Universidade de São Paulo
Escola Politécnica
Engenharia de Computação

Sistemas de Programação
PCS3616

Relatório do Projeto

Alunos:
Fernanda Namie Takemoto Furukita NUSP: 11257501
Victor Hoefling Padula NUSP: 10770051

Professor:
Ricardo Luis de Azevedo da Rocha

São Paulo
Abril de 2021

Universidade de São Paulo
Escola Politécnica
Engenharia de Computação

Sistemas de Programação - Relatório de Projeto

Relatório relativo ao projeto da matéria de Sistemas de Programação do curso de Engenharia de Computação da Universidade de São Paulo, como requisito para conclusão da matéria.

Alunos:
Fernanda Namie Takemoto Furukita
Victor Hoefling Padula
Professor:
Ricardo Luis de Azevedo da Rocha

São Paulo
2021

Conteúdo

1 Resumo

Neste projeto da matéria de Sistemas e Programação, foi selecionada a terceira proposta fornecida, isto é, a proposta do **Ambiente de Execução**. Então, o arquivo ambiente.mvn entregue como solução formulada pela dupla em linguagem de máquina contém um programa que realiza a leitura de um arquivo em formato *asm* resultante da compilação de um outro programa em C - -, linguagem apresentada na segunda proposta de projeto. Assim, em suma, o programa realiza a leitura sequencialmente de um dado arquivo *code.asm*, convertendo a linguagem *asm* para *mvn*, e escrevendo nos respectivos endereços, ou seja, na ordem em que as instruções e constantes são fornecidas. Depois de realizada essa conversão, são executadas as instruções convertidas e o ambiente de execução então para. Caso, durante a leitura do arquivo, seja encontrado algum erro, o ambiente para antes de executar as instruções resultantes da leitura.

2 Resolução

Para a resolução do problema proposto, foi criado um montador em linguagem de máquina. Assim, o programa realiza a leitura do arquivo *asm*, convertendo as instruções para os respectivos valores em hexadecimal e escrevendo-as ordenadamente em um espaço reservado da memória para, depois de escritas todas as instruções, executá-las.

Então, no início ocorre a leitura da primeira linha, que foi determinada como sendo a definição do endereço inicial a partir do uso de @. Por isso, temos uma função para tratar essa leitura. Nela, verifica-se se a linha está no formato esperado (especificado na quarta pergunta da próxima seção) e, em caso positivo, registra o endereço inicial e segue com a leitura. Em caso negativo, ocorre o erro do tipo 1.

Depois, há a chamada de outra função (onde é iniciado o looping de leitura de todas as linhas do arquivo), na qual avalia-se qual o tipo de linha sendo lida. Avalia-se, portanto, se a linha é iniciada com um rótulo ou não, ou se chegou-se na linha final. Para cada caso, a função retorna 1, 0 e -1, respectivamente. Então, ao sair da função, dependendo do valor retornado, é chamada uma função determinada para cada caso.

Para o caso em que há um rótulo antecedendo a instrução, ele é lido e guardado numa tabela criada para cada rótulo. Nela, há dois endereços para guardar o rótulo em si, que possui um tamanho limitado de 4 caracteres, e um endereço para guardar em que endereço do código ele se encontra, para que ele possa ser localizado posteriormente. Depois de reservado o rótulo, o

funcionamento do código segue a lógica do caso sem rótulo, pois depois de lido o rótulo no início da linha, o formato se torna igual ao da linha sem rótulo, com uma instrução e o respectivo parâmetro.

Então, caso não haja um rótulo e não tenha chegado ao fim ou tenha acabado o tratamento do rótulo, é chamada a função sem rótulo. Nela, ocorre a leitura da instrução (JP, JZ, JN, etc.) e então é convertida para o seu respectivo valor em hexadecimal (0000, 1000, 2000, etc.). e guardada em uma variável criada chamada **OPCODE**. Para realizar esse tratamento, foi adaptada a função já utilizada durante o decorrer dos laboratórios da disciplina **MNEM2OP**. Ela foi adaptada de modo a atender os requisitos do projeto, acrescentando o suporte para o caso em que há a declaração de uma constante - K. Além disso, caso a instrução fornecida no arquivo não faça parte do conjunto de instruções reconhecíveis, ocorre o erro do tipo 3.

Depois de lida a instrução, é verificado se o valor que vem depois se trata de uma constante em hexadecimal (antecipada por uma barra) ou de um rótulo. Caso seja uma constante, basta somá-la ao **OPCODE** obtido anteriormente e colocá-la no endereço indicado. Esse endereço toma início no endereço fornecido na primeira linha e é incrementado com 0002 a cada nova linha escrita. Caso se trate de um rótulo, é necessário percorrer a tabela de rótulos para encontrar uma correspondência. Porém, existe também um tratamento para o caso do rótulo não ter sido escrito ainda e estar em uma linha "futura"do código.

Considerando que o rótulo já foi identificado e está na tabela de rótulos, quando encontrada a correspondência, soma-se o endereço em que se encontra esse rótulo registrado com o **OPCODE** e armazena o resultado no endereço respectivo, seguindo a mesma lógica do caso da constante. Entretanto, se não ocorre a correspondência com nenhum rótulo já armazenado, foram criadas novas variáveis para guardar esse rótulo pendente, o seu **OPCODE** e o valor de -1, para indicar que existe um rótulo pendente (quando não há um rótulo pendente, essa variável é zerada). Mas antes de armazenar esses valores, é conferido se já não há um rótulo pendente. Caso exista um rótulo pendente que ainda não foi tratado, resulta em erro. Ou seja, não pode-se ter dois rótulo pendentes seguidos antes da resolução do primeiro.

Então, quando é encontrado um novo rótulo no início de uma linha, é verificado se este último valor é negativo depois de escrevê-lo na tabela de rótulos. Caso sim, verifica-se primeiro se esse novo rótulo é correspondente ao rótulo pendente. Caso não seja, ele apenas continua a rodar.

Depois de encontrada uma correspondência entre o rótulo pendente e o rótulo novo, segue-se o mesmo procedimento. Soma-se **OPCODE** armazenado do rótulo pendente com o endereço desse novo rótulo, colocando o resultado como a nova instrução no endereço respectivo.

Finalmente, caso seja encontrado o fim do arquivo, definido pelo uso da hashtag (#), termina-se a leitura, finalizando o looping da **MAIN**. Depois de finalizado o loop, pula-se para o endereço inicial da região onde o código convertido foi escrito para então executar as instruções em si. Assim, ocorre o fim do programa.

3 Perguntas

3.1 Primeira pergunta

A utilização de memória pode ser mais eficiente? Como? Se sim, por que a forma menos eficiente foi escolhida?

A utilização da memória poderia ser mais eficiente, por exemplo reutilizando uma mesma variável para diferentes propósitos dentro de diferentes funções, pois existem variáveis cujos valores não se interferem dentro de certas funções. Porém, nesse caso, optou-se pela separação do significado de cada variável, de forma que o código seja mais facilmente compreendido, contribuindo para a clareza.

3.2 Segunda pergunta

Os códigos escritos podem ser mais eficientes? Como? Se sim, por que a forma menos eficiente foi escolhida?

O código da função **MNEM2OP** em particular poderia ser bem mais eficiente. Poderia ser implementado um looping de procura na tabela dos mnemônicos, usando um apontador para acessar cada endereço e comparar com o valor lido para a instrução. A partir do endereço em que se encontra o mnemônico correspondente, seria possível também determinar o código em hexadecimal da instrução com uma lógica simples de conversão. No caso do código formulado, a tabela começa no endereço 0x004. Para obter o respectivo código de cada instrução, bastaria subtrair o valor de 0x004 e dividir por 0x002, para depois multiplicar por 1000. Entretanto, a forma menos eficiente foi escolhida, pois foi a solução desenvolvida durante o laboratório e já havia a certeza de que esse método, apesar de considerado menos eficiente, funcionava. Logo, seria um problema a menos a ser considerado na resolução do problema proposto.

3.3 Terceira pergunta

Qual foi a maior dificuldade em implementar o ambiente de execução?

A maior dificuldade provavelmente foi na parte de identificação dos rótulos pendentes, isto é, que ainda não haviam sido lidos mas eram utilizados em instruções. Por isso, para resolver o problema, foi necessário trazer grande simplificação do formato de arquivo, não podendo haver dois rótulos pendentes seguidos antes de ser resolvido o último rótulo pendente.

Também foi pensada a parte do desafio para implementar o debugger, porém não foi possível implementá-la. A ideia seria a utilização de um caracter, como "*"para indicar o *break point*. Entretanto, para imprimir os valores de todas as variáveis existiria o desafio de identificar quais são as variáveis presentes no programa, dado que ter um rótulo no início da linha não é o suficiente. Só seria possível implementar caso houvesse um espaço da memória definido apenas para as variáveis, mas existiria também o desafio de determinar o número de variáveis. Assim, pode ser que essa parte tenha sido a maior dificuldade, tanto que não foi possível implementá-la.

3.4 Quarta pergunta

Quais são os limites recomendados para o código em C– de forma a não gerar problemas na hora de execução?

Foram definidos diferentes limites para o formato do arquivo *asm* que serve como entrada do ambiente de execução. Abaixo, são listados:

- Sempre haver duas quebras de linhas entre as linhas do código, reconhecidas pelo caracter 0x00A;
- Na linha do @, o formato deve ser:

  <ARROBA> <ESPAÇO> <TAB> <BARRA> <END1> <END2>

  Onde END1 e END2 referem-se a uma primeira leitura do endereço de início e a uma segunda leitura deste mesmo endereço;
- Na linha que indica o fim do arquivo, é necessário apenas que haja o hashtag # e um espaço logo em seguida para identificar o fim;
- Para as linhas de instruções existem diferentes casos:

  - Caso com rótulo e instrução é K (constante):

  <WORD1> <WORD2> <TABTAB> <KTAB> <TABBARRA> <WORD3> <WORD4>

Onde as WORD1 e WORD2 referem-se às leituras do rótulo, e as WORD3 e WORD4 referem-se ao valor da constante.

- Caso com rótulo e instrução tem como parâmetro uma constante:

<WORD1> <WORD2> <TABTAB> <MNEM> <TABBARRA> <WORD3> <WORD4>

Onde as WORD1 e WORD2 referem-se às leituras do rótulo, e as WORD3 e WORD4 referem-se ao valor da constante.

- Caso com rótulo e instrução tem como parâmetro um rótulo:

<WORD1> <WORD2> <TABTAB> <MNEM> <WORD3> <WORD4>

Onde as WORD1 e WORD2 referem-se às leituras do rótulo, e as WORD3 e WORD4 referem-se ao valor de um outro rótulo.

- Caso em que não há rótulo no início da linha e instrução é K (constante):

<TABTAB> <KTAB> <TABBARRA> <WORD1> <WORD2>

Onde as WORD1 e WORD2 referem-se ao valor da constante.

- Caso em que não há rótulo no início da linha e instrução tem constante como parâmetro:

<TABTAB> <MNEM> <TABBARRA> <WORD1> <WORD2>

Onde as WORD1 e WORD2 referem-se ao valor da constante.

- Caso em que não há rótulo no início da linha e instrução tem rótulo como parâmetro:

<TABTAB> <MNEM> <TABTAB> <WORD1> <WORD2>

Onde as WORD1 e WORD2 referem-se ao rótulo como parâmetro.

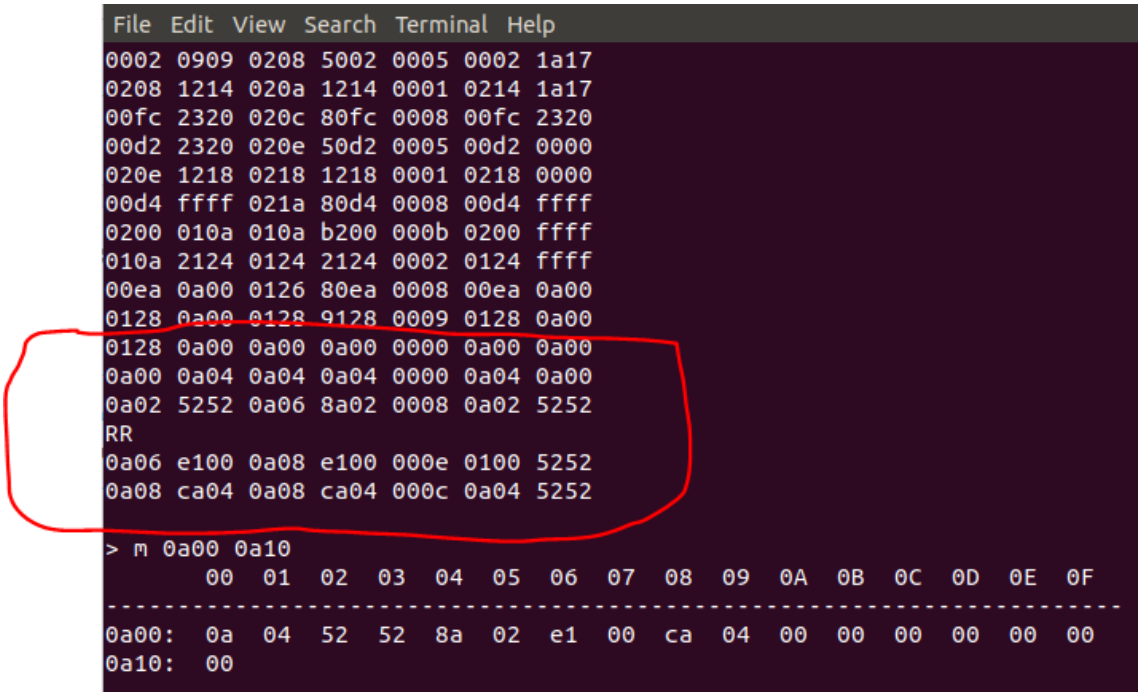
- Como demonstrado acima, assume-se que todos os rótulos tem 4 caracteres;
- Por fim, há a questão do rótulo pendente já descrita nas seções acima. É preciso que, para o correto funcionamento, não haja dois rótulos pendentes seguidos antes de ser resolvido o último rótulo pendente.

4 Simulação

Para o funcionamento do ambiente de execução basta salvar o código no arquivo code.asm no mesmo diretório da mvn, modificar o arquivo disp.lst para conter "3 0 code.asm l", e executar a mvn com o programa exec.mvn normalmente. O código montado é executrado logo após a montagem. A simulação foi feita de forma que o ambiente de execução recebeu o seguinte código em asm:

```
@       /0A00

        JP          INIC

RRAS    K           /5252

INIC    LD          RRAS

        PD      /0100

        HM          INIC

# INIC
```

O código executa um algoritmo simples, o qual escreve RR (lê-se "Ricardo Rocha") na tela do monitor da MVN. O ambiente de execução não só montou e executou o código de forma apropriada, mas tabém seguiu corretamente o endereçamento desejado:


File Edit View Search Terminal Help
0002 0909 0208 5002 0005 0002 1a17
0208 1214 020a 1214 0001 0214 1a17
00fc 2320 020c 80fc 0008 00fc 2320
00d2 2320 020e 50d2 0005 00d2 0000
020e 1218 0218 1218 0001 0218 0000
00d4 ffff 021a 80d4 0008 00d4 ffff
0200 010a 010a b200 000b 0200 ffff
010a 2124 0124 2124 0002 0124 ffff
00ea 0a00 0126 80ea 0008 00ea 0a00
0128 0a00 0128 9128 0009 0128 0a00
0128 0a00 0a00 0a00 0000 0a00 0a00
0a00 0a04 0a04 0a04 0000 0a04 0a00
0a02 5252 0a06 8a02 0008 0a02 5252
RR
0a06 e100 0a08 e100 000e 0100 5252
0a08 ca04 0a08 ca04 000c 0a04 5252
> m 0a00 0a10
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0a00: 0a 04 52 52 8a 02 e1 00 ca 04 00 00 00 00 00 00
0a10: 00

Vale notar que o código foi montado de forma que o conteúdo da memória a partir do endereço 0A00 ficasse:

```
0A00 0A04 ; JP INIC

0A02 5252 ; RRAS K /5252

0A04 8A02 ; INIC LD RRAS

0A06 E100 ; PD /0100

0A08 CA04 ; HM INIC
```

O que mostra que o ambiente de execução monta o código organizando os rótulos de forma correta e executando os comandos apropriados com os endereços corretos. Clar que dentro das restrições estabelecidas para o funcionamento.

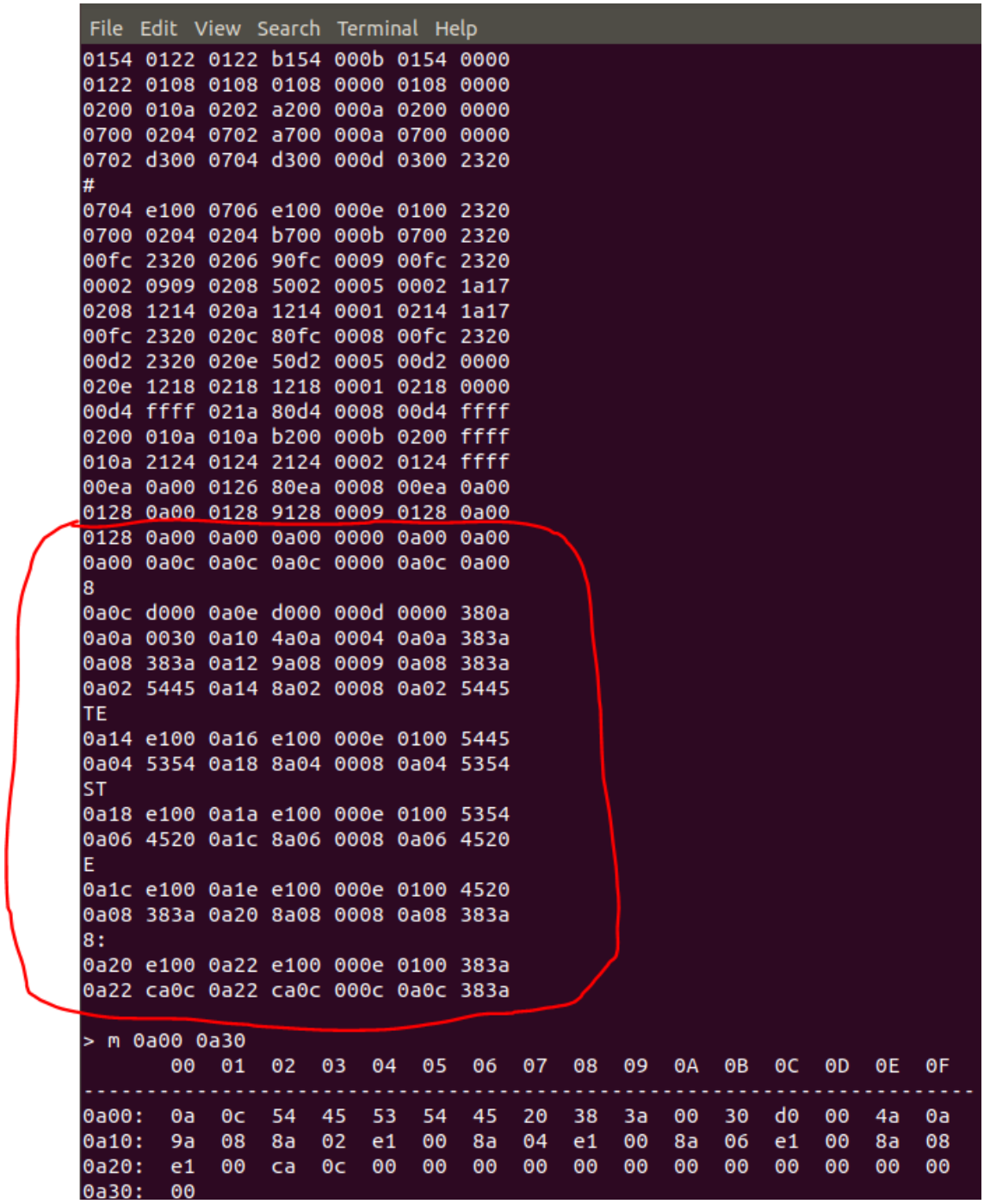
Outro código foi usado executando mais comandos e usando mais rótulos:

```
@       /0A00

        JP      INIC

TEAS    K       /5445

STAS    K       /5354

E_AS    K       /4520

_XAS    K       /2000

TRAS    K       /0030

INIC    GD      /0000

        AD      TRAS

        MM      _XAS

        LD      TEAS

        PD      /0100
```

```
LD          STAS
PD    /0100
LD          E_AS
PD    /0100
LD          _XAS
PD    /0100
HM          INIC
```

INIC

O código consiste da leitura de um número de 0 a 9 no teclado e o programa imprime "TESTE X:"na tela, onde X é o número lido no teclado. O código foi executado pelo ambiente corretamente como mostrado a seguir:


File Edit View Search Terminal Help
0154 0122 0122 b154 000b 0154 0000
0122 0108 0108 0108 0000 0108 0000
0200 010a 0202 a200 000a 0200 0000
0700 0204 0702 a700 000a 0700 0000
0702 d300 0704 d300 000d 0300 2320
#
0704 e100 0706 e100 000e 0100 2320
0700 0204 0204 b700 000b 0700 2320
00fc 2320 0206 90fc 0009 00fc 2320
0002 0909 0208 5002 0005 0002 1a17
0208 1214 020a 1214 0001 0214 1a17
00fc 2320 020c 80fc 0008 00fc 2320
00d2 2320 020e 50d2 0005 00d2 0000
020e 1218 0218 1218 0001 0218 0000
00d4 ffff 021a 80d4 0008 00d4 ffff
0200 010a 010a b200 000b 0200 ffff
010a 2124 0124 2124 0002 0124 ffff
00ea 0a00 0126 80ea 0008 00ea 0a00
0128 0a00 0128 9128 0009 0128 0a00
0128 0a00 0a00 0a00 0000 0a00 0a00
0a00 0a0c 0a0c 0a0c 0000 0a0c 0a00
8
0a0c d000 0a0e d000 000d 0000 380a
0a0a 0030 0a10 4a0a 0004 0a0a 383a
0a08 383a 0a12 9a08 0009 0a08 383a
0a02 5445 0a14 8a02 0008 0a02 5445
TE
0a14 e100 0a16 e100 000e 0100 5445
0a04 5354 0a18 8a04 0008 0a04 5354
ST
0a18 e100 0a1a e100 000e 0100 5354
0a06 4520 0a1c 8a06 0008 0a06 4520
E
0a1c e100 0a1e e100 000e 0100 4520
0a08 383a 0a20 8a08 0008 0a08 383a
8:
0a20 e100 0a22 e100 000e 0100 383a
0a22 ca0c 0a22 ca0c 000c 0a0c 383a
> m 0a00 0a30
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0a00: 0a 0c 54 45 53 54 45 20 38 3a 00 30 d0 00 4a 0a
0a10: 9a 08 8a 02 e1 00 8a 04 e1 00 8a 06 e1 00 8a 08
0a20: e1 00 ca 0c 00 00 00 00 00 00 00 00 00 00 00 00
0a30: 00

O programa foi montado também de forma correta, de forma que o conteúdo da memória a partir do endereço 0A00 ficasse:

```
0A00 0A0C ; JP INIC

0A02 5445 ; TEAS K /5445

0A04 5354 ; STAS K /5354

0A06 4520 ; E_AS K /4520

0A08 383A ; _XAS K /383A

0A0A 0030 ; TRAS K /0030

0A0C D000 ; INIC GD /0000

0A0E 4A0A ; AD TRAS

0A10 9A08 ; MM _XAS

0A12 8A02 ; LD TEAS

0A14 E100 ; PD /0100

0A16 8A04 ; LD STAS

0A18 E100 ; PD /0100

0A1A 8A06 ; LD E_AS

0A1C E100 ; PD /0100

0A1E 8A08 ; LD _XAS

0A20 E100 ; PD /0100

0A22 CA0C ; HM INIC
```

O que mostra que o teste foi executado de forma correta.