test Documentation

Release 1.0.1.dev0

foo

Dec 16, 2017

Contents

Author Mike Naberezny

Version 1.0.1.dev0

Overview

Simulate 6502-based microcomputer systems in Python.

CHAPTER 1

Using the Monitor

1.1 Introduction

Py65 includes a program called Py65Mon that functions as a machine language monitor. This kind of program is sometimes also called a debugger. Py65Mon provides a command line with many convenient commands for interacting with the simulated 6502-based system.

The monitor is started using the `py65mon` command:

```
$ py65mon

Py65 Monitor

       PC  AC XR YR SP NV-BDIZC
6502: 0000 00 00 00 ff 00110000
.
```

Once the monitor has started, it will display a register dump and the dot prompt. You can then enter commands for the monitor at this prompt.

Py65Mon uses commands that are very similar to those used by the monitor included with the VICE emulator for Commodore computers. You can get a list of available commands with `help` or help on a specific command with `help command`.

1.2 Number Systems

When working with Py65Mon, you will frequently need to enter numbers, addresses, and ranges of addresses. Almost all Py65 command support entering numbers in binary, decimal, and hexadecimal.

Numbers can be entered with a prefix to specify the radix, e.g. `$c000` instructs Py65Mon that the number `c000` is hexadecimal. The following prefixes are supported:

- `$c000`: The dollar sign indicates hexadecimal.

- `+828`: The plus sign indicates decimal.
- `%0101`: The percent sign indicates binary.

Numbers can also be entered without a prefix. Most of the time, working in hexadecimal will be the most convenient so this is the default radix. The number `c000` will be assumed to be hexadecimal unless the default radix is changed using the `radix` command.

1.3 Address Ranges

Some commands accept a range of memory addresses:

```
.disassemble ff80:ff84
$ff80  d8        CLD
$ff81  a2 ff     LDX #$ff
$ff83  9a        TXS
$ff84  a0 1c     LDY #$1c
```

The syntax for a range is `start:end`. Each of the two parts may have a prefix to indicate the radix, or no prefix to use the default radix.

Sometimes it is useful to have the starting and ending address in a range be the same, such as when you want to inspect a single byte of memory. In this case, you can enter `ff80:ff80` or simply `ff80`.

1.4 Assigning Labels

Large assembly language programs may have hundreds of procedures. It is difficult to remember the memory address of each procedure and the addresses may change if the program is reassembled.

You can add a label for any memory address using the `add_label` command. This label can then be used anywhere the address could be used:

```
.add_label ff80 start

.disassemble start
$ff80  d8        CLD
```

When using labels, you can also specify an offset (plus or minus):

```
.disassemble start:start+4
$ff80  d8        CLD
$ff81  a2 ff     LDX #$ff
$ff83  9a        TXS
$ff84  a0 1c     LDY #$1c
```

Offsets are interpreted like any other numbers. In the example above, `start+4` implies that the offset (`4`) uses the default radix. This could also be written as `start+$04` for explicit hexadecimal.

1.5 Breakpoints

It is possible to set breakpoints to stop execution when reaching a given address or label. Breakpoints are added using the `add_breakpoint` command:

```
.disassemble start:start+4
$ff80  d8        CLD
$ff81  a2 ff     LDX #$ff
$ff83  9a        TXS
$ff84  a0 1c     LDY #$1c
.add_breakpoint $ff84
Breakpoint 0 added at $FF84
.goto $ff80
Breakpoint 0 reached.
       PC  AC XR YR SP NV-BDIZC
6502: ff84 00 ff 00 ff 10110000
```

Note that a number is assigned to each breakpoint, similar to how VICE operates. Deleting a breakpoint can be done via the `delete_breakpoint` command using the breakpoint identifier given by `add_breakpoint`:

```
.add_breakpoint $ff84
Breakpoint 0 added at $FF84
.delete_breakpoint 0
Breakpoint 0 removed
```

Breakpoint can be listed using the `list_breakpoint` command:

```
.add_breakpoint $1234
Breakpoint 0 added at $1234
.add_breakpoint $5678
Breakpoint 1 added at $5678
.add_breakpoint $9ABC
Breakpoint 2 added at $9ABC
.show_breakpoints
Breakpoint 0 : $1234
Breakpoint 1 : $5678
Breakpoint 2 : $9ABC
```

Keep in mind that breakpoint identifiers are not recycled throughout a session, this means that if you add three breakpoints (#0, #1, #2) and then delete breakpoint #1, the next breakpoint you add will be breakpoint #3, not #1. Also, invoking `reset` clears breakpoints too, not just labels.

CHAPTER 2

Command Reference

`add_breakpoint <address|label>`

Sets a breakpoint on execution at the given address or at the address represented by the given label:

```
.add_breakpoint $1234
.add_label f000 start
.add_breakpoint start
```

Breakpoints get a numeric identifier to be used with `delete_breakpoint`, the list of identifiers can be retrieved with `show_breakpoints`.

`add_label <address> <label>`

Assign a label to an address:

```
.add_label f000 start
```

Once defined, the label may be used in place of the address in other commands. If a label already exists at the address, it will be silently overwritten.

`assemble <address> [<statement>]`

Assemble a single statement at an address:

```
.assemble c000 lda $a0,x
$c000  b5 a0     LDA $a0,X
```

If no statement is given, interactive assembly mode will start:

```
.assemble c000
$c000
```

Enter a statement and it will be assembled at the current address. The address will then be incremented and another statement may be entered. Press Enter or Return without entering a statement to exit interactive assembly mode.

If you have defined labels with add_label, you may use those labels in the address and the operand.

cd <path>

Change the current working directory to the path specified:

```
.cd /path/to/my/files
/path/to/my/files
```

After changing the directory, the new working directory will be displayed. The default working directory is the directory from which the monitor was started.

cycles

Display the number of cycles that the microprocessor has run since it was last reset:

```
.cycles
12
```

delete_breakpoint <breakpoint_id>

Removes the breakpoint associated with the given identifier:

```
.add_breakpoint $1234
Breakpoint 0 added at $1234
.add_label f000 start
.add_breakpoint start
Breakpoint 1 added at $F000
.delete_breakpoint 0
Breakpoint 0 removed
```

The list of identifiers added with `add_breakpoint` can be retrieved with `show_breakpoints`.

delete_label <label>

Delete a label that was previously defined with `add_label`:

```
.delete_label foo
```

If the label does not exist, the command will fail silently.

disassemble <address_range>

Disassemble a range of memory:

```
.disassemble ff80:ff84
$ff80  d8        CLD
$ff81  a2 ff     LDX #$ff
$ff83  9a        TXS
$ff84  a0 1c     LDY #$1c
```

The disassembly will use the instruction set of the selected MPU. For example, the extra instructions of the 65C02 will only be displayed if a 65C02 MPU is selected. On an NMOS 6502, those instructions would be disassembled as `???`.

If labels have been defined, they will be substituted for addresses in the operands.

fill <address_range> <byte> [<byte> <byte> ...]

Fill a range of memory using one or more bytes from the list:

```
.fill c000:c003 aa bb
Wrote +4 bytes from $c000 to $c003

.mem c000:c003
c000:  aa  bb  aa  bb
```

If the range is larger than the number of bytes in the list, the list will repeat as shown above.

goto <address>
Set the program counter to an address and resume execution:

```
.goto c000
```

help [<command>]
Display help for all commands or a single command. If no command is given, a list of commands will be displayed:

```
.help
```

If a command is given, help for that comand is displayed:

```
.help disassemble
disassemble <address_range>
Disassemble instructions in the address range.
```

show_breakpoints
Lists all the breakpoints that have been set so far:

```
.add_breakpoint $1234
Breakpoint 0 added at $1234
.add_breakpoint $5678
Breakpoint 1 added at $5678
.add_breakpoint $9ABC
Breakpoint 2 added at $9ABC
.show_breakpoints
Breakpoint 0 : $1234
Breakpoint 1 : $5678
Breakpoint 2 : $9ABC
```

load <filename> <address>
Load a binary file into memory starting at the address specified:

```
.load hello.bin c000
Wrote +29 bytes from $c000 to $c01c
```

The file will be loaded relative to the current working directory. You may also specify an absolute path. If the filename contains spaces, use quotes around it:

```
.load "say hello.bin" c000
Wrote +29 bytes from $c000 to $c01c
```

Note: Unlike the VICE monitor, Py65Mon's `load` command does not expect the first two bytes to be a Commodore-style load address. It will start reading the data at byte 0, not byte 2.

If the filename is a URL, it will be retrieved:

```
.load https://github.com/mnaberez/py65/raw/0.11/examples/ehbasic.bin 0000
Wrote +65536 bytes from $0000 to $ffff
```

mem <address_range>
Display the contents of memory an address range:

```
.mem ff80:ffa0
ff80:  d8  a2  ff  9a  a0  1c  b9  bb  ff  99  04  02  88  d0  f7  b9  d8  ff
ff92:  f0  06  20  a6  e0  c8  d0  f5  20  a3  e0  90  fb  29  df
```

The contents will be wrapped to the terminal width specified by the `width` command.

mpu [<mpu_name>]
Display or set the current microprocessor. If no argument is given, the current microprocessor will be displayed:

```
.mpu
Current MPU is 6502
Available MPUs: 6502, 65C02, 65Org16
```

If an argument is given, the microprocessor will be changed:

```
.mpu 65C02
Reset with new MPU 65C02
```

The default microprocessor is `6502`, the original NMOS 6502 from MOS Technology.

pwd
Display the current working directory:

```
.pwd
/home/mnaberez
```

quit
Quit the monitor:

```
.quit
```

radix [<H|D|O|B>]
Display or set the default radix that is assumed for numbers that have no prefix. If no argument is given, the default radix is displayed:

```
.radix
Default radix is Hexadecimal
```

If an argument is given, the default radix will be changed:

```
.radix d
Default radix is Decimal
```

The default radix may be changed to Hexadecimal, Decimal, Octal, or Binary.

registers [<name=value>, <name=value>, ...>]
Display or change the registers of the microprocessor. If no arguments are given, the registers are displayed:

```
.registers

       PC  AC XR YR SP NV-BDIZC
6502: 0000 00 00 00 ff 00110000
```

Registers can changed giving `name=value`, separated by commas if multiple registers are to be changed:

```
.registers a=02, x=04
```

```
       PC  AC XR YR SP NV-BDIZC
6502: 0000 02 04 00 ff 00110000
```

reset
Reset the microprocessor to its default state. All memory will also be cleared:

```
.reset
```

return
Continue execution and return to the monitor just before the next RTS or RTI is executed:

```
.return
```

save <filename> <start_address> <end_address>
Save the specified memory range to disk as a binary file:

```
.save hello.bin c000 c01c
Wrote +29 bytes from $c000 to $c01c
```

The file will be saved relative to the current working directory. You may also specify an absolute path. If the filename contains spaces, use quotes around it:

```
.save "say hello.bin" c000 c01c
Wrote +29 bytes from $c000 to $c01c
```

Note: Unlike the VICE monitor, Py65Mon's `save` command does not write the first two bytes as a Commodore-style load address. It will start writing the data at byte 0, not byte 2.

---

show_labels
Display labels that have been defined with `add_label`:

```
.show_labels
ffd2: charout
```

step
Execute a single instruction at the program counter. After the instruction executes, the next instruction is disassembled and printed:

```
       PC  AC XR YR SP NV-BDIZC
6502: 0000 00 00 00 ff 00110000
.registers pc=c000

       PC  AC XR YR SP NV-BDIZC
6502: c000 00 00 00 ff 00110000
.step
$c002  a9 42     LDA #$42

       PC  AC XR YR SP NV-BDIZC
6502: c002 00 00 00 ff 00110000
.
```

In the example above, the instruction at `$C000` executes and the monitor prompt returns.

Note: After the instruction executes, the disassembly of the **next** instruction is printed. This allows you to see

what will be executed on the next step.

tilde

Display a number in the supported number systems:

```
.~ c000
+49152
$c000
140000
1100000000000000
```

The number will be displayed in this order: decimal, hexadecimal, octal, and then binary.

version

Display version information:

```
.version

Py65 Monitor
```

width [<columns>]

Display or set the terminal width. The width is used to wrap the output of some commands like `mem`. With no argument, the current width is displayed:

```
.width
Terminal width is 78
```

If a column count is given, the width will be changed:

```
.width 130
Terminal width is 130
```

The number of columns is always specified as a decimal number.