Fundamentos de Sistemas Computacionais - Trabalho 2

Isabella Leal Becker[*], Leticia Brasil Flores[†]
Escola Politécnica — PUCRS

2 de junho de 2022

Resumo

O Trabalho 2 da disciplina consiste na implementação de dois algoritmos utilizando a linguagem de montagem do computador Viking. A descrição de cada programa com a explicação da solução e a análise utilizando os recursos do simulador são apresentadas nesse relatório.

Descrição e análise de cada programa

No presente relatório, priorizamos a apresentação da solução final, todavia, somente foi possível chegar ao resultado esperado após algumas tentativas de implementações e testes utilizando, principalmente, o recurso de impressão no terminal, para identificação das falhas durante o desenvolvimento. Para cada um dos problemas descrevemos, a seguir, a solução implementada e a análise de alguns detalhes com o simulador Viking-sim:

Problema 1

O Problema 1 refere-se à implementação do algoritmo de ordenação Insertion Sort, que deve receber um vetor de vinte inteiros e retornar a solução ordenada, apresentando o vetor no terminal antes e depois da ordenação. Na solução implementada utilizamos chamada e reutilização de função.

Com base no exemplo da implementação do Bubble Sort do Manual Viking CPU, o programa inicia com a passagem de parâmetros para as funções que serão chamadas durante a execução, armazenando os endereços de retorno das funções em r6 (lr) e utilizando bnz r7 para fazer os desvios. Assim, na segunda linha do código abaixo, carregamos lr com o endereço de ret_print1 e desviamos para a execução de impressão do vetor print_vec. Ao final dessa função restauramos lr saltando para o endereço de retorno, onde faremos nova chamada e retorno de função.

```
1  main
2          ldi lr ,ret_print1
3          bnz r7 ,print_vec
4  ret_print1
5          ldi lr ,ret_insertion_sort
6          bnz r7 ,loop_insertion_sort
7  ret_insertion_sort
8          ldi lr ,ret_print2
9          bnz r7 ,print_vec
10 ret_print2
11         hcf
```

[*]isabella.becker@edu.pucrs.br
[†]leticia.flores@edu.pucrs.br

A função loop_insertion_sort equivale ao laço for da implementação utilizada como exemplo. Nela armazenamos em r3 a referência para o início do vetor, carregamos i no registrador r1 (sendo i = 1), incrementamos r3 a fim de chegar à posição do vetor onde teremos a variável auxiliar e guardamos o valor na variável aux (stw r2,aux). Também guardamos o valor da variável j, que recebe o valor de r1 (i). Em seguida, verificamos se o valor de i é maior que o tamanho do vetor: Se sim, temos o fim da excução desviando para end_i; caso contrário desviamos para o while.

```
1  loop_insertion_sort
2          ldi r3 ,vec
3          ldw r1 ,i
4          add r3 ,r3 ,r1
5          add r3 ,r3 ,r1
6          ldw r2 ,r3
7          stw r2 ,aux
8          stw r1 ,j
9          ldw r2 ,size
10         slt r4 ,r1 ,r2
11         bez r4 ,end_i
12         bnz r7 ,while
```

Em while carregamos o valor de j em r5, verificamos se j é maior que zero: Se não, desviamos para end_j; do contrário, iremos verificar se o elemento na posição anterior (j-1) é menor que a variável auxiliar. Para acessar a posição j-1 guardamos novamente em r3 o ponteiro para início do vetor, somamos duas vezes o valor de j em r3, pois o vetor é de inteiros, e em seguida subtraímos 2. Armazenamos o valor que está em r3 no registrador r2 e fazemos a verificação se o vetor na posição j-1 é maior que auxiliar, ou aux < vec[j-1]. Se aux é menor, desviamos para função swap para fazer a troca, do contrário desviamos para end_j.

```
1  while
2          ldi r4 ,1
3          ldw r5 ,j
4          slt r4 ,r5 ,r4
5          bnz r4 ,end_j
6          ldi r3 ,vec
7          add r3 ,r3 ,r5
8          add r3 ,r3 ,r5
9          sub r3 ,1
10         sub r3 ,1
11         ldw r2 ,r3
12         ldw r4 ,aux
13         slt r1 ,r4 ,r2
14         bnz r1 ,swap
15         bnz r7 ,end_j
```

A principal dificuldade na implementação dessa solução foi fazer o uso correto de ponteiros para avançar nas posições do vetor. Uma vez que conseguimos identificar que deveríamos utilizar o valor armazenado em j para atualizar cada posição, foi possível obter os resultados esperados. Na função swap apontamos novamente para o início do vetor guardando o endereço em r3, somamos duas vezes o valor de r5, que representa o valor de j, e assim chegamos na posição j do vetor onde iremos substituir pelo valor que está em r2, que é o j-1 do passo anterior. Ainda em swap decrementamos o registrador r5 e armazenamos novamente em j. Em seguida, fazemos o desvio incondicional retornando para o while.

```
1  swap
```

```
        ldi r3 ,vec
        add r3 ,r3 ,r5
        add r3 ,r3 ,r5
        stw r2 ,r3
        sub r5 ,1
        stw r5 ,j
        bnz r7 ,while
```

A função end_j armazena o valor de auxiliar na posição j do vetor e incrementa a variável i utilizando o registrador r1, em seguida retorna para a função loop_insertion_sort.

```
end_j
        ldw r4 ,aux
        ldi r3 ,vec
        add r3 ,r3 ,r5
        add r3 ,r3 ,r5
        stw r4 ,r3
        ldw r1 ,i
        add r1 ,1
        stw r1 ,i
        bnz r7 ,loop_insertion_sort
```

A função end_i imprimirá uma quebra de linha para exibição no terminal e fará o desvio para o endereço de retorno em lr, que será ret_print2.

```
end_i
        ldi r4 ,10
        stw r4 ,0xf000
        bnz r7 ,lr
```

A função de impressão dos valores no terminal é utilizada no início da execução do programa para exibição do vetor antes da ordenação e após a ordenação. Como trata-se de uma função simples, não iremos detalhar cada passo, entretanto, cabe destacar sua reutilização e o uso da chamada de função. Na sua primeira execução, a função retorna para o endereço de retorno armazenado em lr, que é ret_print1 e na segunda execução, retorna para ret_print2.

```
print_vec
        ldi r1 ,82
        stw r1 ,0xf000
        ldi r1 ,58
        stw r1 ,0xf000
        ldi r1 ,32
        stw r1 ,0xf000
        ldi r1 ,0
        ldw r2 ,size
        ldi r3 ,vec
loop_print_vec
        ldw r4 ,r3
        stw r4 ,0xf002
        ldi r4 ,32
        stw r4 ,0xf000
        add r1 ,1
        add r3 ,2
        slt r5 ,r1 ,r2
```

```
19          bnz r5 ,loop_print_vec
20          ldi r4 ,10
21          stw r4 ,0xf000
22          bnz r7 ,lr
```

Por fim, também incluímos ao nosso código a funcionalidade para preservar os registradores de r1 a r5, implementando as operações de Push e Pop.

Como forma de verificação da funcionalidade, utilizamos o seguinte vetor de 20 inteiros:

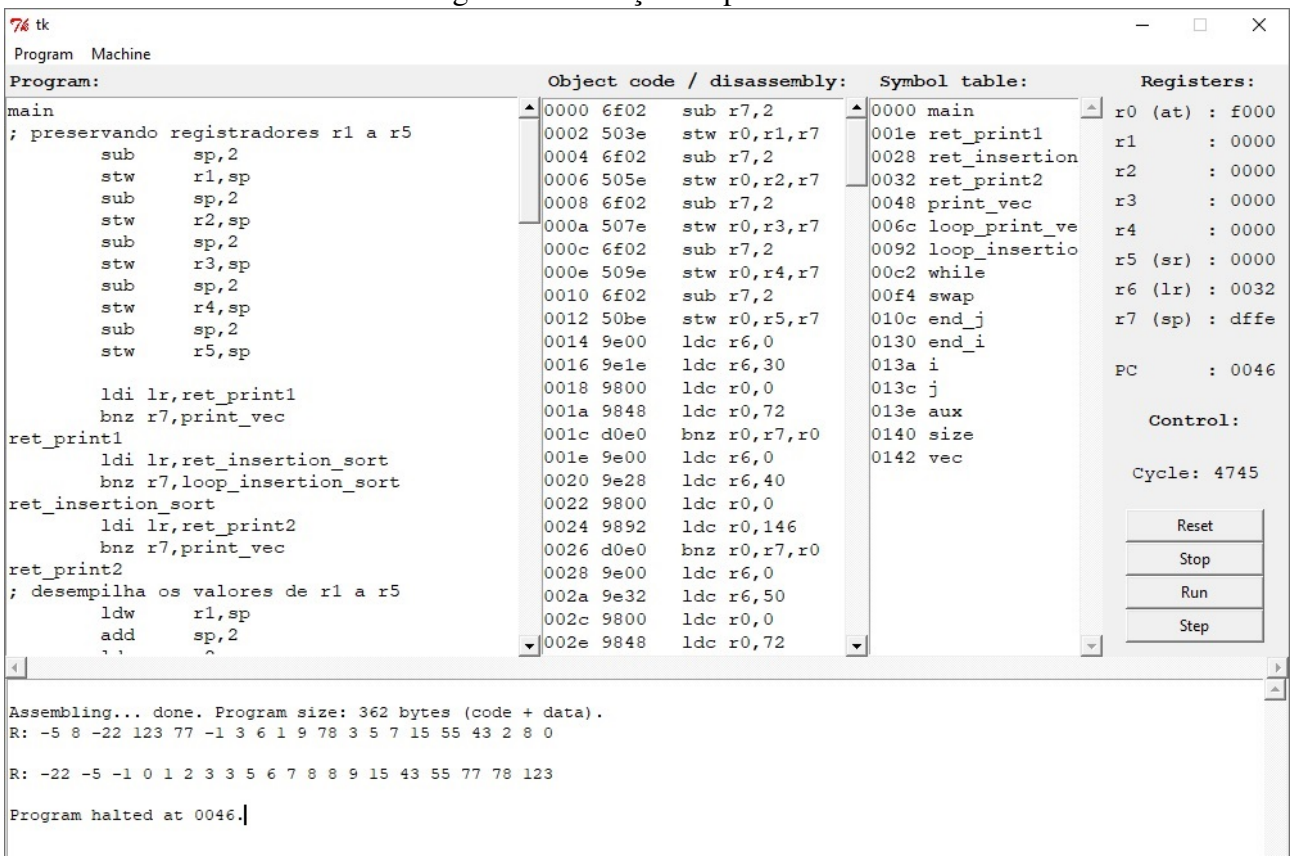
vec -5 8 -22 123 77 -1 3 6 1 9 78 3 5 7 15 55 43 2 8 0

Ao executar o programa (Figura 1) tivemos o retorno esperado apresentado no terminal.

R: -22 -5 -1 0 1 2 3 3 5 6 7 8 8 9 15 43 55 77 78 123

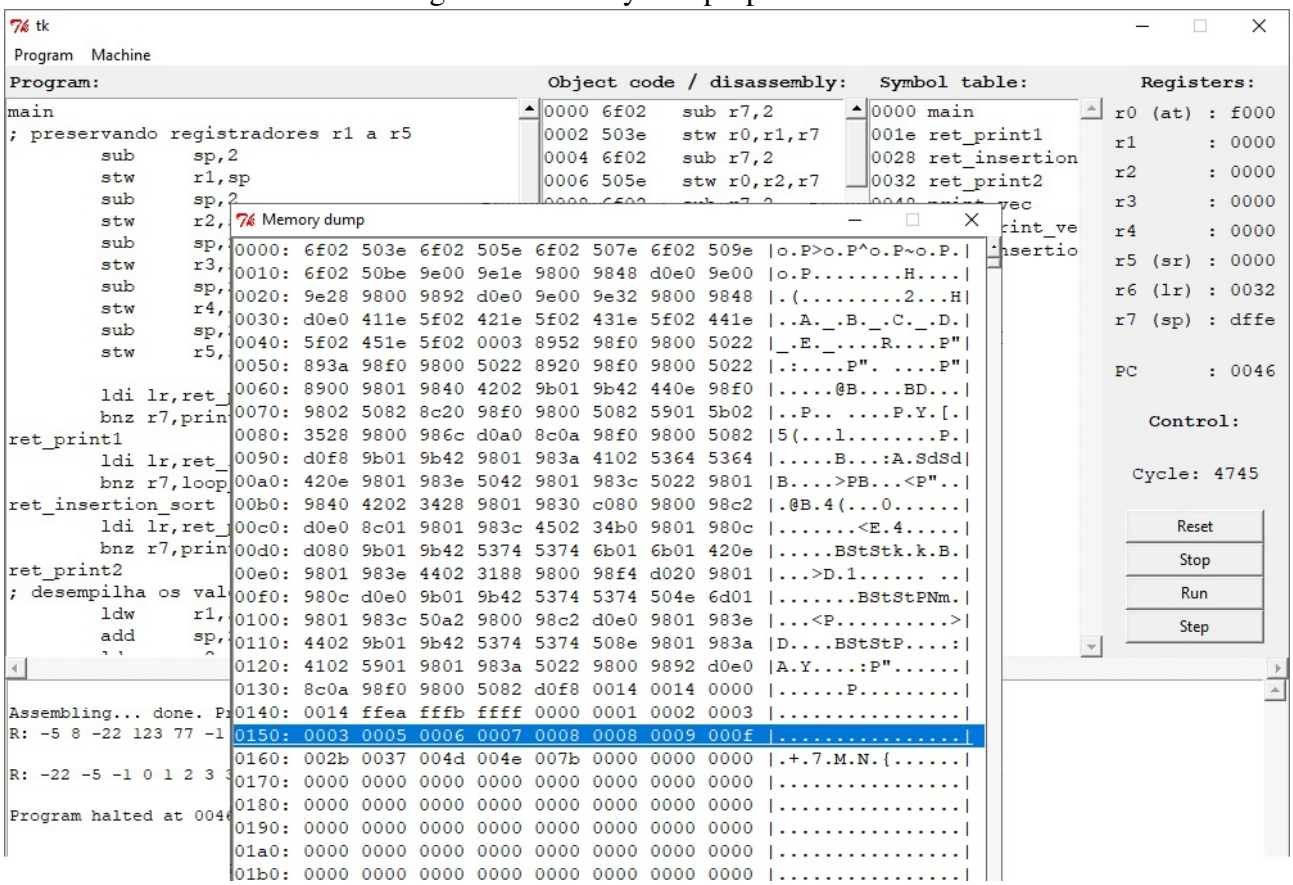
Podemos identificar, na tabela de símbolos, os rótulos do programa e as variáveis que foram utilizadas, e no campo Registers, onde os registradores de r1 a r5 constam como zerados após operação de Pop. Ainda, na tabela de símbolos, podemos identificar a posição inicial do vetor que iremos localizar utilizando o recurso de Memory Dump.

Figura 1: Execução do problema 1



Utilizando o recurso de Memory Dump podemos localizar a posição de memória onde ficou armazenada cada variável do programa, como: tamanho; variável auxiliar e elementos do vetor. Na Figura 2, podemos verificar que a posição 140, indicada na tabela de símbolos, possui o valor 14 em hexadecimal (20 em decimal) referente ao tamanho do vetor. Em seguida, na posição 142, inicia o vetor, que vai até a posição 168, onde está armazenado o último elemento do vetor já ordenado, 7b em hexadecimal (123 em decimal). Durante a execução do programa é possível acompanhar também a variação do registrador r3, entre as posições 142 a 168 do vetor.

Figura 2: Memory dump - problema 1



Problema 2

O Problema 2 deve contar o número de palavras armazenadas em uma String e apresentar o total no terminal. A solução implementada utiliza chamada de função e preserva o registrador responsável por armazenar o valor final da contagem de palavras, através da pilha.

O programa inicia armazenando o endereço da String mensagem no registrador r5, em seguida realizamos a operação Push para reservar um espaço na pilha e guardar o registrador r4, que será utilizado para receber o total de palavras encontradas. Utilizamos o registrador r6 (lr) para carregar o rótulo da função print_count e realizamos o desvio incondicional (bnz r7) para a função first_while.

```
1  main
2          ldi r5 ,mensagem
3          sub sp ,2
4          stw r4 ,sp
5          ldi lr ,print_count
6          bnz r7 ,first_while
```

A função first_while inicia recebendo o caractere armazenado na primeira posição da String e guarda no registrador r2, em seguida, utilizamos o registrador r1 para armazenar um valor da tabela ASCII de acordo com a função utilizada como referência, o que permitirá identificar o término de cada palavra.

Na linha 4 da função fisrt_while, realizamos a comparação: Se o valor armazenado em r2 é menor que o código em r1, realizamos o desvio para a função second_while; caso contrário, incrementamos o endereço r5, somando 1 e armazenamos o novo valor em r2. Ao final da função, verificamos se o novo valor armazenado em r2 é 0 (null): se não, executa a função novamente; caso contrário, significa que chegamos ao final do vetor de String. Desta forma, realizamos o desvio incondicional

para função ret_word que irá finalizar o programa. Durante a implementação, verificamos que, ao fazer essa verificação do caractere null, que significa o final da String, não seria mais necessário utilizarmos um contador para verificar o tamanho da String a cada incremento de r5.

```
first_while
        ldb r2 ,r5
        ldi r1 ,33
        slt r1 ,r2 ,r1
        bez r1 ,second_while
        add r5 ,1
        ldb r2 ,r5
        bnz r2 ,first_while
        bnz r7 ,ret_word
```

A função second_while inicia de forma semelhante a função first_while, armazenando o caractere de r5 em r2 e conferindo com o código da tabela ASCII, sendo necessário repetir essas instruções devido a recursão. Se após entrar no laço second_while encontrarmos um caractere menor que 33 ou maior que 127 da tabela ASCII, significa que chegamos ao final de uma palavra. Para essa verificação, utilizamos as operações SLT e BNZ, que fará o desvio para função count_word. Caso não ocorra o desvio, o registrador r5 é incrementado em 1, armazenando o novo valor em r2. Se r2 não for null repetimos a função second_while, do contrário finalizamos, desviando para função ret_word.

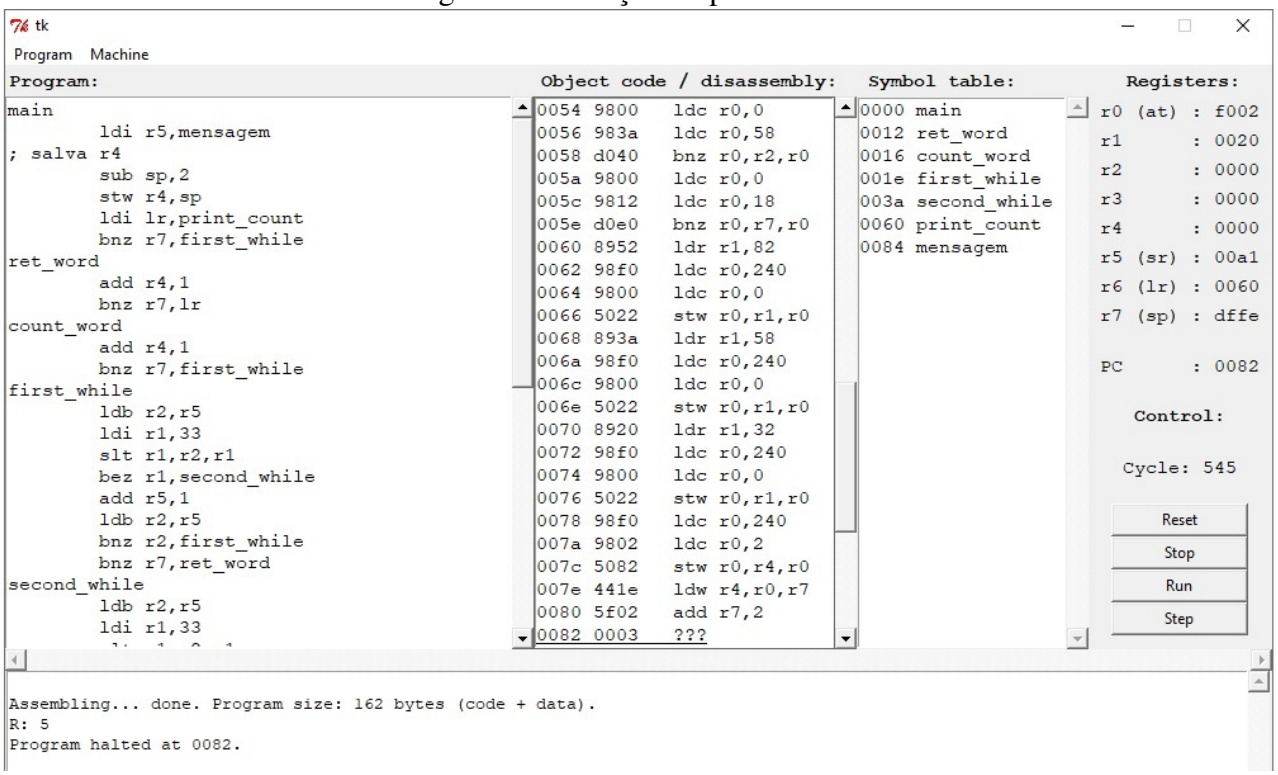
```
second_while
        ldb r2 ,r5
        ldi r1 ,33
        slt r1 ,r2 ,r1
        bnz r1 ,count_word
        ldi r1 ,127
        slt r1 ,r1 ,r2
        bnz r1 ,count_word
        add r5 ,1
        ldb r2 ,r5
        bnz r2 ,second_while
        bnz r7 ,ret_word
```

A função print_count irá apresentar o resultado no terminal. Utilizamos os endereços de saída de caractere e de inteiro do mapa de memória para apresentação do resultado. Ainda nessa função, realizamos a operação Pop para restaurar o registrador r4 e liberar espaço na pilha.

```
print_count
        ldi r1 ,82
        stw r1 ,0xf000
        ldi r1 ,58
        stw r1 ,0xf000
        ldi r1 ,32
        stw r1 ,0xf000
        stw r4 ,0xf002
        ldw r4 ,sp
        add sp ,2
        hcf
```

Como verificação da funcionalidade (Figura 3), utilizamos a String "labore et dolore magna aliqua", realizamos o teste e recebemos como resultado cinco palavras, R: 5.

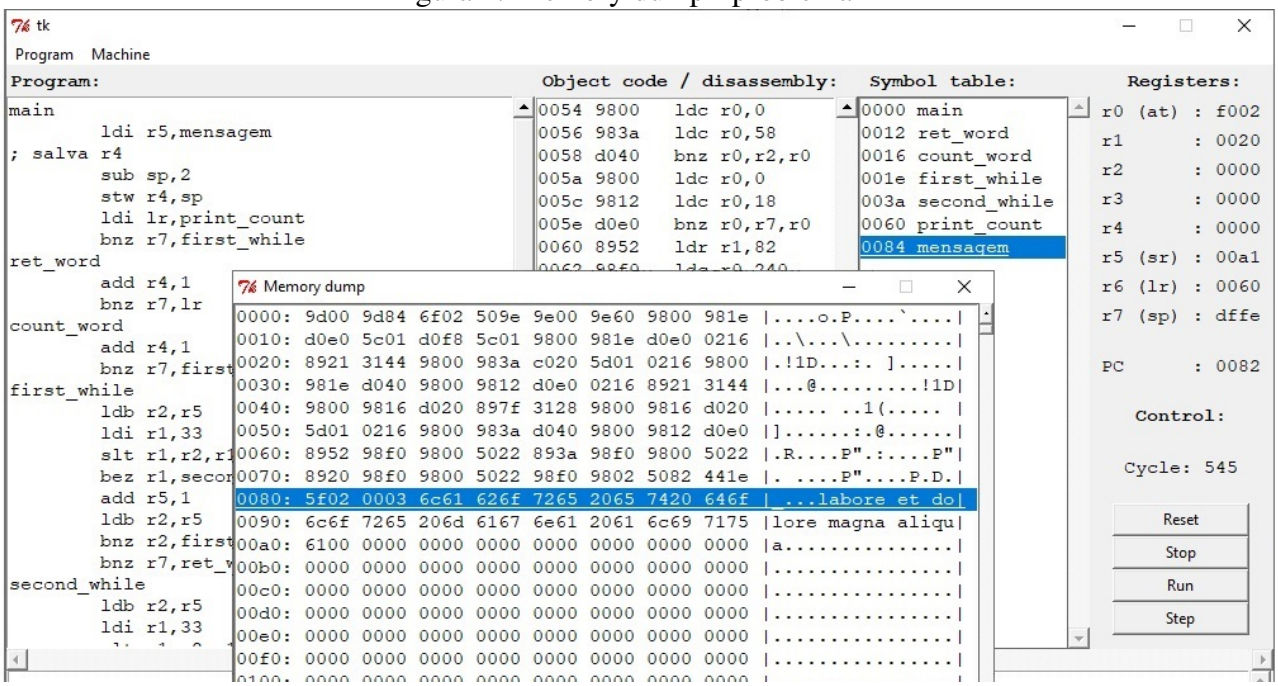
Figura 3: Execução do problema 2



Observa-se, a partir dos recursos do simulador Viking-sim, que após a montagem, todos os rótulos do programa são apresentados na tabela de símbolos.

Podemos analisar uma parte do nosso programa a partir desse recurso. Na tabela de símbolos, indica que a String, que é a nossa mensagem, está na posição 0084 (Figura 4). Ao verificarmos em Memory dump, podemos localizar os valores da String em hexadecimal. (Exemplo: 6c equivale a letra l, 61 equivale a letra a, ...).

Figura 4: Memory dump - problema 2



Logo, quando utilizamos a operação ldi r5,mensagem, estamos guardando esse endereço de memória em r5 (ponteiro) e quando incrementamos r5, estamos avançando em uma posição da String. Ao fazer a operação ldb r2,r5 estamos guardando em r2 o caractere que está armazenado em r5.

Ainda no simulador em Registers (Figura 5), podemos observar o comportamento do registrador r4, que utilizamos para armazenar o total de palavras, quando r2 possui o valor 20 (espaço) significa que chegamos a fim de uma palavra, portanto em r4 é somado 1. Assim, durante a execução, o r4 vai sendo modificado e ao final do programa é zerado, conforme esperado após operação Pop.

Figura 5: Registers - problema 2