
Python

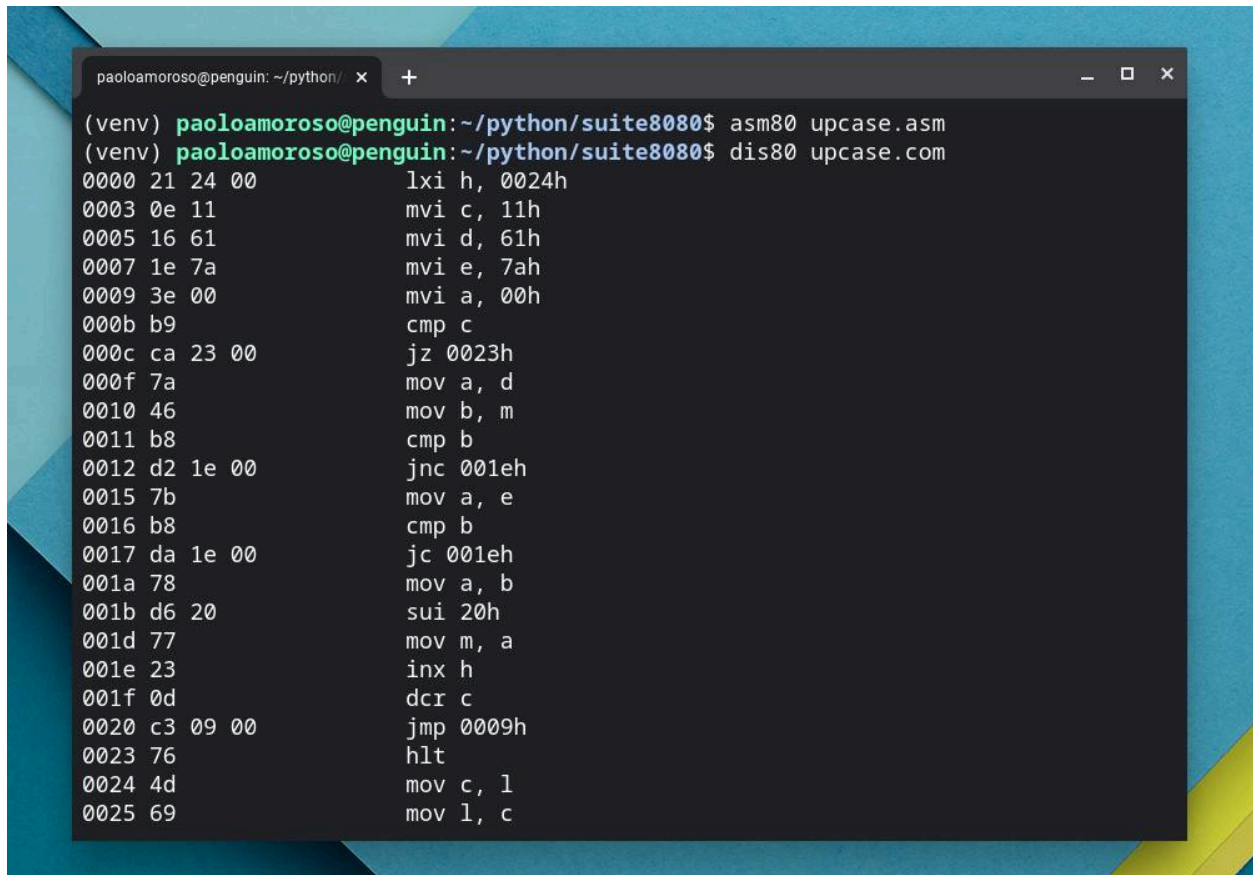
Paolo Amoroso

Nov 26, 2023

CONTENTS

1	Getting started	3
1.1	Installation	3
1.2	Usage examples	3
2	Using the Suite8080 tools	5
2.1	Assembler	5
2.2	Disassembler	8
3	Design notes	9
3.1	Source code organization	9
3.2	Assembler	9
3.3	Future work	10
4	Reference	11
4.1	Assembler	11
4.2	Disassembler	13
	Python Module Index	15
	Index	17

Suite8080 is a suite of Intel 8080 Assembly cross-development tools written in Python.



```
paoloamoroso@penguin: ~/python/
(venv) paoloamoroso@penguin:~/python/suite8080$ asm80 upcase.asm
(venv) paoloamoroso@penguin:~/python/suite8080$ dis80 upcase.com
0000 21 24 00      lxi h, 0024h
0003 0e 11        mvi c, 11h
0005 16 61        mvi d, 61h
0007 1e 7a        mvi e, 7ah
0009 3e 00        mvi a, 00h
000b b9         cmp c
000c ca 23 00    jz 0023h
000f 7a         mov a, d
0010 46         mov b, m
0011 b8         cmp b
0012 d2 1e 00    jnc 001eh
0015 7b         mov a, e
0016 b8         cmp b
0017 da 1e 00    jc 001eh
001a 78         mov a, b
001b d6 20      sui 20h
001d 77         mov m, a
001e 23         inx h
001f 0d         dcr c
0020 c3 09 00    jmp 0009h
0023 76         hlt
0024 4d         mov c, l
0025 69         mov l, c
```

The suite comprises the following command-line programs and more will come:

- `asm80`: assembler
- `dis80`: disassembler

This project is inspired by [a series of blog posts](#) by Brian Robert Callahan on demystifying programs that create programs. In an ongoing series of posts on my own blog I'm telling about [my work on and experience with developing Suite8080](#).

The executable files generated and processed by the tools are supposed to run on any Intel 8080 system such as CP/M computers, both actual devices and emulated ones.

Suite8080, which is developed with [Replit](#), requires Python 3.6 or later and depends on Pytest for unit tests.

Documentation and sample programs

For information on how to use Suite8080 and its design see the [documentation](#).

The `asm` directory of the source tree contains sample Assembly programs, some of which run on CP/M and others on a bare Intel 8080 system with no host environment. You can use the Suite8080 tools to process these programs, for example assemble the sources with `asm80` and disassemble the executables with `dis80`.

Status

Suite8080 is in early development and some of the planned tools and features are not available yet.

Release history

See the [list of releases](#) for notes on the changes in each version.

Author

[Paolo Amoroso](#) developed Suite8080. Email: info@paoloamoroso.com

License

This code is distributed under the MIT license, see the `LICENSE` file.

GETTING STARTED

The Suite8080 tools are command line programs. Follow these instructions for installing the tools and giving them a try.

1.1 Installation

Install Suite8080 from PyPI with the command:

```
$ pip install suite8080
```

1.2 Usage examples

1.2.1 Linux

To run the assembler on Linux execute:

```
$ asm80 file.asm
```

where `file.asm` is an Intel 8080 Assembly source file. You can disassemble the resulting program with:

```
$ dis80 file.com
```

where `file.com` is an executable Intel 8080 program.

1.2.2 Replit

To run the programs online on Replit visit the [Suite8080 REPL](#) with a browser. You first have to set up the environment by forking the REPL, opening the Shell pane, and editing `~/.bashrc` to add `export PYTHONPATH=.:$PYTHONPATH`. Next, click Run. Finally, change to the `suite8080/suite8080` directory of the source tree.

To run the assembler execute:

```
$ python3 -m asm80 file.asm
```

where `file.asm` is an Intel 8080 Assembly source file.

You can disassemble a program with the command:

```
$ python3 -m dis80 file.com
```

where `file.com` is an executable Intel 8080 program.

USING THE SUITE8080 TOOLS

This document describes the usage and features of the programs comprising [Suite8080](#), a suite of Intel 8080 cross-development tools. See the file [README.md](#) in the source tree for an overview of Suite8080 and installation instructions.

2.1 Assembler

The `asm80` cross-assembler takes an Intel 8080 Assembly source file as input and generates an executable program in `.com` format. The assembler supports the full Intel 8080 instruction set but not the additional Intel 8085 or Z80 instructions.

If the assembler detects a syntax error, it prints an error message and exits.

2.1.1 Usage

The `asm80` command line program has the following syntax:

```
asm80 [-h] [-o OUTFILE] [-v] filename
```

All arguments are optional except for the input file `filename`, which may be `-` to read from standard input:

- `-h, --help`: prints a help message and exits
- `-o, --outfile`: output file name, which defaults to `program.com` if the input file is `-` and `-o` is not supplied
- `-s, --symtab`: saves the symbol table to a file with the name of the input file and the `.sym` extension; the argument of `-o` and the `.sym` extension; or `program.sym` if the input file is `-` and `-o` is not supplied
- `-v, --verbose`: increases output verbosity

Although no input file name extension is enforced, and any is accepted or may be skipped altogether, I recommend `.asm` or `.a80` for Assembly source files and `.m4` for m4 macro files.

The symbol table is saved in the `.sym` CP/M file format described in section 1.1 “SID Startup” on page 4 of the [SID Users Guide](#) manual published by Digital Research.

2.1.2 Assembly syntax

Except for macros, `asm80` recognizes most of the Assembly language of early Intel 8080 assemblers such as the ones by Intel, Digital Research, and Microsoft. However, source files written for those tools may need minor adaptations to work with `asm80`.

An Assembly source line has the syntax:

```
[label:] [mnemonic [operand1[, operand2]]] [; comment]
```

Although the 8080 mnemonics and directives accept from zero to two arguments, the `db` directive can take multiple arguments that may be numbers, strings, characters constants, and labels:

```
[label:] [db [argument1[, ..., argumentN]]] [; comment]
```

Two-letter abbreviations of register pairs are valid along with single-letter ones. In other words, the assembler, for example, accepts both `d` and `de` as the name of the register pair consisting of the `d` and `e` registers.

Character constants such as `'C'` or `'*'` can be immediate operands of Assembly instructions, as well as arguments of the `db` and `equ` directives.

2.1.3 Numbers

Only non-negative integers are accepted.

Numbers may be decimal, hexadecimal, or octal. Hexadecimal numbers must end with `h` (for example `1dh`), octal ones with `q` (e.g. `31q`). Hexadecimal numbers beginning with the digits `a` to `f` must be prefixed with `0`, such as `0bh`.

2.1.4 Expressions

The arguments of the `equ` directive support expressions of the form:

```
$OPnumber
```

where `$` is the current address, `OP` an operator, `number` a number, and no spaces are allowed at either side of the operator. Valid operands are `+`, `-`, `*`, `/`, and `%` (modulus).

No other expressions are supported.

2.1.5 Strings and character constants

Strings are sequences of characters enclosed within single `'` or double `"` quotes, such as `'This is a string'`. Strings delimited by single quotes may contain double quotes, and vice versa, as in `"I'm a string"` or `'This is a "string"'`.

Character constants, also known as ASCII constants, are strings containing only one character. For example, `'F'`, `"b"`, and `'*'` are character constants.

2.1.6 Macros

Reading from standard input by supplying `-` as the input file makes it possible to use the Unix program `m4` as an Assembly macro processor, as demonstrated by the sample files with the `.m4` extension in the `asm` directory of the source tree. However, `m4` macros are not compatible with the ones of traditional Intel 8080 macro assemblers.

For example, to assemble the `filename.m4` source file containing `m4` macros, run a pipe such as this on Linux:

```
$ cat filename.m4 | m4 | asm80 - -o filename.com
```

To view the Assembly source with the macros expanded execute:

```
$ cat filename.m4 | m4 | more
```

2.1.7 Running Intel 8080 programs

The programs `asm80` assembles can run on actual Intel 8080 or Z80 machines, such as CP/M computers, or emulated ones. I use and recommend the following emulators:

- **z80pack**: the most versatile Z80 emulator with support for different machines and CP/M versions
- **ANSI CP/M Emulator and disk image tool**: it allows invoking from the Linux shell the emulator and passing as an argument a CP/M program to run, e.g. `cpm cpmprogram`
- **ASM80**: works fully in the cloud, can run code that doesn't require a host operating system environment, and supports inspecting registers, memory, and program state
- **Intel 8080 CPU Emulator (documentation)**: an online Intel 8080 and CP/M emulator

2.1.8 Limitations and issues

The assembler is in early development and, although it performs basic syntax checking, there's little or no input validation.

Identifiers

Identifiers such as labels and mnemonics can be all lowercase (e.g. `equ`) or all uppercase (e.g. `EQU`), but not in mixed case like `Equ`. The assembler may accept some mixed case elements, but it's safer to stick with all lowercase or all uppercase.

Strings

In addition, strings must not contain comma `,` characters. As a workaround, break the string into parts not containing commas and insert the comma code (2C hex) at the appropriate place. Here's an example of allocating the string `I, robot`:

```
robot: db 'I', 2ch, ' robot'
```

Numbers

The assembler accepts only non-negative integers. A workaround is to enter negative numbers as 2's complement unsigned integers, e.g. 255 or 0ffh instead of -1.

Directives

The labels used as operands of `org` or `ds` must be defined before use. No forward references are allowed.

2.2 Disassembler

The `dis80` disassembler takes an executable Intel 8080 program file as input and prints to the standard output the sequence of instructions in symbolic form, along with an hexadecimal dump of the opcodes and operands. It supports the full Intel 8080 instruction set but not the additional Intel 8085 or Z80 instructions.

2.2.1 Usage

The `dis80` command line program has the following syntax:

```
dis80 [-h] filename
```

where `filename` is a required Intel 8080 executable input file. The only command line option is `-h` or `--help`, which prints a help message and exits.

2.2.2 Limitations and issues

The disassembler doesn't distinguish between instructions and data bytes, which may result in spurious instructions interleaved between valid ones. In addition, if some data bytes encode a transfer of program control that results in a jump beyond the last valid address, the disassembly may end prematurely without notice.

DESIGN NOTES

Suite8080 is a suite of Intel 8080 Assembly cross-development tools. See the `README.md` file in the source tree for an overview of what the system does and how to use it.

The initial implementation of Suite8080 closely follows the design of the corresponding tools developed in D language by Brian Robert Callahan and published in a [blog post series](#).

Therefore, the features and limitations, the function and variable names, the data structures, and the source organization are similar to Brian's code except where Python features make the code more readable or idiomatic with little effort. I renamed some of the variables to make them less terse and more clear. In Suite8080 a few functions, unlike in Brian's code, return values to simplify testing.

As I gain confidence with the algorithms and the system, I will refactor to make the code more Pythonic and add new features.

3.1 Source code organization

The `suite/suite8080` directory in the source tree defines a package and contains the source files of the command-line programs in the suite, one Python module for each tool. For example, `dis80.py` holds the code of the disassembler.

Replit Python projects require a `main.py` file at the root of the source tree. The Suite8080 `main.py` file is currently empty but I may add some code to demo the tools.

3.2 Assembler

The `asm80` assembler examines one source line at a time and doesn't rely on recursive descent or traditional parsing algorithms. It doesn't have a specific lexical analysis subsystem either.

3.2.1 Parser

Function `parse()` implements the parser. It scans a source line from right to left, looking for the symbols that separate successive syntactic elements. When it finds a symbol, the parser splits the line at the symbol to break the line into the syntactic element at the right of the symbol, and the rest of the line to process at the left.

For example, consider the syntax of a source line:

```
[label:] [mnemonic [operand1[, operand2]]] [; comment]
```

If the parser finds a semicolon, it splits the line at `;` to break the comment text from the rest of the line to parse. Next, it looks for the comma `,` separating the operands of the assembly instruction and splits there, thus breaking the second operand from the rest of the line to parse. And so on.

At each step, the parser calls `str.rpartition()` to scan for a symbol. The variables that unpack the values `str.rpartition()` returns have names that start with the name of the syntactic element looked for and end in `_l` (the remainder of the line at the left of the separator symbol), `_sep` (the separator symbol), and `_r` (the part of the line at the right of the symbol, i.e. the syntactic element).

Suppose the parser recognizes a comment. To scan for the `operand2` syntactic element, i.e. the second operand of the instruction, the parser then executes the statement:

```
operand2_l, operand2_sep, operand2_r = comment_l.rpartition(',')
```

The following step will start with the parser calling the `str.rpartition()` method on the `operand2_l` string, the remaining part of the line.

Function `parse()` updates the parsing state and output via a number of global variables, some of which hold the syntactic elements the scanning steps break from the line and produce as output (`label`, `mnemonic`, `operand1`, `operand2`, and `comment`). The strings in the variables are stripped of leading and trailing whitespace but are otherwise raw.

Once parsing completes, for each Assembly instruction a function with the same name accesses the global variables to further process the syntactic elements (e.g. for converting the text of a numeric literal to its value) and generate the code. These functions may access other global parsing state, such as the current address (`address`), line number (`lineno`), or source code pass (`source_pass`).

Function `parse()` supplies the syntactic elements also as return values, but they are currently used only for unit testing.

There are two exceptions to the scanning and splitting steps described above. The first is the `db` directive, which is parsed in the separate function `parse_db()`. The second is a special case inside function `parse()` to handle the `equ` directive.

3.3 Future work

I'd like to add to Suite8080 an IDE with a GUI to provide a dashboard for running the various tools and viewing their output. The project's `main.py` file may hold the IDE's source or code to start the IDE.

REFERENCE

This chapter lists the functions of the Suite8080 tools. Although the functions don't make up an API, and they aren't supposed to be called from outside of Suite8080, it's useful to list the functions and give a bird's eye view of how the system is organized.

4.1 Assembler

An Intel 8080 cross-assembler.

`suite8080.asm80.add_label()`

Add a label to the symbol table.

`suite8080.asm80.address16()`

Generate code for 16-bit addresses.

`suite8080.asm80.assemble(lines)`

Assemble source lines.

`suite8080.asm80.check_operands(valid)`

Report error if argument isn't Truthy.

`suite8080.asm80.dollar(current_address, expression)`

Calculate value of \$-address expression.

`suite8080.asm80.get_number(input)`

Return value of hex or decimal numeric input string.

`suite8080.asm80.immediate_operand(operand_type=8)`

Generate code for an 8-bit or 16-bit immediate operand.

`suite8080.asm80.is_char_constant(string)`

Return True if string is a character constant.

A character constant is a quote-delimited string containing only one character such as 'Z' or '*'.

`suite8080.asm80.is_quote_delimited(string)`

Return True if string is enclosed between single or double quotes.

`suite8080.asm80.main()`

Parse the command line and pass the input file to the assembler.

`suite8080.asm80.parse(line)`

Parse a source line.

`suite8080.asm80.parse_db(line)`

Parse db directive.

Parse the source line to check whether it's a valid db directive. If it is return 'db' as the second value and the arguments as the third. The first value is the label if present, otherwise a null string.

Assume the source line doesn't contain a comment.

Parameters `line` (*string*) – Source line

Returns A tuple (`label`, `directive`, `arguments`) where `label` is a lowercase label if present (otherwise ''), `directive` is 'db' if the line contains a valid db directive (otherwise ''), and `arguments` is a string of arguments if the line contains a db directive (otherwise '').

Return type tuple

`suite8080.asm80.parse_db_arguments(string)`

Return a list of db arguments parsed from string.

Split string into arguments, strip whitespace from them, and return a list of the resulting arguments.

`suite8080.asm80.pass_action(instruction_size, output_byte, should_add_label=True)`

Build symbol table in pass 1, generate code in pass 2.

Parameters

- **instruction_size** (*int*) – Number of bytes of the instruction
- **output_byte** (*bytes*) – Opcode, b'' if no output should be generated.
- **should_add_label** (*bool*) – True if the label, when present, should be added

`suite8080.asm80.process_instruction()`

Check instruction operands and generate code.

`suite8080.asm80.register_offset16()`

Return encoding of 16-bit register pair.

`suite8080.asm80.register_offset8(raw_register)`

Return encoding of 8-bit register.

`suite8080.asm80.report_error(message)`

Display an error message and exit returning an error code.

`suite8080.asm80.write_binary_file(filename, binary_data)`

Write `binary_data` to `filename` and return number of bytes written.

`suite8080.asm80.write_symbol_table(table, filename)`

Save symbol table to `filename` and return the number of symbols written.

The table is written to a text file in the CP/M .sym file format. No file is created if the table is empty.

4.2 Disassembler

An Intel 8080 disassembler.

PYTHON MODULE INDEX

S

`suite8080.asm`[80](#), [11](#)

`suite8080.dis`[80](#), [13](#)

INDEX

A

`add_label()` (in module *suite8080.asm80*), 11
`address16()` (in module *suite8080.asm80*), 11
`assemble()` (in module *suite8080.asm80*), 11

C

`check_operands()` (in module *suite8080.asm80*), 11

D

`dollar()` (in module *suite8080.asm80*), 11

G

`get_number()` (in module *suite8080.asm80*), 11

I

`immediate_operand()` (in module *suite8080.asm80*), 11
`is_char_constant()` (in module *suite8080.asm80*), 11
`is_quote_delimited()` (in module *suite8080.asm80*), 11

M

`main()` (in module *suite8080.asm80*), 11
`module`
 suite8080.asm80, 11
 suite8080.dis80, 13

P

`parse()` (in module *suite8080.asm80*), 11
`parse_db()` (in module *suite8080.asm80*), 11
`parse_db_arguments()` (in module *suite8080.asm80*), 12
`pass_action()` (in module *suite8080.asm80*), 12
`process_instruction()` (in module *suite8080.asm80*), 12

R

`register_offset16()` (in module *suite8080.asm80*), 12

`register_offset8()` (in module *suite8080.asm80*), 12
`report_error()` (in module *suite8080.asm80*), 12

S

suite8080.asm80
 module, 11
suite8080.dis80
 module, 13

W

`write_binary_file()` (in module *suite8080.asm80*), 12
`write_symbol_table()` (in module *suite8080.asm80*), 12