Gustavo P. Gouveia (6482819), Victor Lassance (6431325)

Relatório de Compiladores
Quarta Etapa
Especificação do Ambiente de Execução
Linguagem de programação CZAR

Texto apresentado à Escola Politécnica da Universidade de São Paulo como requisito para a aprovação na disciplina Linguagens e Compiladores no quinto módulo acadêmico do curso de graduação em Engenharia de Computação, junto ao Departamento de Engenharia de Computação e Sistemas Digitais (PCS).

Universidade de São Paulo

Escola Politécnica

Engenharia de Computação - Curso Cooperativo

Professor: Ricardo Luis de Azevedo da Rocha

São Paulo

2013

Sumário

1 Introdução

Até a terceira entrega do compilador, focamos nas duas primeiras etapas da compilação de um programa, a análise léxica e a análise sintática. Para essa entrega, focaremos no ambiente de execução. O compilador por nós criado terá como linguagem de saída um programa que será executado na máquina virtual chamada Máquina de von Neumann (MVN).

O Modelo de von Neumann procura oferecer uma alternativa prática, disponibilizando ações mais poderosas e ágeis em seu repertório de operações que o modelo de Turing. Isso viabiliza, codificações muito mais expressivas, compactas e eficientes. Para isso, a Máquina de von Neumann utiliza:

- Memória endereçável, usando acesso aleatório
- Programa armazenado na memória, para definir diretamente a função corrente da máquina (ao invés da Máquina de Estados Finitos)
- Dados representados na memória (ao invés da fita)
- Codificação numérica binária em lugar da unária
- Instruções variadas e expressivas para a realização de operações básicas muito frequentes (ao invés de sub-máquinas específicas)
- Maior flexibilidade para o usuário, permitindo operações de entrada e saída, comunicação física com o mundo real e controle dos modos de operação da máquina

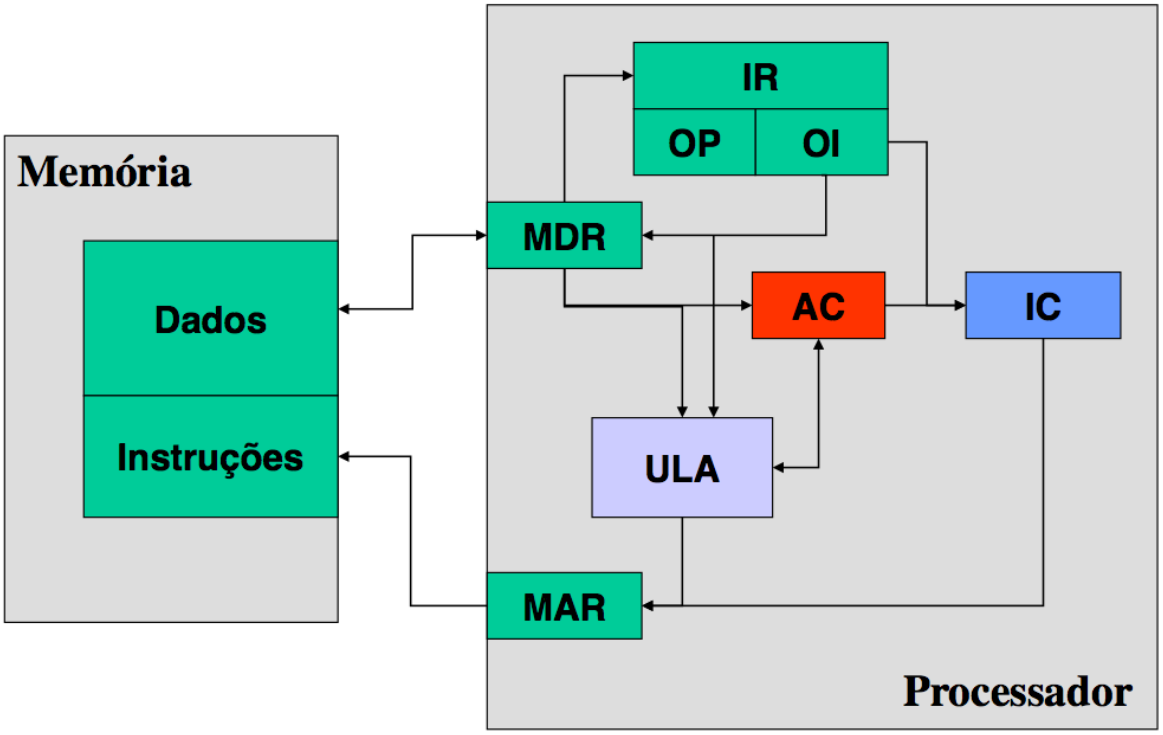
Dessa forma, utilizaremos essa máquina para executar nosso compilador e realizar os testes necessários.

A arquitetura de Von Neumann é composta por um processador e uma memória principal. Na memória principal armazenam-se as instruções do código-fonte e os dados, sendo a divisão mostrada na figura 1 apenas ilustrativa. Além da Unidade Lógica Aritmética (ULA), responsável pelo processamento de operações lógicas e aritméticas, o processador possui um conjunto de elementos físicos de armazenamento de informações e é comum dividir esses componentes nos seguintes módulos resgistradores:

1. MAR - Registrador de endereço de memória

   Indica qual é a origem ou o destino, na memória principal, dos dados contidos no registrador de dados de memória.

Figura 1 – Arquitetura MVN



2. MDR - Registrador de dados da memória

   Serve como ponte para os dados que trafegam entre a memória e os outros elementos da máquina.

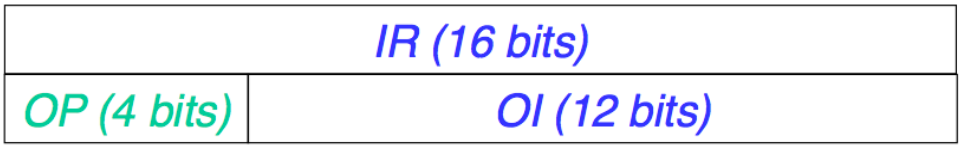
3. IC - Registrador de endereço de instrução

   Indica a cada instante qual será a próxima instrução a ser executada pelo processador.

4. IR - Registrador de instrução

   Contém a instrução atual a ser executada. é subdividido em dois outros registradores, como na figura 2.

Figura 2 – Estrutura do registro de instrução (IR)



   a) OP - Registrador de código de operação

      Parte do registrador de instrução que identifica a instrução que está sendo executada.

b) OI - Registrador de operando de instrução

Complementa a instrução indicando o dado ou o endereço sobre o qual ela deve agir.

5. AC - Acumulador

Funciona como a área de trabalho para execução de operações lógicas ou aritméticas. Acumula o resultado de tais operações.

A máquina executa um programa em diversos passos, listadas abaixo:

1. Determinação da Próxima Instrução a Executar

2. Fase de Obtenção da Instrução

Obter na memória, no endereço contido no registrador de Endereço da Próxima Instrução, o código da instrução desejada.

3. Fase de Decodificação da Instrução

Decompor a instrução em duas partes: o código da instrução e o seu operando, depositando essas partes nos registradores de instrução e de operando, respectivamente. Selecionar, com base no conteúdo do registrador de instrução, um procedimento de execução dentre os disponíveis no repertório do simulador (passo 4).

4. Fase de Execução da Instrução

Executar o procedimento selecionado no passo 3, usando como operando o conteúdo do registrador de operando, preenchido anteriormente.

Caso a instrução executada não seja de desvio, incrementar o registrador de endereço da próxima instrução a executar. Caso contrário, o procedimento de execução já terá atualizado convenientemente tal informação.

a) Execução da instrução (decodificada no passo 3)

De acordo com o código da instrução a executar (contido no registrador de instrução), executar os procedimentos de simulação correspondentes (detalhados adiante).

b) Acerto do registrador de Endereço da Próxima Instrução para apontar a próxima instrução a ser simulada:

Incrementar o registrador de Endereço da Próxima Instrução.

2 Instruções da Linguagem de Saída

As instruções da MVN podem ser resumidas pela tabela da figura 3.

Figura 3 – Lista de instruções da MVN

Código (hexa)	Instrução	Operando
0	Desvio incondicional	endereço do desvio
1	Desvio se acumulador é zero	endereço do desvio
2	Desvio se acumulador é negativo	endereço do desvio
3	Deposita uma constante no acumulador	constante relativa de 12 bits
4	Soma	endereço da parcela
5	Subtração	endereço do subtraendo
6	Multiplicação	endereço do multiplicador
7	Divisão	endereço do divisor
8	Memória para acumulador	endereço-origem do dado
9	Acumulador para memória	endereço-destino do dado
A	Desvio para subprograma (função)	endereço do subprograma
B	Retorno de subprograma (função)	endereço do resultado
C	Parada	endereço do desvio
D	Entrada	dispositivo de e/s
E	Saída	dispositivo de e/s
F	Chamada de supervisor	constante (**)

(**) por ora, este operando (tipo da chamada) é irrelevante, e esta instrução nada faz.

A seguir, especificaremos o que é realizado pela máquina ao executar cada tipo de operação.

- Registrador de instrução = 0 (desvio incondicional)

  Modifica o conteúdo do registrador de Endereço da Próxima Instrução (IC) armazenando nele o conteúdo do registrador de operando (OI)

  IC := OI

- Registrador de instrução = 1 (desvio se acumulador é zero)

  Se o conteúdo do acumulador (AC) for zero, então modifica o conteúdo do registrador de Endereço da Próxima Instrução (IC), armazenando nele o conteúdo do registrador de operando (OI)

  Se AC = 0 então IC := OI

  Se não IC := IC + 1

- Registrador de instrução = 2 (desvio se negativo)

  Se o conteúdo do acumulador (AC) for negativo, isto é, se o bit mais significativo for 1, então modifica o conteúdo do registrador de Endereço da Próxima Instrução (IC) armazenando nele o conteúdo do registrador de operando (OI)

  Se AC < 0 então IC := OI

  Se não IC := IC + 1

- Registrador de instrução = 3 (constante para acumulador)

  Armazena no acumulador (AC) o número relativo de 12 bits contido no registrador de operando (OI), estendendo seu bit mais significativo (bit de sinal) para completar os 16 bits do acumulador

  AC := OI

  IC := IC +1

- Registrador de instrução = 4 (soma)

  Soma ao conteúdo do acumulador (AC) o conteúdo da posição de memória indicada pelo registrador de operando MEM[OI]. Guarda o resultado no acumulador

  AC := AC + MEM[OI]

  IC := IC + 1

- Registrador de instrução = 5 (subtração)

  Subtrai do conteúdo do acumulador (AC) o conteúdo da posição de memória indicada pelo registrador de operando MEM[OI]. Guarda o resultado no acumulador

  AC := AC - MEM[OI]

  IC := IC + 1

- Registrador de instrução = 6 (multiplicação)

  Multiplica o conteúdo do acumulador (AC) pelo conteúdo da posição de memória indicada pelo registrador de operando MEM[OI]. Guarda o resultado no acumulador

  AC := AC * MEM[OI]

  IC := IC + 1

- Registrador de instrução = 7 (divisão inteira)

  Dividir o conteúdo do acumulador (AC) pelo conteúdo da posição de memória indicada pelo registrador de operando MEM[OI]. Guarda a parte inteira do resultado no acumulador

  AC := int (AC / MEM[OI])

  IC := IC + 1

- Registrador de instrução = 8 (memória para acumulador)

  Armazena no acumulador (AC) o conteúdo da posição de memória endereçada pelom registrador de operando (OI)

  AC := MEM[OI]

  IC := IC + 1

- Registrador de instrução = 9 (acumulador para memória)

  Guarda o conteúdo do acumulador (AC) na posição de memória endereçada pelo registrador de operando (OI)

  MEM[OI] := AC

  IC := IC + 1

- Registrador de instrução = A (desvio para subprograma)

  Armazena o conteúdo do registrador de Endereço da Próxima Instrução (IC), incrementado de uma unidade, no registrador de endereço de retorno (RA). Armazena no registrador de Endereço da Próxima Instrução (IC) o conteúdo do registrador de operando (OI).

  RA := IC + 1

  IC := OI

- Registrador de instrução = B (retorno de subprograma)

  Armazena no registrador de Endereço da Próxima Instrução (IC) o conteúdo do registrador de endereço de retorno (RA), e no acumulador (AC) o conteúdo da posição de memória apontada pelo registrador de operando (OI)

  AC := MEM[OI]

  IC := RA

- Registrador de instrução = C (stop)

  Modifica o conteúdo do registrador de Endereço da Próxima Instrução (IC) armazenando nele o conteúdo do registrador de operando (OI) e para o processamento

  IC := OI

- Registrador de instrução = D (input)

  Aciona o dispositivo padrão de entrada e aguardar que o usuário forneça o próximo dado a ser lido. Transfere o dado para o acumulador

  Aguarda

  AC := dado de entrada

  IC := IC + 1

- Registrador de instrução = E (output)

  Transfere o conteúdo do acumulador (AC) para o dispositivo padrão de saída. Aciona o dispositivo padrão de saída e aguardar que este termine de executar a operação de saída

  dado de saída := AC

  aguarda

  IC := IC + 1

- Registrador de instrução = F (supervisor call)

  Não implementado: por enquanto esta instrução não faz nada.

  IC := IC + 1

Escrever um programa usando diretamente codificação binária não é uma tarefa simples, e tampouco agradável. Naturalmente, se um programa é muito grande ou se lida com diversas estruturas complexas (listas, etc.), a sua codificação se torna ainda mais difícil e complexa.

Por conta disso, torna-se imprescindível construir alguma abstração que facilite a programação e a verificação dos programas. A primeira idéia, mais natural, é utilizar o modelo de máquina existente e, a partir dele, definir nomes (mnemônicos) para cada instrução da máquina. Posteriormente, verifica-se que somente isso não basta, pois é necessário lidar com os endereços dentro de um programa (rótulos, operandos, sub-rotinas), com a reserva de espaço para tabelas, com valores constantes. Enfim, é necessário definir uma linguagem simbólica.

Para a construção de um montador, cujo esquema geral está representado na figura 4 pressupõe-se que sejam tratadas as seguintes questões:

- definição das instruções: determinar os mnemônicos que as representam simbolicamente;
- definição das pseudo-instruções: determinar os mnemônicos que as representam, bem como sua função para o montador.

As instruções para a MVN são apresentadas na figura 5.

Figura 4 – Esquema geral de um montador

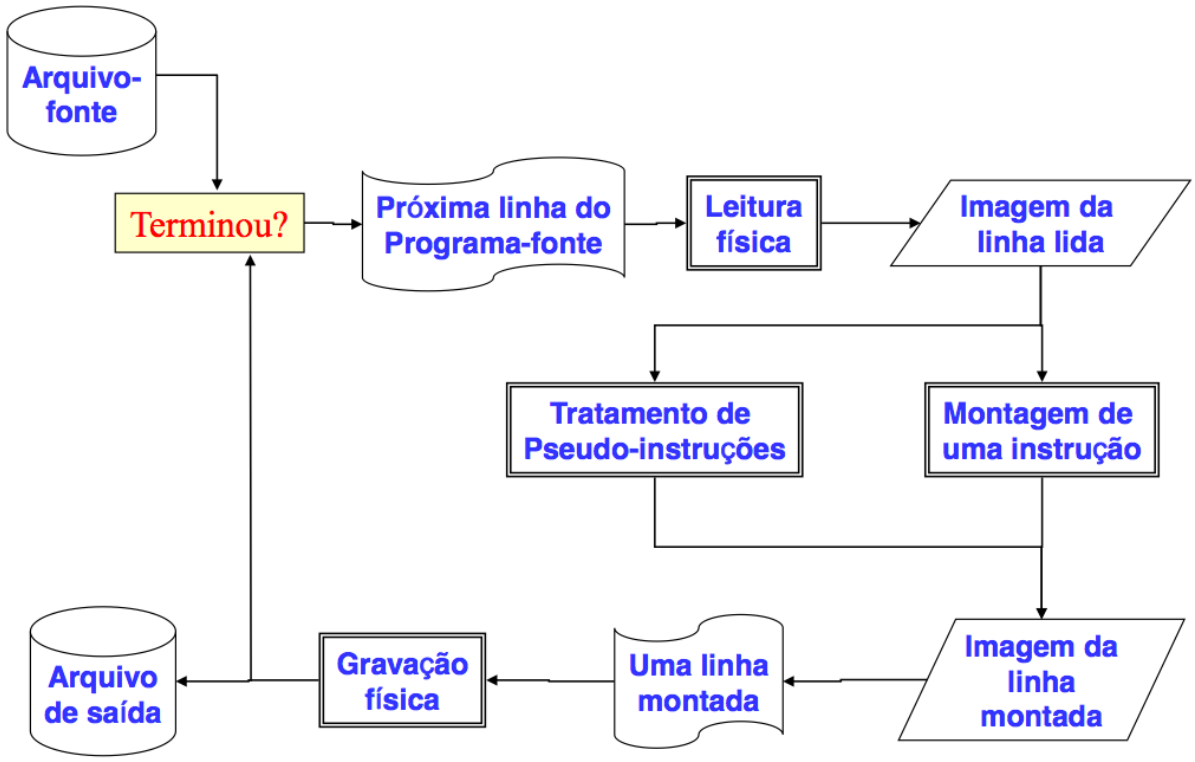


Figura 5 – Tabela de mnemônicos para a MVN (de 2 caracteres)

Operação **0** **Jump** Mnemônico **JP**	Operação **1** **Jump if Zero** Mnemônico **JZ**	Operação **2** **Jump if Negative** Mnemônico **JN**	Operação **3** **Load Value** Mnemônico **LV**
Operação **4** **Add** Mnemônico **+**	Operação **5** **Subtract** Mnemônico **–**	Operação **6** **Multiply** Mnemônico *****	Operação **7** **Divide** Mnemônico **/**
Operação **8** **Load** Mnemônico **LD**	Operação **9** Move to **Memory** Mnemônico **MM**	Operação **A** **Subroutine Call** Mnemônico **SC**	Operação **B** **Return from Sub.** Mnemônico **RS**
Operação **C** **Halt Machine** Mnemônico **HM**	Operação **D** **Get Data** Mnemônico **GD**	Operação **E** **Put Data** Mnemônico **PD**	Operação **F** **Operating System** Mnemônico **OS**

3 Pseudoinstruções da Linguagem de Saída

Programas absolutos são executáveis estritamente nas posições de memória em que foram criados, tornando difícil a manutenção e o trabalho em equipe. A utilização de programas relocáveis permitem sua execução em qualquer posição de memória, tornando possível utilizar partes de código projetadas externamente (uso de bibliotecas, por exemplo).

Para que se possa exprimir um programa relocável e com possibilidade de construção em módulos, separadamente desenvolvidos, é necessário que:

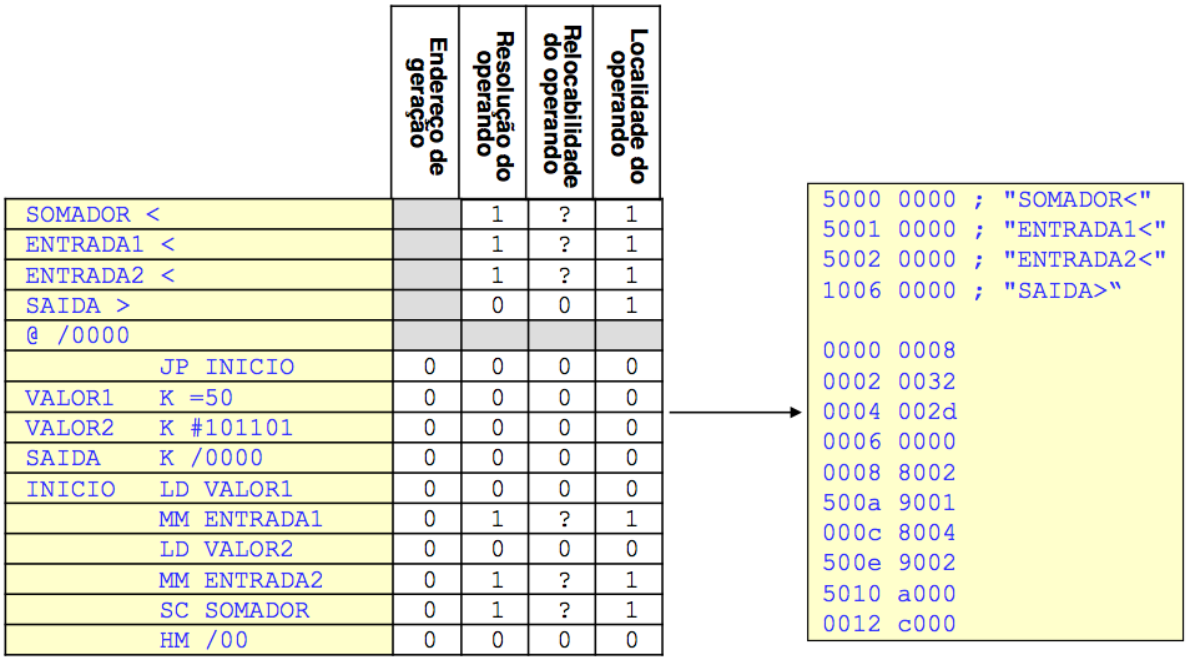
- Haja a possibilidade de representar e identificar endereços absolutos e endereços relativos;
- Um programa possa ser montado sem que os seus endereços simbólicos estejam todos resolvidos;
- Seja possível identificar, em um módulo, símbolos que possam ser referenciados simbolicamente em outros módulos.

Sendo assim, a linguagem simbólica não possui somente os mnemônicos das instruções da MVN, mas também comandos chamados de pseudo-instruções da linguagem de montagem. Na linguagem de montagem, as pseudo-instruções também são representadas por mnemônicos, listados abaixo:

- @ : Origem Absoluta. Recebe um operando numérico, define o endereço da instrução seguinte;
- K : Constante, o operando numérico tem o valor da constante (em hexadecimal). Define uma área preenchida por uma CONSTANTE de 2 bytes;
- $: Reserva de área de dados, o operando numérico define o tamanho da área a ser reservada. Define um BLOCO DE MEMóRIA com número especificado de words;
- # : Final físico do texto fonte;
- & : Origem relocável;
- > : Endereço simbólico de entrada (entry point). Define um endereço simbólico local como entry-point do programa;
- < : Endereço simbólico externo (external). Define um endereço simbólico que referencia um entry-point externo.

Na figura 6, temos um exemplo de um somador escrito em linguagem de montagem, visto na aula de Fundamentos de Eng. de Computação, e sua respectiva tradução pelos módulos Montador, *Linker* e Relocador, módulos extras porém integrados no nosso caso:

Figura 6 – Exemplo de um somador



	Endereço de geração	Resolução do operando	Relocabilidade do operando	Localidade do operando
SOMADOR <		1	?	1
ENTRADA1 <		1	?	1
ENTRADA2 <		1	?	1
SAIDA >		0	0	1
@ /0000				
JP INICIO	0	0	0	0
VALOR1 K =50	0	0	0	0
VALOR2 K #101101	0	0	0	0
SAIDA K /0000	0	0	0	0
INICIO LD VALOR1	0	0	0	0
MM ENTRADA1	0	1	?	1
LD VALOR2	0	0	0	0
MM ENTRADA2	0	1	?	1
SC SOMADOR	0	1	?	1
HM /00	0	0	0	0

```
5000 0000 ; "SOMADOR<"
5001 0000 ; "ENTRADA1<"
5002 0000 ; "ENTRADA2<"
1006 0000 ; "SAIDA>"

0000 0008
0002 0032
0004 002d
0006 0000
0008 8002
500a 9001
000c 8004
500e 9002
5010 a000
0012 c000
```

4 Características Gerais

4.1 Organização da memória

O ambiente de execução da MVN fornece aos programadores um tamanho limitado de memória para ser usado no geral, a ser compartilhado entre o código e as variáveis do programa. O montador aloca a memória com base nos endereços relativos especificados no código do programa. Do total, a parte inicial da memória é reservada para guardar as instruções que serão executadas pelo programa. A parte final da memória deve ser usada especialmente para o uso do registro de ativação.

De maneira mais objetiva, reserva-se uma parte do código para a área de dados, uma parte para a função principal e as subrotinas e uma parte dedicada a pilhas de variáveis e endereços que viabilizam a chamada de subrotinas.

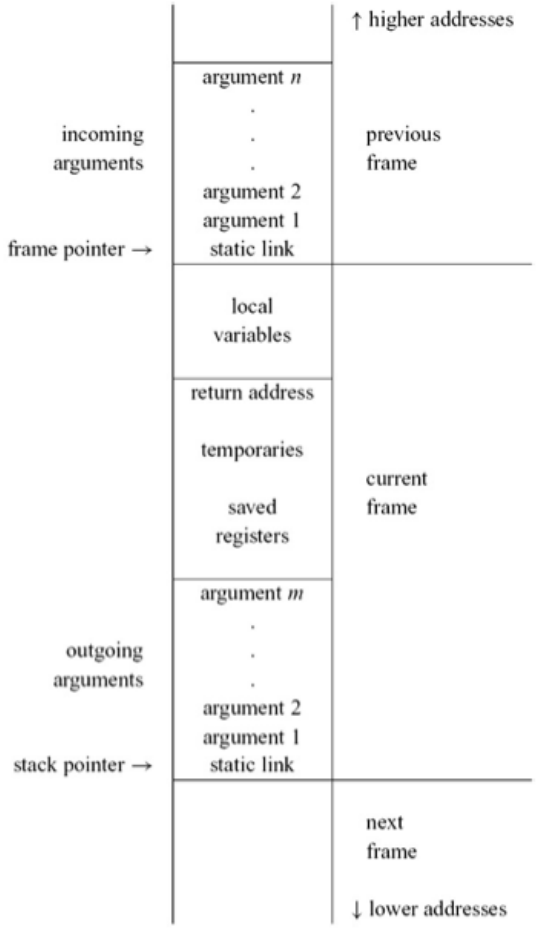
4.2 Registro de ativação

As funções em programas têm variáveis locais, que devem ser criadas na chamada da função e sobrevivem até que a função retorne. Elas também possuem recursão, onde cada instância da função tem seus próprios parâmetros e locais. As chamadas de funções se comportam de maneira LIFO, portanto podemos usar uma pilha como estrutura.

As operações push e pop dessa pilha não podem ser feitas individualmente para cada variável. Desa forma, manipula-se conjuntos de variáveis, e precisamos ter acesso a todas elas. Com isso, definimos dois conceitos:

- *Stack Pointer* (SP):
  - Todas as posições além do SP são lixo;
  - Todas as anteriores estão alocadas.
- *Activation Record* ou *Stack Frame*
  - área na pilha reservada para os dados de uma função (parâmetros, locais, endereço de retorno, etc).
  - esta parte da pilha foi fusionada à parte anterior, facilitando o uso da pilha e diminuindo a quantidade de dados na mesma. Esta decisão não afeta a implantação, uma vez que no caso desta linguagem o compilador tem total controle do tamanho das estruturas sendo utilizadas.

Figura 7 – Esquema do Registro de Ativação



A figura 7 ilustra a organização da pilha. O uso do registro de ativação permite entre outras coisas a chamada recursiva de funções, uma vez isso não é possível de forma nativa no ambiente da MVN. No caso da MVN, a pilha cresce para baixo e as subrotinas são executadas utilizando as seguintes instruções:

- Desvio para subprograma - mnemônico SC (0xA): armazena o endereço de instrução seguinte (atual + 1) na posição de memória apontada pelo operando. Em seguida, desvia a execução para o endereço indicado pelo operando e acrescido de uma unidade.
- Retorno de subprograma - mnemônico RS (0xB): desvia a execução para o endereço indicado pelo valor guardado na posição de memória do operando.

Foi criada por nós uma biblioteca em assembly para implementar funções auxiliares de entrada e saída de dados, além da funcionalidade de empilhar, desempilhar e ter acesso a informações contidas na pilha discutida anteriormente. Essas funções são explicadas na próxima seção.

5 Biblioteca Desenvolvida em Assembly

A biblioteca padrão desenvolvida é dividida em dois módulos o primeiro implementa as funções básicas de empilhamento, sendo chamado de `std.asm` O segundo módulo implementa as operações de input e output de dados, de nome `stdio.asm`.

5.1 STD

A manipulação de pilhas é feita pela biblioteca padrão, sendo que deseja-se seguir a estrutura abaixo definida, facilitando o uso e acesso das variáveis. Vemos abaixo um exemplo do uso da biblioteca. Na linha 23 salvamos uma variável recebida por parâmetro na pilha e na linha 30 recuperamos seu valor.

Logo antes de retornar devemos executar a função `POP_CALL` ela é responsável por escrever o endereço de retorno na função em que estamos, assim aproveitando das chamadas existentes na `MVN` (funções devem ser *stateless* para tanto). Percebe-se que é possível executar a função recursivamente (linha 55 e 57). Para tanto é necessário chamar a função `PUSH_CALL` para que a mesma efetue o empilhamento e escreva o endereço de retorno atual na pilha.

```
1  ;; VARIAVEIS GLOBAIS
2  ;; comeco da pilha = FFF
3  ;; tamanho da pilha = 2FF
4  ;;    | ptr to old_stack_head  | \____ STACK_PTR
5  ;;    |      savedregist       |
6  ;;    |         ...            |
7  ;;    |      local var         |
8  ;;    |         ...            |
9  ;;    |      temporaries       |
10 ;;    |      parameters        |
11 ;;    |         ...            |
12 ;;    |     ref parameters     |  _____ OLD STACK_PTR
13 ;;    |      returnaddrs       | /      (STACK_PTR points here)
14
15 EXAMPLE_STACK_ARG    K  /0000
16 EXAMPLE_STACK        JP /000
17                      SC PRINT_STACK_ADDRS   ;; deve imprimir 0fff
18                      ;;; SALVAR ARGUMENTOS na pilha
19                      LV =0
20                      MM WORD_TO_SAVE
21                      LV EXAMPLE_STACK_ARG
22                      MM ORIGIN_PTR
23                      SC SAVE_WORD_TO_LOCAL_VAR
```

```
24                          ;;;; CORPO DA FUNCAO
25                          ;;; CARREGANDO UM VALOR DA PILHA
26                          LV =0
27                          MM WORD_TO_GET
28                          LV EXAMPLE_STACK_ARG
29                          MM STORE_PTR
30                          SC GET_WORD_LOCAL_VAR
31                          ;;; IMPRIME
32                          LV COUNT_IS
33                          MM STRING_PTR
34                          SC P_STRING  ;; inline fct, no need to stack
35                          LD EXAMPLE_STACK_ARG
36                          MM TO_BE_PRINTED
37                          SC P_INT_ZERO
38                          SC P_LINE
39
40                          LD EXAMPLE_STACK_ARG
41                          JZ RETURN_EXAMPLE_STACK
42
43                          LD EXAMPLE_STACK_ARG
44                          - ONE
45                          MM EXAMPLE_STACK_ARG
46
47                          LV =1
48                          MM PUSH_CALL_SIZELV
49                          LV =0
50                          MM PUSH_CALL_RET_ADDRS
51                          LV =0
52                          MM PUSH_CALL_TMP_SZ
53                          LV =0
54                          MM PUSH_CALL_PAR_SZ
55                          SC PUSH_CALL
56
57                          SC EXAMPLE_STACK  ;; chamada recursiva
58                          ;;;; FIM DO CORPO DA FUNCAO
59 RETURN_EXAMPLE_STACK LV EXAMPLE_STACK
60                          MM POP_CALL_FCT
61                          SC POP_CALL ;; trickery!
62
63                          SC PRINT_STACK_ADDRS   ;; deve imprimir 0fff
64                          RS EXAMPLE_STACK
```

Abaixo podemos ver a implementação das funções de `PUSH` e `POP`

A pilha é implementada dos valores mais altos da memória para os valores mais baixos, sendo assim, o ponteiro de pilha começa apontando para `0x0FFF`.

A pilha funciona como uma lista ligada que guarda o endereço da última célula da

pilha. Sendo assim, a operação de `POP` é trivial. Estas funções fazem a gestão do endereço de retorno automaticamente, contanto que se siga a premissa de chamada (chamada da função logo após a chamada de `PUSH_CALL` e seus parâmetros).

```
1  ;; *** PUSH_CALL ***
2  PUSH_CALL            JP /000
3                       LD PUSH_CALL  ;; get return addrs
4                       + TWO ;; return address of the callee
5                       + LOADV_CONST
6                       MM LOAD_RETURN_ADDRS
7                       LD STACK_PTR
8                       - TWO         ;; new return addrs
9                       + MOVE_CONST
10                      MM MOVE_RETURN_ADDRS
11 LOAD_RETURN_ADDRS    JP /000
12 MOVE_RETURN_ADDRS    JP /000    ;; return addrs salvo
13                      LD STACK_PTR
14                      - TWO
15                      - TWO
16                      - PUSH_CALL_SIZELV
17                      - PUSH_CALL_RET_ADDRS
18                      - PUSH_CALL_TMP_SZ
19                      - PUSH_CALL_PAR_SZ
20                      - TWO  ;; return addrs
21                      MM TMP_1
22                      LD TMP_1
23                      + MOVE_CONST
24                      MM MRKR_PC_SAVE_HEAD
25                      LD STACK_PTR
26 MRKR_PC_SAVE_HEAD    JP /000
27                      LD TMP_1
28                      MM STACK_PTR
29                      RS PUSH_CALL
30 ;; **** POP_CALL ****
31
32 POP_CALL_FCT         K /0000
33 POP_CALL             JP /000 ; retorno
34 POP_CALL_INIT        LD STACK_PTR
35                      + LOAD_CONST
36                      MM MRKR_PC_LOAD_HEAD
37 MRKR_PC_LOAD_HEAD    JP /000
38                      MM STACK_PTR
39                      LD STACK_PTR
40                      - TWO
41                      + LOAD_CONST
42                      MM LOAD_RETURN_ADDRS_2
43                      LD POP_CALL_FCT
44                      + MOVE_CONST
```

```
45                          MM MOVE_RETURN_ADDRS_2
46 LOAD_RETURN_ADDRS_2 JP /000
47 MOVE_RETURN_ADDRS_2 JP /000   ;; engana a funcao para ela pensar que ela
48                               ;; tem que retornar para esse valor
49                          RS POP_CALL
```

As rotinas de salvaguarda e carregamento dos valores locais, parâmetros, referências pode ser feita por meio das chamadas abaixo, `SAVE_WORD_TO_LOCAL_VAR` e `GET_WORD_LOCAL_VAR` respectivamente.

```
1  ;; **** SAVE_WORD_TO_LOCAL_VAR WORD_TO_SAVE ORIGIN_PTR ****
2  SAVE_WORD_TO_LOCAL_VAR       JP /000
3                               LD STACK_PTR
4                               + TWO           ;; first word
5                               + WORD_TO_SAVE
6                               + WORD_TO_SAVE  ;; WORD_TO_GET * 2
7                               + MOVE_CONST    ;;
8                               MM MOVE_WORD_LOCAL_VAR_2
9                               LD ORIGIN_PTR
10                              + LOAD_CONST
11                              MM LOAD_WORD_LOCAL_VAR_2
12 LOAD_WORD_LOCAL_VAR_2        JP /000 ;; 8FROMPTR
13 MOVE_WORD_LOCAL_VAR_2        JP /000 ;; 9TOPTR
14                              RS SAVE_WORD_TO_LOCAL_VAR
15
16 ;; **** GET_WORD_LOCAL_VAR WORD_TO_GET STORE_PTR ****
17
18
19 WORD_TO_GET          K /000
20 STORE_PTR            K /000
21
22 GET_WORD_LOCAL_VAR           JP /000
23                              LD STACK_PTR
24                              + TWO           ;; first word
25                              + WORD_TO_GET
26                              + WORD_TO_GET   ;; WORD_TO_GET * 2
27                              + LOAD_CONST    ;;
28                              MM LOAD_WORD_LOCAL_VAR
29                              LD STORE_PTR
30                              + MOVE_CONST
31                              MM MOVE_WORD_LOCAL_VAR
32 LOAD_WORD_LOCAL_VAR          JP /000 ;; 8FROMPTR
33 MOVE_WORD_LOCAL_VAR          JP /000 ;; 9TOPTR
34                              RS GET_WORD_LOCAL_VAR
```

5.2 STDIO

O ambiente de execução também é provido de funções de input/output:

Para a impressão de *strings* podemos utilizar a função `P_STRING`, passando o ponteiro para o começo de uma `string`. Em `CZAR` consideramos *strings* como sendo *bytes* em um vetor de *word* terminados pelo *byte* `0x0000`. Vale salientar que esta forma de armazenamento não causa problemas com outros tipos de armazenamento mais compactos, como a utilização dos dois *bytes* da *word* para armazenamento de *chars* subsequentes. Quando a função recebe a *word* `0x0030`, primeiramente ela vai imprimir `0x00` que é o caractere nulo, portanto, sem impressão e então imprimir o caractere correspondente a `0x30`.

```
1  ;; ****  P_STRING &STRING_PTR ****
2  ;;    Imprime a string apontada por STRING_PTR ate
3  ;; o caractere /000
4
5  P_STRING              JP /000            ; endereco de retorno
6  PSTRINGINIT           LD STRING_PTR
7                        MM TO_BE_PRINTED_TMP
8  LOAD_TO_BE_PRINTED    LD TO_BE_PRINTED_TMP
9                        + LOAD_CONST
10                       MM LABELLOAD
11 LABELLOAD             K /0000
12                       JZ P_STRING_END  ; se zero vamos para o final!
13                       PD /100
14                       LD TO_BE_PRINTED_TMP
15                       + TWO
16                       MM TO_BE_PRINTED_TMP
17                       JP LOAD_TO_BE_PRINTED
18 P_STRING_END          RS P_STRING
```

Para a leitura de *strings* seguimos o padrão definido anteriormente, um *byte* (*char*) por *word*:

```
1  ;; *** GETS STORE_PTR_IO ***
2  ;; Existe um problema de buffer aqui... nao vamos
3  ;; trata-lo, pois este e' um problema intri'nseco da
4  ;; MVN. (leitura e subsequente bloqueio por word)
5  LAST_CONTROL_CHAR_P_ONE    K /0021
6  ARRAY_POS_BYTE  JP /000
7  GETS            JP /000
8                  LD STORE_PTR_IO
9                  MM ARRAY_POS_BYTE
10 GETS_LOOP       GD /000
11                 MM HIGH_V
12                 SC HIGH_LOW
13                 LD HIGH_V
14                 - LAST_CONTROL_CHAR_P_ONE
```

```
15                      JN RETURN_GETS
16                      LD ARRAY_POS_BYTE
17                      +  MOVE_CONST
18                      MM MOVE_HIGH_V
19                      LD HIGH_V
20 MOVE_HIGH_V          JP /000
21
22                      LD ARRAY_POS_BYTE
23                      +  TWO
24                      MM ARRAY_POS_BYTE
25
26                      LD LOW_V
27                      -  LAST_CONTROL_CHAR_P_ONE
28                      JN RETURN_GETS
29                      LD ARRAY_POS_BYTE
30                      +  MOVE_CONST
31                      MM MOVE_LOW_V
32                      LD LOW_V
33 MOVE_LOW_V           JP /000
34
35                      LD ARRAY_POS_BYTE
36                      +  TWO
37                      MM ARRAY_POS_BYTE
38
39                      JP GETS_LOOP
40
41 RETURN_GETS          LD ARRAY_POS_BYTE
42                      +  MOVE_CONST
43                      MM MOVE_ZERO
44                      LV =000
45 MOVE_ZERO            JP /000
46
47                      LD ARRAY_POS_BYTE
48                      +  TWO
49                      MM ARRAY_POS_BYTE
50                      RS GETS
```

A biblioteca também é capaz de realizar a leitura e escrita de valores inteiros (funções auxiliares estão disponíveis no pacote em anexo):

```
1 ;; *** READ_INT STORE_PTR_IO ***
2 ;; doesnt care about buffers, should have a trailing char at the end of the
3 ;; stream otherwise it will just discard it..
4 STORE_PTR_IO      JP /000
5 ZERO_M_ONE        K  /002F
6 NINE_P_ONE        K  /0039
7
8 LOW               K  /0000
```

```
 9 HIGH                K   /0000
10 GO_IF_NUMBER        K   /0000
11 TO_BE_TRIMMED       K   /0000
12 TBT_TMP             K   /0000
13
14 TRIM_INT            JP  /000
15                     LD  TO_BE_TRIMMED
16                     /   SHIFT_BYTE
17                     *   SHIFT_BYTE
18                     MM  TBT_TMP
19                     LD  TO_BE_TRIMMED
20                     -   TBT_TMP
21                     MM  TO_BE_TRIMMED
22                     RS  TRIM_INT
23
24 READ_INT_WORD       JP  /000
25                     GD  /000
26                     MM  TMP_3
27                     LD  TMP_3
28                     /   SHIFT_BYTE
29                     MM  TO_BE_TRIMMED
30                     SC  TRIM_INT
31                     LD  TO_BE_TRIMMED
32                     MM  HIGH
33
34                     LD  TMP_3
35                     MM  TO_BE_TRIMMED
36                     SC  TRIM_INT
37                     LD  TO_BE_TRIMMED
38                     MM  LOW
39                     RS  READ_INT_WORD
40
41 READ_INT            JP  /000
42                     LV  =0
43                     MM  TMP_4
44 READ_INT_LOOP       SC  READ_INT_WORD
45                     LD  HIGH
46                     MM  TMP_3
47                     LV  CONT1
48                     MM  GO_IF_NUMBER
49                     JP  IF_NUMBER_CONTINUE
50 CONT1               LD  LOW
51                     MM  TMP_3
52                     LV  READ_INT_LOOP
53                     MM  GO_IF_NUMBER
54                     JP  IF_NUMBER_CONTINUE
55 NOT_NUMBER          LD  STORE_PTR_IO
```

```
56                          +   MOVE_CONST
57                          MM MOVE_READ_INT
58                          LD TMP_4
59 MOVE_READ_INT            JP /000
60                          RS READ_INT
61
62 IF_NUMBER_CONTINUE       LD TMP_3
63                          -   ZERO_M_ONE
64                          JN NOT_NUMBER
65                          LD NINE_P_ONE
66                          -   TMP_3
67                          JN NOT_NUMBER
68
69                          LD TMP_4
70                          *   TEN
71                          MM TMP_4
72
73
74                          LD TMP_3
75                          -   ZERO_M_ONE
76                          -   ONE
77                          +   TMP_4
78                          MM TMP_4
79
80                          LD GO_IF_NUMBER
81                          MM END_READ_INT
82 END_READ_INT             JP /000
```

A impressão de inteiros, por ser crítica e muito importante para a correção de erros, foi feita de forma simples e direta. Sem laços (unwind de `GOTO` explícito) ou complicações, resultando em uma função bem determinada e robusta.

```
1  ;; *** P_INT_ZERO TO_BE_PRINTED ***
2  ;;  Imprime um inteiro (com zeros a esquerda)
3  ;; ex:
4  ;;  INT_2 K  =345
5  ;;          LD INT_2
6  ;;          MM TO_BE_PRINTED
7  ;;          SC P_INT_ZERO
8  ;; imprime 00345
9  ;;
10 ;;
11 ;; Esta funcao esta com o loop inline
12 ;; sendo simples e robusta
13
14 P_INT_ZERO            JP /000
15 P_INT_INIT            JP P_INT_REAL_INIT
16 ZERO_BASE             K /30
```

```
17 ;; bases para a conversao:
18 INT_POT_1          K =10000
19 INT_POT_2          K =1000
20 INT_POT_3          K =100
21 INT_POT_4          K =10
22 INT_POT_5          K =1
23 P_INT_REAL_INIT    LD TO_BE_PRINTED      ;; PRIMEIRO CHAR
24                    MM TMP_1
25                    /  INT_POT_1
26                    +  ZERO_BASE
27                    PD /100                    ;; imprime
28                    LD TMP_1
29                    /  INT_POT_1
30                    *  INT_POT_1
31                    MM TMP_2
32                    LD TMP_1
33                    -  TMP_2
34                    MM TMP_1
35                    /  INT_POT_2               ;; segundo char
36                    +  ZERO_BASE
37                    PD /100                    ;; imprime
38                    LD TMP_1
39                    /  INT_POT_2
40                    *  INT_POT_2
41                    MM TMP_2
42                    LD TMP_1
43                    -  TMP_2
44                    MM TMP_1
45                    /  INT_POT_3               ;; terceiro char
46                    +  ZERO_BASE
47                    PD /100                    ;; imprime
48                    LD TMP_1
49                    /  INT_POT_3
50                    *  INT_POT_3
51                    MM TMP_2
52                    LD TMP_1
53                    -  TMP_2
54                    MM TMP_1
55                    /  INT_POT_4               ;; quarto char
56                    +  ZERO_BASE
57                    PD /100                    ;; imprime
58                    LD TMP_1
59                    /  INT_POT_4
60                    *  INT_POT_4
61                    MM TMP_2
62                    LD TMP_1
63                    -  TMP_2
```

```
64                  MM TMP_1
65                  /  INT_POT_5            ;; quinto char
66                  +  ZERO_BASE
67                  PD /100                 ;; imprime
68                  LD TMP_1
69                  /  INT_POT_5
70                  *  INT_POT_5
71                  MM TMP_2
72                  LD TMP_1
73                  -  TMP_2
74                  MM TMP_1
75                  RS P_INT_ZERO
```

6 Exemplo de Execução

Foi escrito um script em BASH para facilitar a compilação e execução da MVN:

```
1  # mvnrc
2  #
3  # Author: gpg
4  #
5  # Usage:
6  #  $ source ./mvnfunctions.sh
7  #  $ makelib libraries.asm
8  #  $ makemain mainfile.asm
9  #    be happy :D
10
11 JAVARUN="java -cp"
12 MVNDL="java -jar MvnPcs4_wDumperLoader.jar"
13 MLR="$JAVARUN PCS2302_MLR.jar"
14 MONTA="$MLR montador.MvnAsm"
15 LINKA="$MLR linker.MvnLinker"
16 RELOCA="$MLR relocator.MvnRelocator"
17
18 MVNFILES=mvnfiles
19 # Usage:
20 # makelib lib.asm
21 function makelib () {
22     $MONTA $1 && {
23         echo `basename $1 .asm`.mvn >> $MVNFILES
24         return 0
25     }
26     return 1
27 }
28
29 function prog_size() {
30     lines=$(cat $1 | wc -l)
31     echo $lines*2 | bc 1>&2
32     echo $lines*2+100 | bc
33 }
34
35 function clean_reloca() {
36     rm -f relocados.list
37 }
38
39 function reloca_var() {
40     OUTNAME=`basename $1 .mvn`_relocado.mvn
41     OUTSIZE=$(prog_size $1)
```

```
42      echo RELOCANDO $1, comeco: $2, tamanho: $OUTSIZE 1>&2
43      START=$(printf "%X\n" $2)
44      $RELOCA $1 $OUTNAME $2 1>&2 || return 1
45      cat $OUTNAME 1>&2
46      echo $OUTNAME >> relocados.list
47      echo $OUTSIZE | bc
48  }
49
50  function makemain () {
51      OUTNAME=`basename $1 .asm`.out
52      MAIN=`basename $1 .asm`.mvn
53      BINARY=`basename $1 .asm`
54
55      TMPNAME=tmpfile.tmp
56
57      # make main or die
58      $MONTA $1 || {
59          return 1
60      }
61      {
62          clean_reloca
63
64          SIZEREL=$(prog_size $MAIN) || return 1
65          BUTTER=$(cat $MVNFILES | sort -u)
66          echo Ligando as bagaca sizerel is $SIZEREL
67          $LINKA $MAIN $BUTTER -s $OUTNAME && {
68              echo Jogando tudo pra baixo da main
69              $RELOCA $OUTNAME $BINARY $SIZEREL && {
70                  echo "All Ok!"
71                  echo -e '\E[37;44m'"\033[1mYour binary is called: $BINARY
                        \033[0m"
72                  borala
73              } || {
74                  echo Error
75                  return 0
76              }
77          } || {
78              echo Error
79              return 0
80          }
81      }
82  }
83
84  function cleanlib () {
85      rm -f $MVNFILES
86  }
87
```

```
88 function clean_all_mvn () {
89     rm -f *.dump
90     rm -f *.lst
91     rm -f *.mvn
92 }
93
94 function borala () {
95     rlwrap $MVNDL
96 }
```

6.1 Exemplo de chamada recursiva

Temos aqui um exemplo de leitura e chamada recursiva:

O programa pede por um número na linha 76, uma vez digitado (deve-se utilizar a tecla ENTER duas vezes devido ao problema de *buffer* citado no capítulo anterior) o programa carrega o valor em uma variável local e passa a mesma para uma função que reduz o valor indicado por uma unidade e chama ela mesma até que o valor se resume a zero, neste momento a função retorna e toda a pilha é desalocada.

```
1
2 gpg@rancheiro: --( ~/Poli/PCS2056-Compiladores/github/compiladores/MVN )
3 $ . mvnfunctions.sh
4 gpg@rancheiro: --( ~/Poli/PCS2056-Compiladores/github/compiladores/MVN )
5 $ makelib std.asm
6 ================================================================================

7                 PCS2302/PCS2024  Montador da Maquina de Von Neumann
8                    Versao 1.1 (a)2010  Todos os direitos reservados
9
10 Montador finalizou corretamente, arquivos gerados.
11 gpg@rancheiro: --( ~/Poli/PCS2056-Compiladores/github/compiladores/MVN )
12 $ makelib stdio.asm
13 ================================================================================

14                PCS2302/PCS2024  Montador da Maquina de Von Neumann
15                   Versao 1.1 (a)2010  Todos os direitos reservados
16
17 Montador finalizou corretamente, arquivos gerados.
18 gpg@rancheiro: --( ~/Poli/PCS2056-Compiladores/github/compiladores/MVN )
19 $ makemain usestd.asm
20 ================================================================================

21                PCS2302/PCS2024  Montador da Maquina de Von Neumann
22                   Versao 1.1 (a)2010  Todos os direitos reservados
23
```

```
24 Montador finalizou corretamente, arquivos gerados.
25 352
26 Ligando as bagaca sizerel is 452
27 Arquivo gerado com sucesso.
28 Jogando tudo pra baixo da main
29 Arquivo gerado com sucesso.
30 All Ok!
31 Your binary is called: usestd
32 Inicializacao padrao de dispositivos
33 MVN Inicializada
34
35                 Escola Politecnica da Universidade de Sao Paulo
36             PCS2302/PCS2024 - Simulador da Maquina de von Neumann
37          MVN versao 4.2 (Novembro/2010) - Todos os direitos reservados
38
39  COMANDO  PARAMETROS            OPERACAO
40 ------------------------------------------------------------------------------
41      i                          Re-inicializa MVN
42      p      [arq]               Carrega programa para a memoria
43      r      [addr] [regs]       Executa programa
44      b                          Ativa/Desativa modo Debug
45      l                          Loader
46      d                          Dumper
47      s                          Manipula dispositivos de I/O
48      g                          Lista conteudo dos registradores
49      m      [ini] [fim] [arq]   Lista conteudo da memoria
50      h                          Ajuda
51      x                          Finaliza MVN e terminal
52
53 > p usestd
54 Programa usestd carregado
55
56 > r 000
57 Exibir valores dos registradores a cada passo do ciclo FDE (s/n)[s]: n
58 please insert a string:
59 Poli_USP!!!
60 Poli_USP!!!!"#Teste$
61
62 00000
63 00001
64 00002
65 00003
66 00004
67 00005
68 00006
69 00007
70 00008
```

```
71 00009
72 00010
73 00011
74
75 Stack should be:04095
76 Please write a number, we will count recursively until it gets to zero:
77 23
78
79 Stack was: 04088
80 Counter is:00023
81 Stack was: 04081
82 Counter is:00022
83 Stack was: 04074
84 Counter is:00021
85 Stack was: 04067
86 Counter is:00020
87 Stack was: 04060
88 Counter is:00019
89 Stack was: 04053
90 Counter is:00018
91 Stack was: 04046
92 Counter is:00017
93 Stack was: 04039
94 Counter is:00016
95 Stack was: 04032
96 Counter is:00015
97 Stack was: 04025
98 Counter is:00014
99 Stack was: 04018
100 Counter is:00013
101 Stack was: 04011
102 Counter is:00012
103 Stack was: 04004
104 Counter is:00011
105 Stack was: 03997
106 Counter is:00010
107 Stack was: 03990
108 Counter is:00009
109 Stack was: 03983
110 Counter is:00008
111 Stack was: 03976
112 Counter is:00007
113 Stack was: 03969
114 Counter is:00006
115 Stack was: 03962
116 Counter is:00005
117 Stack was: 03955
```

```
118 Counter is:00004
119 Stack was: 03948
120 Counter is:00003
121 Stack was: 03941
122 Counter is:00002
123 Stack was: 03934
124 Counter is:00001
125 Stack was: 03927
126 Counter is:00000
127 Stack was: 03934
128 Stack was: 03941
129 Stack was: 03948
130 Stack was: 03955
131 Stack was: 03962
132 Stack was: 03969
133 Stack was: 03976
134 Stack was: 03983
135 Stack was: 03990
136 Stack was: 03997
137 Stack was: 04004
138 Stack was: 04011
139 Stack was: 04018
140 Stack was: 04025
141 Stack was: 04032
142 Stack was: 04039
143 Stack was: 04046
144 Stack was: 04053
145 Stack was: 04060
146 Stack was: 04067
147 Stack was: 04074
148 Stack was: 04081
149 Stack was: 04088
150 Stack was: 04095
151 Stack should be:04095
```