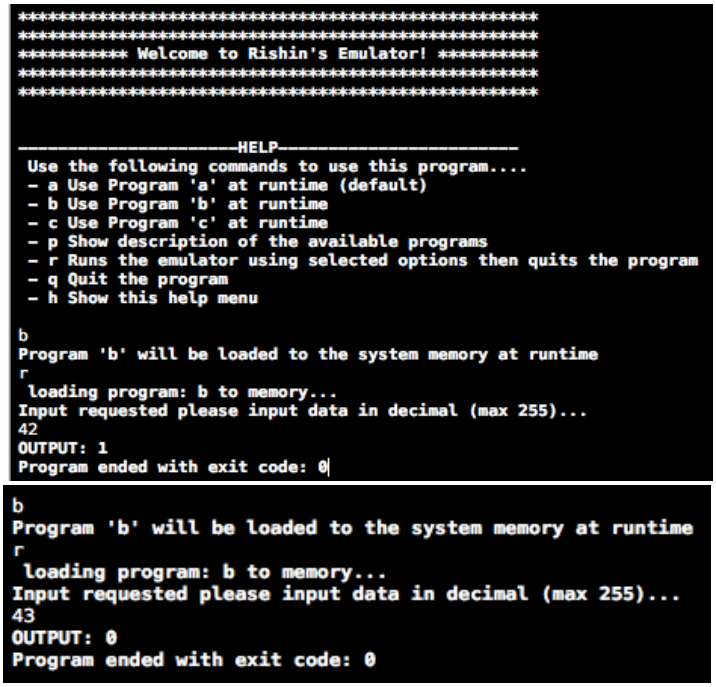


Figure 4 - output for program 'b' running in 'normal' mode

Better conclusions about the correct functionality of the program can be drawn when running the program in debug mode. The contents of all the memory locations can first be checked, to ensure the correct instructions and data are loaded. Figure 5 shows exactly this, the data loaded all seems to be as expected. Figure 5 reveals that code written in assembly is converted to machine code at compile time and stored as machine code in memory.

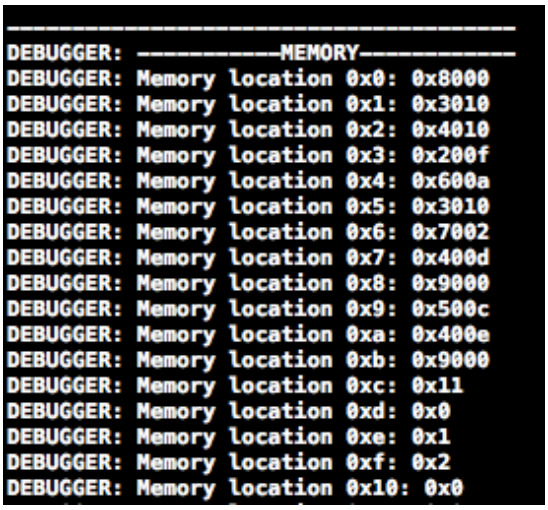


Figure 5 - terminal output revealing contents of system memory in debugger mode for program ‘b’

Stepping through the program in debugger mode gives the expected results at each stage. The console output is too long to include in this report, however figure 6 shows an example of the console output after each instruction is fetched, decoded and then executed.

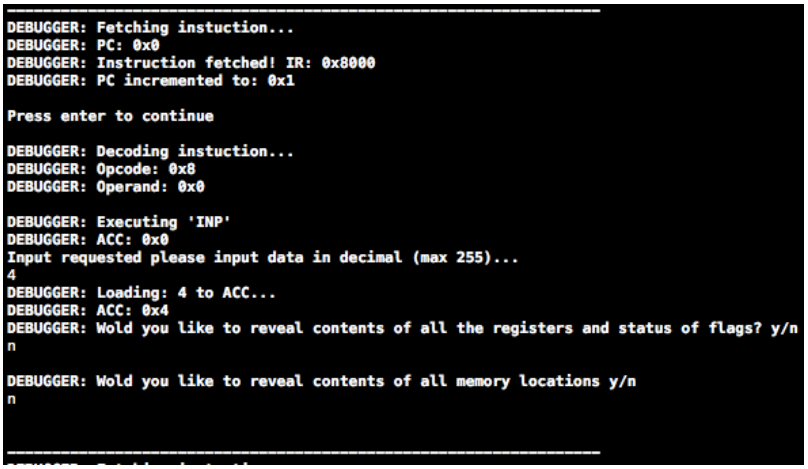


Figure 6 - example of debugger output to console for fetching, decoding and executing a single instruction

### 4.2 Testing program ‘c’

Program ‘c’ takes in two inputs, adds them together then outputs the result. Figure 6 shows the assembly code for this program.

```
mem[0] = INP;
mem[1] = STA | 6;
mem[2] = INP;
mem[3] = ADD | 6;
mem[4] = OUT;
mem[5] = HLT;
mem[6] = 0;
```

Figure 6 - Assembly code for program 'c'

When running in ‘normal’ mode the output is as expected, as shown in figure 7.

```
c
Program 'c' will be loaded to the system memory at runtime
r
 loading program: c to memory...
Input requested please input data in decimal (max 255)...
3
Input requested please input data in decimal (max 255)...
5
OUTPUT: 8
Program ended with exit code: 0
```

Figure 5 - terminal output for program 'c' running in 'normal' mode

The terminal output is shown below in bold for when the program is run in debugger mode:

**Running the program in 'DEBUGGER' mode**

**----------**
**DEBUGGER: Fetching instruction...**
**DEBUGGER: PC: 0x0**
**DEBUGGER: Instruction fetched! IR: 0x8000**
**DEBUGGER: PC incremented to: 0x1**

**Press enter to continue**

**DEBUGGER: Decoding instruction...**
**DEBUGGER: Opcode: 0x8**
**DEBUGGER: Operand: 0x0**

**DEBUGGER: Executing 'INP'**

**DEBUGGER: ACC: 0x0**
**Input requested please input data in decimal (max 255)...**
3
**DEBUGGER: Loading: 3 to ACC...**
**DEBUGGER: ACC: 0x3**
**DEBUGGER: Would you like to reveal contents of all the registers and status of flags? y/n**
n

**DEBUGGER: Would you like to reveal contents of all memory locations y/n**
n

**----------**
**DEBUGGER: Fetching instruction...**
**DEBUGGER: PC: 0x1**
**DEBUGGER: Instruction fetched! IR: 0x3006**
**DEBUGGER: PC incremented to: 0x2**

**Press enter to continue**

**DEBUGGER: Decoding instruction...**
**DEBUGGER: Opcode: 0x3**
**DEBUGGER: Operand: 0x6**

**DEBUGGER: Executing 'STA'**
**DEBUGGER: Storing: 0x3 to memory location: 0x6**
**DEBUGGER: Would you like to reveal contents of all the registers and status of flags? y/n**
n

**DEBUGGER: Would you like to reveal contents of all memory locations y/n**
n

**----------**
**DEBUGGER: Fetching instruction...**
**DEBUGGER: PC: 0x2**
**DEBUGGER: Instruction fetched! IR: 0x8000**
**DEBUGGER: PC incremented to: 0x3**

**Press enter to continue**

**DEBUGGER: Decoding instruction...**
**DEBUGGER: Opcode: 0x8**
**DEBUGGER: Operand: 0x0**

**DEBUGGER: Executing 'INP'**
**DEBUGGER: ACC: 0x3**
**Input requested please input data in decimal (max 255)...**
5
**DEBUGGER: Loading: 5 to ACC...**
**DEBUGGER: ACC: 0x5**
**DEBUGGER: Would you like to reveal contents of all the registers and status of flags? y/n**
n

**DEBUGGER: Would you like to reveal contents of all memory locations y/n**
n

**----------**
**DEBUGGER: Fetching instruction...**
**DEBUGGER: PC: 0x3**
**DEBUGGER: Instruction fetched! IR: 0x1006**
**DEBUGGER: PC incremented to: 0x4**

**Press enter to continue**

**DEBUGGER: Decoding instruction...**
**DEBUGGER: Opcode: 0x1**
**DEBUGGER: Operand: 0x6**

**DEBUGGER: Executing 'ADD'**
**DEBUGGER: Adding 0x5 to: 0x3**
**DEBUGGER: ACC: 0x8**
**DEBUGGER: Would you like to reveal contents of all the registers and status of flags? y/n**
n

**DEBUGGER: Would you like to reveal contents of all memory locations y/n**
n

**----------**
**DEBUGGER: Fetching instruction...**
**DEBUGGER: PC: 0x4**
**DEBUGGER: Instruction fetched! IR: 0x9000**
**DEBUGGER: PC incremented to: 0x5**

**Press enter to continue**

**DEBUGGER: Decoding instruction...**
**DEBUGGER: Opcode: 0x9**
**DEBUGGER: Operand: 0x0**

**DEBUGGER: Executing 'OUT'**
**DEBUGGER: ACC: 0x8**
**OUTPUT: 8**
**DEBUGGER: Loading ACC to OUTPUT...**
**DEBUGGER: OUT: 0x8**
**DEBUGGER: Would you like to reveal contents of all the registers and status of flags? y/n**
n

**DEBUGGER: Would you like to reveal contents of all memory locations y/n**
n

**----------**
**DEBUGGER: Fetching instruction...**
**DEBUGGER: PC: 0x5**
**DEBUGGER: Instruction fetched! IR: 0x11**
**DEBUGGER: PC incremented to: 0x6**

**Press enter to continue**

**DEBUGGER: Decoding instruction...**
**DEBUGGER: Opcode: 0x0**

**DEBUGGER: Operand: 0x11**

**DEBUGGER: Executing 'HLT' - Program end**
**DEBUGGER: halt flag set high**

**DEBUGGER: Would you like to reveal contents of all the registers and status of flags? y/n**
n

**Program ended with exit code: 0**

Working through this terminal output it is clear that the program and emulator behave as expected at each stage.

## 5 Conclusions

A simple computer architecture was designed and then successfully emulated. The testing procedure was thorough and revealed correct performance thanks to the debugger mode. A greater understanding of how a computer works has been gained as a result of this task.