Assembly code tools for the Supercon.6 Badge

Assembler and disassembler for the 4-bit Supercon.6 Badge designed by Voja Antonic. Tools and documenation originally written by Mike Szczys.

Usage:

It's recommended that you use `.asm` for your source code file. The assembler will produce a binary `.hex` file which can be uploaded to the badge via serial (save/load examples found below).

```
python3 assemble.py input_filename.asm
```

The disassembler will produce `.s` files so as not to overwrite existing user-provided `.asm` files.

```
python3 disassemble.py binary_file.hex
```

Both scripts have flags to configure what output is shown to the user. Use `-h` for more info about these options.

Serial Operations on Linux

Each time the USB-to-Serial cable is plugged in, set the port to raw mode: `stty -F /dev/ttyUSB0 raw`

When badge is in DIR mode, programs can be sent from the computer to the badge via serial by pressing `load` button. Press the button then run this command on the computer: `cat out.hex > /dev/ttyUSB0`

Programs can also be sent from the badge to the computer via serial by pressing the `save` button. Before doing so, prepare the computer with this command: `cat /dev/ttyUSB0 > out.hex` and press CTRL-C when done. The resulting file may be view with a hex editor such as `xxd out.hex`

Pseudo-operations

In addition to the opcodes that directly translate to machine code, this assembler provides a number of pseudo-opcodes to assist in writing code: labels, EQU, ORG, GOTO, GOSUB, BYTE, ASCII

Labels:

```
MOV R0,R7

timesfour:
ADD R7,R7
ADD R7,R7
RET R0,0

startloop:
INC R8
```

Labels are variables that represent the register number of the next valid assembly code that follows. When the assembler runs, it makes a first pass through the code, recording line numbers of the labels wherever they are declared. Labels can then be used to with instructions that set the Program Counter to these values. In the above example, `GOSUB timesfour` and `GOTO startloop` are two examples of how these may be used.

EQU

```
x EQU 12
y EQU x-4
MOV R0,y
```

EQU assigns a numeric literal to a variable. Place the variable name before EQU and the value assignment afterward. As shown in the example, mathematical operations and other symbols may be used in the assignment. This example will produce machine code that loads the value 8 to register R0: `MOV R0,0b1000`

ORG

```
MOV R0,R7
ORG 32
MOV R8,R0 ; This line will be placed at register 32 (0x020)
```

The ORG command places the next line of valid assembly code at the register address specified. Machine code 0x000 (equivalent to `CP R0,0`) will be generated to fill the space until the address specified by ORG is reached.

GOTO

```
MOV R7,4
start:
MOV R0,R7
INC R0,
GOTO start
```

GOTO moves the program counter to the specified value. This can be a number [0..4095] or a label. This pseudo-op will generate two lines of machine code, one to load the PCH, PCM values, and another to write to PCL, which triggers the Program Counter change. The GOTO in the example generates this code:

```
MOV PC,[0b0000:0b0000]
MOV PCL,0b0001
```

GOSUB

```
timesfour:
ADD R7,R7
ADD R7,R7
RET R0,0

startloop:
INC R8
MOV R7,R8
GOSUB timesfour
```

GOSUB is nearly identical to GOTO and also geneartes two lines of machine code. The specified value may be a number [0..4095] or a label. The difference with GOSUB is that it writes the lowest nibble to JSR (instead of PCL). This still changes the Program Counter to the desired value, but also causes the processor to first store the current Program Counter value for use by the RET (return) function. The GOSUB in the example generates this code:

```
MOV PC,[0b0000:0b0000]
MOV JSR,0b0000
```

BYTE

```
BYTE 0b11100101
BYTE 0b00010110
```

The BYTE pseudo-op is used to store an 8-bit value as two 4-bit numbers, low nibble first, by generating two RET instructions. To retrieve the value, call each RET in sequence while checking the contents of R0. An example of this can be seen in the font set used by Voja Antonic's scrolling text program. The demo above generates the following code:

```
RET R0,0b0101
RET R0,0b1110
RET R0,0b0110
RET R0,0b0001
```

ASCII

```
ASCII "Hello!"
```

Strings can be stored as two RET instructions per character by using the ASCII pseudo-op. Strings must start and end with quotation marks (") and the semicolon (;) is forbidden as it will appear to the assembler as the beginning of a comment. The following characters are valid (beginning with a space):

```
 !"#$%&\'()*+,-./0123456789:<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^_`abcdefghijklmnopqrstuvwxyz{|}~
```

An example of the `ASCII` pseudo-op can be seen in Voja Antonic's scrolling text program.The example above will product the following instructions:

```
RET R0,0b1000
RET R0,0b0100
RET R0,0b0101
RET R0,0b0110
RET R0,0b1100
RET R0,0b0110
RET R0,0b1100
RET R0,0b0110
RET R0,0b1111
RET R0,0b0110
RET R0,0b0001
RET R0,0b0010
```

Modifier Keywords

```
elite EQU 0x539 ; 1337 or 0b 0101 0011 1001

MOV PC,[HIGH elite : MID elite]
MOV PCL,LOW elite
```

The modifying keywords `HIGH`, `MID`, and `LOW` can be used to access the three 4-bit nibbles of a 12 bit value. The example above assigns a 12-bit register the variable `elite`, then uses the keywords to move the program counter to that location. The example will product the following code:

```
MOV PC,[0b0101:0b0011]
MOV PCL,0b1001
```