JSHypotheticalMachine

JSHypotheticalMachine is a hypothetical machine designed for those who want learn about how a computer works under the hood.

It can be used by teachers at the school to teach students Computer Architecture.

JSHypotheticalMachine has only 11 'native' instructions, plus 4 extended instructions to work with the stack, which allows developers to call subroutines and return from them.

This project is written in Javascript and is opensource. Feel free to use it. If possible give me some feedback and/or help to improve it by sending pool requests.

Technical specification

Addressing mode:

It cannot have PC-relative addressing mode because the operand is 1 byte long, and if we have PC-relative addressing mode there is no room to have negative and positive operands and at the same time access all memory space. If memory has 256 bytes, the operand should have 9 bits to address relatively all memory space, the 9th bit would be used to sign.

Due this limitations and because the goal of this project is to make things easy and simple, the effective address for an absolute instruction address is the address parameter itself with no modifications.

Data/Address wide:

Every instruction is 8bit long. Some instructions have operands, which are also 8bit long.

The CPU internally has 3 registers, as follows:

- An Accumulator (AC)
- A Program Counter (PC)
- A State register, which has 2 flags:
  - Negative (N) - Meaning that the last operation resulted in a negative number.
  - Zero (Z) - Meaning that the last operation resulted in zero.

Instruction set

Code	Instruction	Description
0000	NOP	No Operation
0001	STA	MEM[PC+1] << AC
0010	LDA	AC << MEM[MEM[PC+1]]
0011	ADD	AC << MEM[MEM[PC+1]] + AC
0100	OR	AC << MEM[MEM[PC+1]] OR AC
0101	AND	AC << MEM[MEM[PC+1]] AND AC

Code	Instruction	Description
0110	NOT	AC << NOT AC
1000	JMP	PC << MEM[PC+1]
1001	JN	IF N=1 THEN PC << MEM[PC+1]
1010	JZ	IF Z=1 THEN PC << MEM[PC+1]
1111	HLT	Execution stops (halt)

Extended instructions

Code	Instruction	Description
1011	CALL	Call a subroutine (TOS << PC, PC << MEM[PC+1])
1100	RET	Returns from a subroutine (PC << TOS)
1011	PUSH	Push the AC into TOS (TOS << AC)
1011	POP	Pop the TOS into AC (AC << TOS)

Compiler

Syntax

.at

Specify where to put in memory the next instructions.

.def

Defines a constant.

E.q.:

```
.def MY_CONST_ADDRESS 0x55
lda MY_CONST_ADDRESS
```

.db

Specify a value to a given memory address.

E.q.: .db 0x5 0x6 Stores the value 0x6 at the memory address 0x5

NOTE: You can use .db at any point in your program. However, .db has precedence, so if you put some instruction on the same place, the .db will override the instruction.

#

It is an indication of a commentary, the line that starts with # will be considered as a comment.

:label

Labels are defined when a **:** starts a new line. If some :label appears on any other place in the line, it will be considered as you are using it.

NOTE: You can use a label before create it. However, if the label is not created until the end of the program, an error will be raised.

Snippet of code about defining and using a label:

```
:my_label
add 0x01
jmp :my_label
```

This is an example of code to draw lines on a LCD:

```
.def LCD_OP  0xfa
.def LCD_ARG0  0xfb
.def LCD_ARG1  0xfc

.db LCD_OP 0x00
.db LCD_ARG0 0x00
.db LCD_ARG1 0x00

.def X_INC 0xf9
.def Y_INC 0xf8

.db X_INC 60
.db Y_INC 50

.def NO_OP 0xf7
.def LINETO_OP 0xf6

.db NO_OP 0x00
.db LINETO_OP 0x02

jmp :begining

.at 4
    reti

:begining

:loop
    call :no_op
    lda LCD_ARG0
    add X_INC
    sta LCD_ARG0

    lda LCD_ARG1
    add Y_INC
    sta LCD_ARG1
    call :line_to_op
    jmp :loop

:no_op
    lda NO_OP
    sta LCD_OP
    ret
```

```
:line_to_op
    lda LINETO_OP
    sta LCD_OP
    ret
```