PCS-3216

May 13, 2020

1 Emulador de uma Máquina de Von Neumann

Projeto desenvolvido para a disciplina de Sistemas de Programação da Escola Politécnica da USP.

1.1 Compilação

O projeto foi desenvolvido no Ubuntu 19.10, e deve funcionar para qualquer sistema Linux. Basta executar o comando `make` para compilar os arquivos.

1.2 Arquitetura da Máquina de Von Neumann

A Máquina de Von Neumann (MVN) implementada possui instruções de 16 *bits*, sendo os quatro primeiros reservados para o código de operação (opcode) e os restantes para operando da instrução. Possui apenas dois registradores, o contador de instruções (PC) e o acumulador (AC). O endereçamento de memória utiliza 12 *bits*, de forma que são disponíveis 4096 posições de memória acessíveis. Cada posição possui um *byte* de largura.

As operações disponíveis são:

opcode	mnemônico	efeito
0xxx	JP	PC = xxx
1xxx	JZ	if (AC == 0) PC = xxx
2xxx	JNE	if (AC < 0) PC = xxx
3xxx	LV	AC = 0xx (último byte)
4xxx	ADD	AC += mem[xxx]
5xxx	SUB	AC -= mem[xxx]
6xxx	MUL	AC *= mem[xxx]
7xxx	DIV	AC /= mem[xxx]
8xxx	LD	AC = mem[xxx]
9xxx	MM	mem[xxx] = AC
Axxx	SC	mem[xxx, xxx+1] = PC; PC = xxx+2
Bxxx	RS	PC = xxx
Cxxx	HM	para a execução
Dxxx	GD	lê um byte do arquivo de entrada para AC
Exxx	PD	escreve AC no arquivo de saída
Fxxx	OS	chamada de sistema (ainda não implementada)

A parte de *input/output* é feita por meio de arquivos em *plain text* dentro da pasta `/filesystem`.

Por *default*, o arquivo de entrada é o `input.txt` e o de saída é o `output.txt`.

1.2.1 O Motor de Eventos

A MVN foi implementada utilizando o conceito de um motor de eventos. Isto é, em um loop de simulação, a MVN busca uma instrução na memória, decodifica sua operação e operandos e ativa os procedimentos específicos de reação para a simulação das intstruções correspondentes. Ao final de um ciclo, atualiza o ponteiro (*program counter*) para o endereço da prócima instrução a ser executada.

1.3 Utilização

A interface com o usuário é feita por meio de uma interface de linha de comando (CLI):

```
__     __          _   _                                                __     ____  __
\ \   / /__  _ __ | \ | | ___ _   _ _ __ ___   __ _ _ __  _ __   \ \   / /  \/  |
 \ \ / / _ \| '_ \|  \| |/ _ \ | | | '_ ` _ \ / _` | '_ \| '_ \   \ \ / /| |\/| |
  \ V / (_) | | | | |\  |  __/ |_| | | | | | | (_| | | | | | | |   \ V / | |  | |
   \_/ \___/|_| |_|_| \_|\___|\__,_|_| |_| |_|\__,_|_| |_|_| |_|    \_/  |_|  |_|

                                                                        --Ver. 1.0--

Type 'help' for available commands.

~ $
```

Os comandos disponíveis estão apresentados a seguir. São descritos em maior detalhe por meio do comando `help`.

comando	efeito
`assemble [src] [-w] [-o out]`	monta um arquivo-fonte em assembly para um arquivo de listagem e um em código-objeto.
`exit`	sai do programa.
`help`	imprime a mensagem de ajuda.
`load [program]`	carrega um programa em código-objeto para a memória da MVN.
`ls`	lista os arquivos na pasta `/filesystem`.
`print [file]`	imprime o conteúdo de um arquivo.
`md [address] [range]`	imprime o conteúdo da memória da MVN em *chunks* de 16 posições seguidas.
`mm [address] [data]`	modifica o conteúdo da memória da MVN.
`rm [file(1)]...[file(n)]`	remove um ou mais arquivos da pasta `/filesystem`
`run [address]`	aciona a execução da MVN. É possível especificar o endereço inicial.
`set [register] [data]`	modifica o conteúdo de um registrador (AC ou PC).

comando	efeito
`status`	imprime o conteúdo dos registradores da MVN.
`step [address]`	executa as instruções da MVN passo-a-passo, pressionando `enter`.
`turn [on off]`	liga ou desliga a MVN.

1.3.1 Assembler

O *assembler* da MVN foi escrito em alto-nível. Trata-se de um montador **absoluto**, isto é, todas as posições de memória já estão definidas. Para utilizá-lo, basta digitar o comando `assemble [src]` na linha de comando do programa, em que `[src]` é um arquivo-fonte em *assembly* localizado no diretório `/filesystem`.

A estrutura de um programa típico da MVN é descrita a seguir:

```
; Comentários iniciam-se com ';'

        ORG $100  ; Posição da primeira instrução

START:  ADD $120  ; Labels terminam com ':'
        SUB 440   ; Operandos podem estar tanto em decimal quanto em hexadecimal
        MUL VAR1  ; Operandos podem ser outras labels também
        GD        ; Tanto GD quanto
        PD        ; PD não recebem operandos.
        ...
        JP  START

        ORG $120  ; Posição da próxima instrução

VAR1:   CON $F2   ; Constantes só possuem um byte de tamanho
        END START ; Fim do programa
        MM  VAR1  ; Não será processado pelo assembler
```

Note que, além das instruções de máquina da MVN, existem três pseudo-instruções do assembler:

mnemônico	efeito
ORG	Modifica o endereço das próximas instruções.
CON	Define uma constante de um byte.
END	Indica ao *assembler* que o programa acabou.

Os endereços disponíveis para os programas do usuário iniciam-se na posição `$070` (ver *loader*)

1.3.2 Loader

O *loader* foi implementado em linguagem de máquina, e ocupa os primeiros 148 bytes da memória da MVN. Ele é injetado pela interface na inicialização da MVN. Para utilizá-lo, basta digitar o

comando `load [program]` na linha de comando do programa, em que `[program]` é um código-objeto no formato '.vnc'.

O formatao '.vnc' nada mais é do que um arquivo em *plain-text*, em que os dados são escritos em "hexadecimal". Os dados a serem escritos na memória da MVN são divididos em blocos, cuja estrutura é:

```
[origem][tamanho][     dados    ][checksum]
   2*       1*         1-15*         1*

* Tamanho em bytes (cada byte são dois dígitos hexadecimais).
```

O programa-fonte do *loader* encontra-se a seguir:

```
; LOADER (starts at $000)
            ld      fail
            sub     fail
            mm      fail        ; Resets fail flag
            jp      readAddr    ; Jumps data area to first instruction

; //---------Data Area----------//

counter:    con     $00         ; Counts number of bytes of data to be read
one:        con     $01         ; 1 constant
opcode:     con     $90         ; MM opcode value
fail:       con     $00         ; If a chacksum failed
checksum:   con     $00         ; Calculated Checksum
compAux:    con     $FF         ; Auxiliar to calculate 1's complement

; //--------Main Program--------//

; Reads base address:
readAddr:   ld      checksum    ; Loads checksum.
            sub     checksum    ; Zero.
            mm      checksum    ; Reset checksum.
            sc      readByte    ; Reads first address byte from file
            add     opcode      ; Adds upper address to opcode
            mm      write_1     ; Writes result to write_1
            sc      readByte    ; Reads second address byte from file
            mm      write_2     ; Writes to write_2

; If the read address was zero, we end the program:
            jz      zero1       ; If the last byte was zero, we check the upper byte.
            jp      readSize    ; Else, we continue to read the size byte.
zero1:      ld      write_1     ; Loads upper byte (OP + upper address).
            sub     opcode      ; Removes opcode.
            jz      eop         ; If it is also zero, we terminate the program.

readSize:   sc      readByte    ; Reads byte count from file
            mm      counter     ; Writes byte count to counter
```

```
; Memory writing loop
loop:       sc      readByte    ; Reads a byte of data
write_1:    con     $90         ; MM first byte (will be read as instruction)
write_2:    con     $00         ; MM second byte (will be read as instruction)
            ld      write_2     ; Loads MM second byte (lower address byte)
            add     one         ; write_2 += 1
            mm      write_2     ; Writes new lower address byte
            jz      addUpper    ; if equals zero (overflow) we add one to upper byte.
write_3:    ld      counter     ; Loads value from counter
            sub     one         ; Counter--
            mm      counter     ; Writes new value of counter
            jz      checkChks   ; Compares checksums if counter = 0
            jp      loop        ; Goes back to loop if counter != 0

; Adds one to upper MM byte
addUpper:   ld      write_1     ; Loads upper MM byte.
            add     one         ; Write_1++
            mm      write_1     ; Writes result.
            jp      write_3     ; Continues back to loop

; Compares the checksums
checkChks:  ld      compAux     ; Loads $FF
            sub     checksum    ; One's complement to calculate checksum
            mm      checksum    ; Saves computed checksum to memory
            gd                  ; Reads checksum from file
            sub     checksum    ; Compares both checksums
            jz      readAddr    ; If they are different, we continue and read next address.
            mm      fail        ; If they are differetn, we raise error flag and terminate prog

; Terminates program
eop:        hm      readAddr    ; Halts machine.

; //--------Subroutines---------//

; Gets a byte and adds to checksum.
readByte:   jp      $000        ; Return address.
            gd                  ; Reads a byte of data.
            mm      data        ; Saves read byte.
            add     checksum    ; Adds to checksum.
            mm      checksum    ; Saves results to checksum variable.
            ld      data        ; Recovers data that was read.
            rs      readByte    ; Returns from soubroutine.
; ReadByte data
data:       con     $00         ; Data that was read.
```