

EDE: ELB816 Development Environment

James Bowden (110104485)

March 10, 2014

Abstract

The ELB816 Development Environment consists of an assembler, emulator and debugger for the ELB816 microprocessor system. This report details the design and usage of each of its elements.

Contents

I	Introduction and Specification	3
1	Motivations	3
2	Project Aims	3
3	Methodology	3
3.1	Assembler	3
3.2	Emulator	4
3.3	Debugger	4
II	Assembler	5
4	Data Structures	6
5	Functions	7
5.1	first_pass	8
5.2	second_pass	9
5.3	tokenize	10
5.4	stoi	11
6	Assembly language manual	12
III	Emulator	13
7	Core microprocessor emulation	13
7.1	iset.c and iset.h	13
7.2	mem.c and mem.h	14
7.3	emu.c	15
8	Peripherals	16

Part I

Introduction and Specification

1 Motivations

The ELB816 architecture is designed to be a “simple to understand 8-bit microprocessor system to help learn about microprocessor electronics.”

The combination of an ELB816 emulator, debugger and assembler could be used as a set of tools for learning or teaching microprocessor programming without the intricacies of real-world commercial microprocessors getting in the way of a fundamental understanding of the subject.

A PC based emulator would allow students to quickly develop and debug programs written in a simple assembly language on any modern desktop or laptop and an MCS-51 port running on an 8052 would allow students to test programs in an actual circuit.

2 Project Aims

- Develop an assembler for the ELB816 assembly language.
- Develop an emulated programmable microprocessor system based on the ELB816 architecture.
- Develop a debugger that allows interactive debugging of programs running on the emulator.

3 Methodology

3.1 Assembler

Language: Python

Priority: First

The assembler will be developed before anything else so that it can subsequently be used to assemble test programs during development of the emulator.

3.2 Emulator

Language: C

Priority: Second

The emulator will use only standard libraries in order to ensure it is portable between compilers and platforms. Specifically GCC for x86 and Keil C51 for Intel MCS-51. The emulator will first be developed on Linux to facilitated rapid development. It will be ported to MCS-51 once it is complete

3.3 Debugger

Language: C/Python

Priority: Second

The debug interface will be developed along side the emulator. It will consist of a simple text based interface built into the emulator that will read commands using C's `stdio.h` library. This means that on Linux the commands will be issued using `STDIN` and on the MCS-51 version they will be issued over a serial interface. Python will be used to provide a cleaner interface for common debug procedures such as writing programs to memory and setting break-points.

The remainder of this report is split into three parts, one for each component of the project, and will attempt to demonstrate the design and usage of each of these components.

Part II

Assembler

The assembler is written in pure Python 2 using only the standard library. It assembles the assembly language described in the ELB816 specification with a few minor differences. These differences are:

- In-line arithmetic must be wrapped in curved brackets eg. start with '(' and end with ')'. This is a limitation of the design of the program and to change it would require a large amount of code to be re-written.
- The only directives that have been implemented are ORG, EQU, DB and DS. The other directives listed in the specification have not been implemented, but their omission is only due to time constraints and they could easily be implemented in a later version.
- Macros have not been implemented also due to time constraints.

The assembler consists of two files:

- `language.py` which contains the language definition in an index and some functions to help encode instructions.
- `assembler.py` which contains the first and second pass functions and handles opening source files and writing binary files.

The following sections detail the design and behavior of the assembler. However it must be noted that these are abstract and high level descriptions that do not fully explain minor routines, but give an overview of the entire process. The full source code is attached in the Appendix and should be referenced for a deeper understanding of the program's operation. The final section is a short programmers manual demonstrating the assembler's features.

4 Data Structures

- reserved arguments

This structure contains a list of string representations of the reserved word arguments for the instruction set. These all equate to registers or register pointers. The full list is as follows:

```
a, c, bs, ie, flags,
r0, r1, r2, r3,
dptr, dpl, dph,
sp, sph, spl,
@a+pc, @a+dptr, @dptr
```

- relative instructions

This structure contains a list of string representations of the mnemonics of instructions that use relative addressing. The full list is as follows:

```
djnz, cjne, sjmp, jz,
jnz, jc, jnc, jpo,
jpe, js, jns
```

- instruction index

This structure contains an index of all possible instructions in the instruction set, along with the the corresponding opcode and instruction width. This is implemented using a combination of Python's dictionary, tuple and list objects. Its structure is demonstrated below:

```
mnemonic: (arg type, arg type, ...): [opcode, width]
```

Each mnemonic has an entry in the parent index which returns another index of possible argument formats for that mnemonic with their corresponding opcode and length. Argument types can be either be one of the reserved arguments or one of the following values: address, pointer, data or label . Width is represented in number of bytes, ie. width = 3 means 1 byte of opcode and 2 bytes of arguments.

- label index

This structure is used to store an index of label definitions.

- equate index

This structure is used to store an index of equated strings.

5 Functions

- `first_pass(source file)`

This function pre-processes a source file and stores it in a format containing the necessary data for the `second_pass()` function to assemble it. It processes labels and EQU directives by storing strings and their corresponding values in indexes and replacing any subsequent appearances of the string with the value. It prepares ORG and DB statements for the `second_pass()`. It uses the `tokenize()` function to determine the argument symbols and operand bit string. Finally it uses the `instruction index` to determine the instruction width.

- `second_pass(asm, label index)`

This function takes the pre-processed assembly code and `label index` output by `first_pass()` as input. First it checks for ORG and DB statements and handles them if necessary. Then it replaces any labels that were used before they were defined and therefore not replaced on by `first_pass()`. It uses the `instruction index` to determine the opcode and the width of the instruction, then it writes the opcode and operand to the file. If the combined width of the opcode and operand is greater than the instruction width the function raises an error.

- `tokenize(mnemonic, arguments)`

This function processes an instruction in order to produce a hashable symbol that represents the format of its arguments. This symbol is used to look up opcodes in the `instruction index`. It also detects string representations of numbers in the arguments and stores a C type struct representation of the operands to be returned along with the symbol. It does this with the help of the `stoi()` function and Python's `struct` module.

- `stoi(string)`

This function is a general purpose function that is actually used throughout the code, although mainly in the `tokenize()` function. It takes a string as an input and tries to convert it to an integer using Python's integer representation syntax. It can recognize decimal, octal, hexadecimal and binary numbers which are denoted with different prefixes. If it receives a string it can not represent as an integer it returns the string 'NaN', (Not a Number)

Below is an abstract representation of each functions logical process. The `first_pass()` and `second_pass()` are represented in pseudo-code, however `stoi()` and `tokenize()` are more easily understood when represented as flowcharts.

5.1 first_pass

```
first_pass(source file):

    address = 0

    for statement in source file:

        remove comments

        for word in statement:

            if word is in equate index:
                replace word with equated value
            else if word is in label index:
                replace word with address at label

            if first word == 'org'
                address = second word
            else if last character of first word == ':':
                remove ':'
                add word = address to label index
            next statement
            else if second word == 'equ'
                add first word = third word to equate index
            next statement

        mnemonic = first word
        arguments = [second word ... last word]

        symbol, constant = tokenize(arguments)
        if mnemonic == 'db':
            address = address + width of constant
        next statement

        width = instruction index[mnemonic][symbol][width]
        address = address + width

        append [mnemonic, argument, symbol, constant] to asm

    return asm, label index
```

5.2 second_pass

```
second_pass(file, asm, label_index):

    address = 0

    for line in asm:

        file_offset = address

        mnemonic, arguments, symbol, constant = line

        if mnemonic == 'org':
            address = first_argument
            next_line
        else if mnemonic == 'db':
            write_constant_to(file)
            address = address + width_of_constant
            next_line

        for argument in arguments:
            if argument is a_label:
                replace_argument_with_address_at_label(
                    symbol, data = tokenize(argument))
                append_data_to_constant

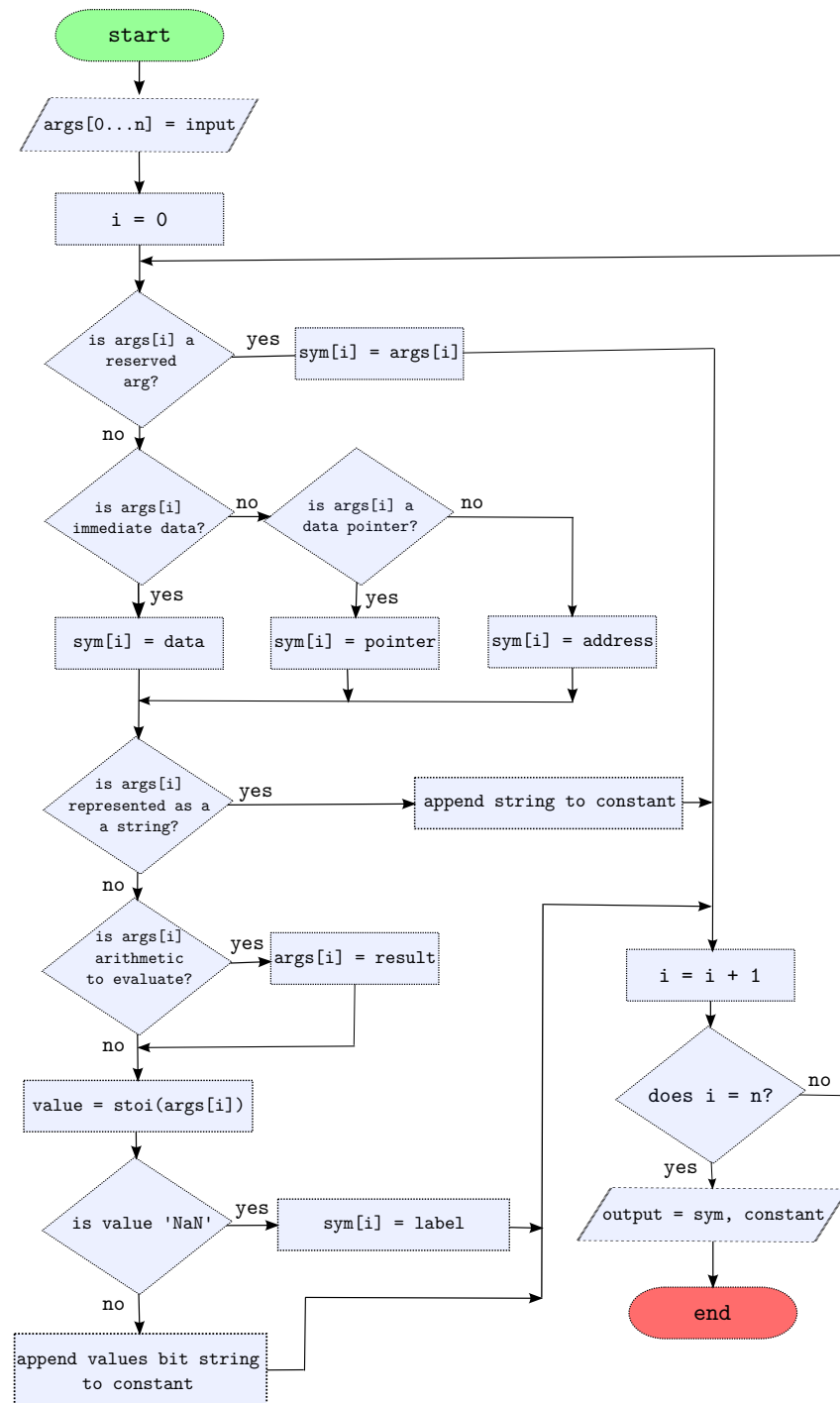
        op, width = instruction_index[mnemonic][symbol]

        write_op_to(file)

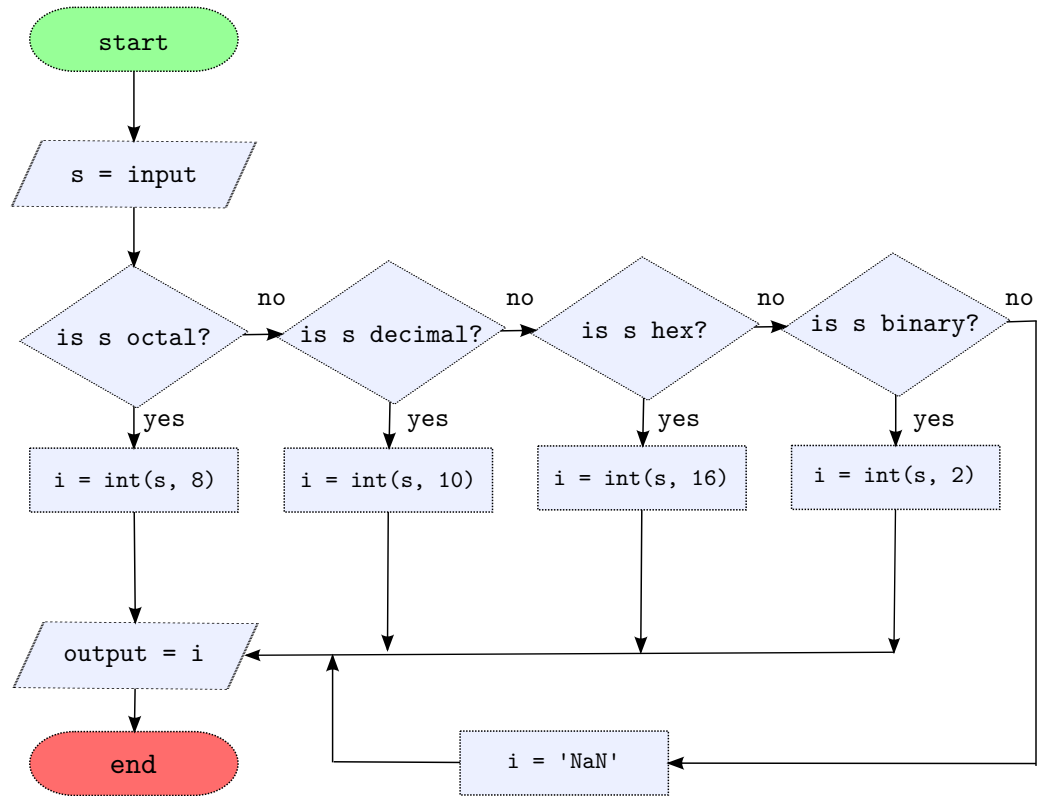
        if width_of_constant - width + 1 > 0:
            raise_error
        else if:
            write_constant_to(file)
            address = address + width

    return file
```

5.3 tokenize



5.4 stoi



6 Assembly language manual

Part III

Emulator

7 Core microprocessor emulation

The core of the emulator is written in C using only standard libraries. It executes the machine code output by the assembler according to the ELB816 specification. It consists of the following files:

- `iset.c` and `iset.h`

These files contain the emulator instruction functions and function look-up table.

- `mem.c` and `mem.h`

These files contain the emulators memory structure and memory access functions.

- `emu.c`

This file contains the program's `main()` function. It initializes the emulator and executes the programs fetch/decode/execute cycle.

Below is a high level description of the content of each of these files which should demonstrate how the emulator works. There is also a large amount of material relevant to the emulator's design in the appendix, which will be referenced when applicable.

7.1 `iset.c` and `iset.h`

Each mnemonic in the ELB816 instruction set has a function defined in these files. Each function is responsible for execution of all the instructions that use its corresponding mnemonic. The function look-up table is an array of pointers to these functions, where a pointer's position in the list corresponds to the opcode of the instruction to be executed.

7.2 `mem.c` and `mem.h`

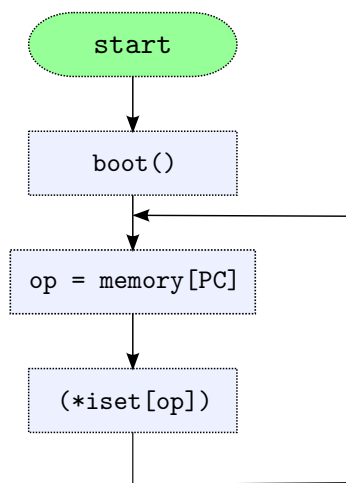
The figures bellow illustrate the emulator's memory layout as defined in the `mem.h` header file.

`mem.c` contains functions that can be used to access this memory from the rest of the code.

7.3 `emu.c`

This file contains the emulator's set-up and control procedures. It includes all of the projects header files and controls the execution of the functions contained in them.

It first executes a number of initialization procedures and then passes control over to the main fetch/decode/execute cycle. This procedure is shown below as a flowchart. To understand this it you must be familiar with C's function pointer syntax.



8 Peripherals