Insper

Elementos de Sistemas - Projeto I - Ferramental - Linguagem VM

Rafael Corsi - rafael.corsi@insper.edu.br

Maio, 2018

TODO de inicialização: Scrum Master:

1. Atualizar arquivo ScramMaster.txt
2. Atualizar repositório com o upstram
3. Adicionar o novo script de teste ao travis : I-VM/scripts/testeVM.py
4. Criar projeto no github
5. Atribuir tarefas e acompanhar o desenvolvimento

Esse projeto tem como objetivo introduzir a linguagem VM de pilha utilizadado no Z01.

Projeto I - VM

Entendendo a Organização do Projeto

A pasta do projeto H no repositório Z01, possui a seguinte estrutura:

1. scripts: Scripts em python que automatizam a execução dos testes;
   - compileVM : Compila os .vm gerando . nasm e salva o resultado em bin/nasm/
   - compileNASM : Compila os .nasm gerando .hack e .mif e salva o resultado em bin/hack/
   - testeVM : Executa os códigos .vm no Z01 a fim de verificar os resultados (compara com os testes).
2. src/vm/*.vm: Arquivos VM que serão implementados pelo grupo;
3. src/vmExamples/*.vm: Arquivos VM de exemplo
4. bin/hack/ : Arquivos .hack convertidos via Z01-Assembler.jar
5. bin/nasm/ : Arquivos .nasm convertidos via Z01-VMTranslator.jar
6. tests/tst/*: Arquivos que realizam o teste nos arquivos códigos nasm.

Testes

Os testes verificam regiões de memórias específicas, deve-se seguir os comentários no próprio código a fim de salvar o resultado no local correto.

Linguagem VM

A linguagem VM proposta para o curso é baseada em pilha (assim como tantas outras), as operações nesse nível não mais lidam com registradores do computador mas sim com dados que são colocados e tirados de uma pilha (stack). Uma grande vantagem disso é a abstração do hardware, agora não precisamos mais nos preocuparmos com a manipulação dos dados em baixo nível (o VMTranslator será encarregado disso). Um código escrito em VM passa pelas seguintes etapas antes de ser executado em máquina :

```
      VMTranslator           Assembler
 .vm ------------> .nasm ------------> .hack
```

O código vm é traduzido para linguagem nasm pelo VMTranslator (vocês vão ter que fazer esse programa), e então é montado pelo Assembler para linguagem de máquina.

Temos diversas vantagens quando programamos em linguagem virtual :

a. Abstração de Hardware
   - (já não mais lidamos com o hardware diretamente)
b. Portabilidade
c. Código mais alto nível
   - (chamada de funções, linguagem mais próxima do que estamos acostumados, …)

Pilha

A linguagem VM é baseada em pilha, ou seja, todas as operações que serão realizadas serão feitas na pilha. A pilha é uma região da memória RAM (começando no endereço 256) reservada para armazenar os dados que estão sendo manipulados.

A pilha cresce conforme operações de push (envio de dados para a pilha) vão sendo executados, e decresce conforme operações de pull (retirar dados da pilha) são executadas.

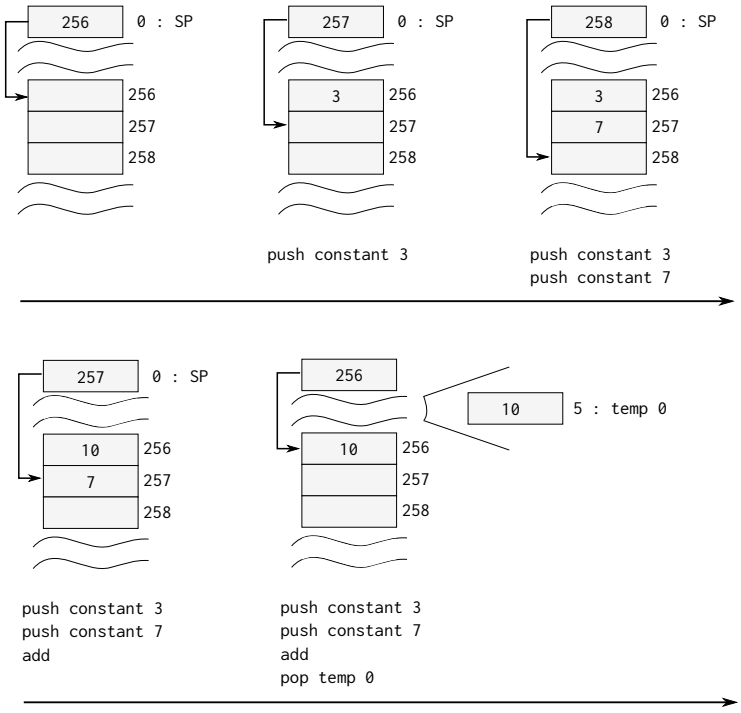


Figure 1: Stack

Stack Pointer (SP)

O Stack Pointer é o endereço de memória (RAM[0]) reservado por apontar o topo da pilha, ou seja, a próxima posição vazia da pilha. O SP é salvo na RAM 0 (R0), e deve ser incrementado/decrementado conforme a pilha vai sendo manipulada.

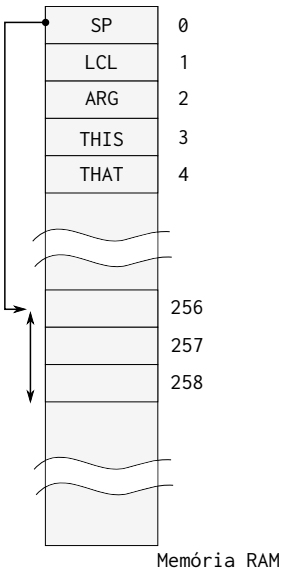


Figure 2: Stack Pointer

Operação

Considerando a seguinte disposição na pilha :

```
      ...
      ...
       X
       Y
SP ->
```

São suportadas as seguintes operações aritméticas na pilha:

- add
  - executa: X + Y
- sub
  - executa: X - Y
- neg
  - executa: $-Y$ (complemento de dois)

- eq
  - compara X == Y
    - True : resulta em b“1111111111111111”, 0xFFFF
    - False: resulta em b“0000000000000000”“, 0x0000
- gt
  - compara X > Y
    - True : resulta em b“1111111111111111”, 0xFFFF
    - False: resulta em b“0000000000000000”“, 0x0000
- lt
  - compara X < Y
    - True : resulta em b“1111111111111111”, 0xFFFF
    - False: resulta em b“0000000000000000”“, 0x0000
- and
  - executa: X and Y (bit a bit)
- or
  - executa: X or Y (bit a bit)
- not
  - executa: not Y (bit a bit)

Note que as operações de comparação (eq, gt, lt) resulta em um True e False e esse resultado é salvo na pilha, considere o exemplo a seguir (em hexa) que possui inicialmente na pilha os valores 2 3 e 5, e após a operação de eq os valores 3 e 5 são comparados e resulta em um valor True ou False (0xFFFF ou 0x0000).

```
          -> eq ->      -> gt ->
     0x2            0x2          0xFFFF
     0x3            0x0     SP->
     0x5       SP->
SP->
```

As operações na pilha não apagam o resultado que já estava na pilha, se olharmos a memória real do exemplo anterior seria a seguinte :

```
          -> eq ->      -> gt ->
     0x2            0x2          0xFFFF
     0x3            0x0     SP-> 0x0
     0x5       SP-> 0x5          0x5
SP-> 0x0            0x0          0x0
```

O mesmo acontece com arquivos que são deletados do seu computador, o sistema operacional não “limpa a memória” sempre que um arquivo é excluído, apenas apaga o ponteiro para aquele arquivo.

Acesso a memória

Os comandos **push** e **pop** são a única maneira que temos de acessar/manipular a memória. O comando **push** traz para a pilha um valor da memória (RAM

ou ROM) e o comando **pop** salva na memória um valor da pila (RAM). Os comandos possuem a seguinte sintaxe :

- **push** *segment* index*
- **pop** *segment* index*

Onde segment pode ser :

segment	Uso	Possíveis valores	Comentário
argument	Local onde o argumento da função está salvo	0 ..	Alocado dinamicamente pelo VMTranslator quando a função é chamada
local	Local das variáveis locais da função	0 ..	Alocado dinamicamente pelo VMTranslator quando a função é chamada
static	Local onde as variáveis do objeto estão salvos	0 ..	Essas variáveis são compartilhadas por todas as funções do mesmo .vm, assim como em um objeto
constant	Carrega uma constante na pilha	0 .. 32767	Mesmo uso do leaw (carrega da ROM um valor na RAM)
this/that	Segmentos de uso geral, pode apontar para qualquer lugar	0 ..	Usado para ler e escrever de endereços da memória, por exemplo, acessar o LCD
pointer	Altera os valores do this e do that	0, 1	Usado para modificar a onde o this e o that apontam
temp	Local para uso de variáveis temporárias	0 .. 7	Acessado por qualquer função, é armazenado nos endereços R5 .. R12 da RAM

Exemplo, acessando o temp

Por exemplo, para trazermos para a pilha uma constante realizamos a seguinte operação :

```
push constant 15
```

- nesse caso o segmento acessado é o constant e o parâmetro é o 15.

Para salvarmos o valor 15 no temp 3 (endereço da RAM 7), basta :

```
push constant 15
pop temp 3
```

Podemos também trazer o temp 3 para a pilha :

```
push temp 3
```

Escrevendo um pixel no LCD

Para atualizarmos o LCD via VM será necessário primeiro atualizarmos para onde o **that** aponta, **that** é a maneira que possuímos de escrever em qualquer endereço da memória. O exemplo a seguir ilustra como usamos o segmento **that** para escrever nos pixels centrais do LCD, supondo que gostaríamos de realizar a seguinte operação em C.

```
int *pLCD = 16384
*(pLCD + 1200) = 0xFFFF
```

Nesse pequeno código em C o que está acontecendo é que primeiramente definimos um ponteiro pLCD que aponta para 16384, depois fazemos com que o endereço desse ponteiro + 1200 receba 0xFFFF, o mesmo código em VM é realizado da seguinte maneira :

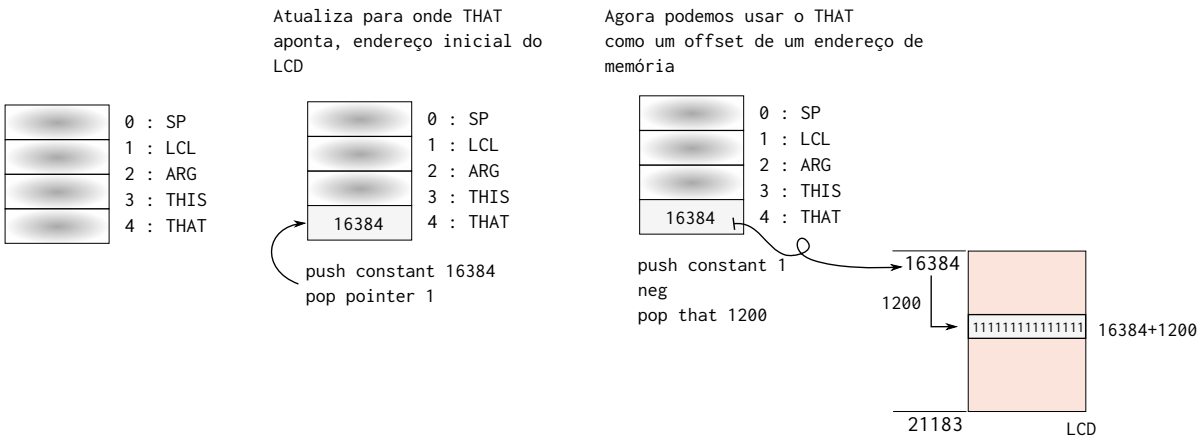


Figure 3: exemplo that atualizando LCD

```
push constant 16384   -- carrega 16384 para a pilha
pop pointer 1         -- atualiza para onde that aponta (int *pLCD = 16384)
push contant 1        -- carrega 1 para a pilha
neg                   -- nega o 1 para obter o valor 0xFFFF
                      -- poderia ter realizado o push contant 4095 no lugar
                      -- dessas duas operações
pop that 1200         -- faz com que o endereço da memória 16384 + 1200 = 0xFFFF
```

GOTO

Goto é a maneira de desviarmos uma execução em .vm, e possui a seguinte sintaxe:

- **goto** LABEL
- **if-goto** LABEL

podemos utilizar dois tipos :

- goto : incondicional , salta sem condição
- if-goto : condiconal, salta se o último valor da pilha for True

Exemplo 1 : Salta para IGUAL se 3 = 2

```
push constant 3
push constant 2
eq
if-goto IGUAL
..
..
label IGUAL
..
..
```

A seguir um exemplo de loop utilizando goto :

```
// for(i=0; i<10; i++)
//     x = x+1;
push constant 0
pop temp 0
push constant 1
pop temp 1
label LOOP_START
  push temp 0
  push constant 10
  eq
  if-goto END       // se temp0 = 10 salta para o fim
  push temp 0
  push constant 1
  add
  pop temp 0
  push temp 1
  push temp 1
  add
  pop temp 1
  goto LOOP_START  // If counter > 0, goto LOOP_START
label END
```

labels

Os labels são definidos pela keyword **label** + nome :

- **label** nome

Funções

A linguagem VM possibilita o uso de funções, as funções são definidas em novos arquivos .vm na mesma pasta do arquivo Main.vm. Por exemplo :

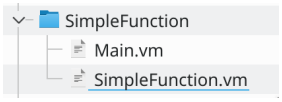


Figure 4: Função

Nesse projeto SimpleFunction possuímos duas funções : O **Main.vm** e a **SimpleFunction.vm**. A função main deve sempre existir no projeto, e será a primeira chamada na inicialização do sistema (assim como no python e C**.

Para definirmos uma função em VM basta criarmos um arquivo com a extensão **.vm** (que precisa ter o mesmo nome da função) que será como uma classe do nosso projeto, podendo conter mais que um método/função.

Olhe o exemplo src/vmExamples/StatiTest/ para ver como isso funciona.

Uma função é definida pela seguinte estrutura :

- **function** *functionName numberOfVars*

Onde :

```
- function : é uma palavra reservado (keyword) para definir funções
- functionName : é o nome da função
- numberOfVars : a quantidade de variáveis locais que essa função possui.
```

Como exemplo, vamos transformar a seguinte função em python para vm :

```python
def SimpleFunction(a, b):
    aux0 = a + b
    aux1 = a - b
    return(aux1+aux0)
```

Em vm:

```
function SimpleFunction 2
    push argument 0
    push argument 1
    add
    pop local 0       // aux0 = a + b
    push argument 0
    push argument 1
```

```
sub
pop local 1       // aux1 = a - b
push local 0
push local 1
add               // aux0 + aux1
return
```

Essa função possui duas variáveis locais, que pode ser acessada pelo segmento **local**, os parâmetros passados para a função (a e b) são acessíveis pelo segmento **argument** :

- **push argument 0**

  `- acessa o primeiro argumento da função ( **a** ), trazendo o dado para a pilha.`

- **push argument 1**

  `- acessa o primeiro argumento da função ( **b** ), trazendo o dado para a pilha.`

- **push/pop temp 0**

  `- acessa ou grava na primeiro variável local da função ( **aux0** ).`

- **push/pop temp 1**

  `- acessa ou grava na primeiro variável local da função ( **aux1** ).`

Note que os parâmetros devem ser apenas leitura, não devendo escrever nesses segmentos.

return

A função considera como retorno o último valor da pilha, e sempre retorna um único

Chamada de função

A chamada de função ocorre na própria pilha, para isso é necessário colocar na pilha os parâmetros da função, no exemplo anterior :

```
     a
     b
SP->
```

e em seguida fazer a chamada de função que possui a seguinte estrutura :

- **call** *functionName numberOfPar*

Onde :

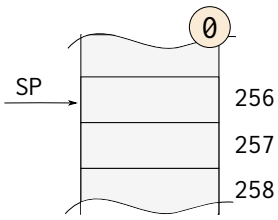
- **call** : palavra reservada para chamada de funções
- **functioName** : nome da função a ser chamada

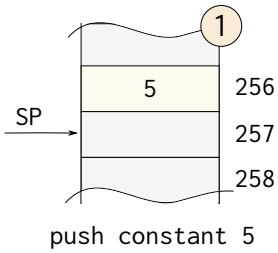
- **numberOfPar** : quantidade de parâmetros que essa função recebe.

O exemplo a seguir chama a função SimpleFunction com os valores 5 e 8

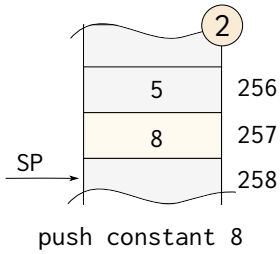
```
function Main.main 0
    push constant 5
    push constant 8
    call SimpleFunction 2
```

function main 0

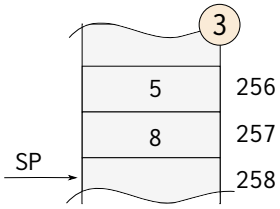




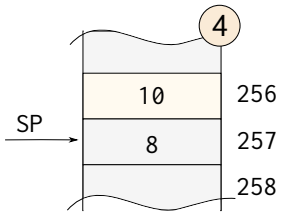
push constant 5



push constant 8

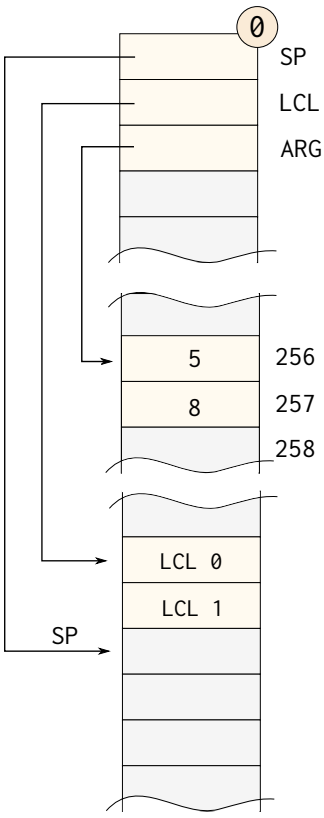


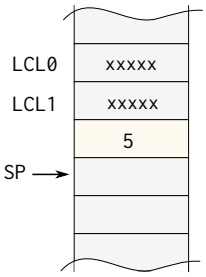
call SimpleFunction 2



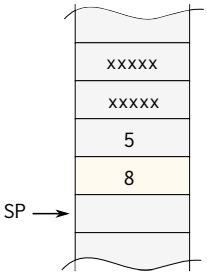
call SimpleFunction 2

function SimpleFunction 2

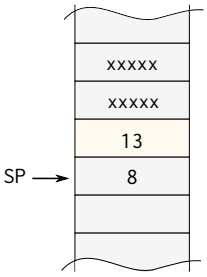




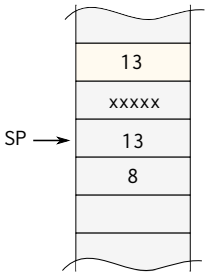
push argument 0



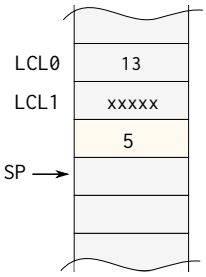
push argument 1



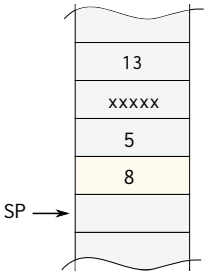
add



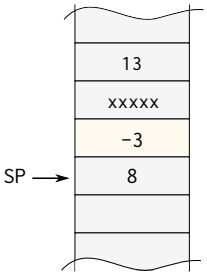
pop local 0



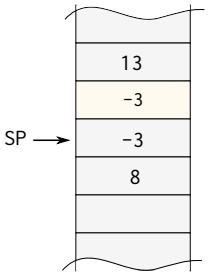
push argument 0



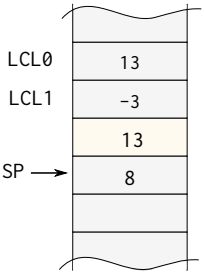
push argument 1



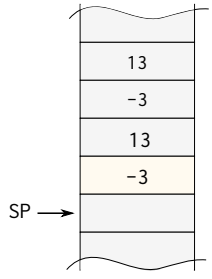
sub



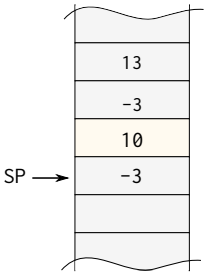
pop local 0



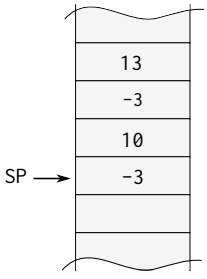
push local 0



push local 1



add



return