Universidade Federal de São Paulo - Unifesp
Instituto de Ciência e Tecnologia - ICT

Arquitetura e Organização de Computadores
Professora Denise Stringhini

Alunos:
Beatriz Martins Angelo - 120202
Claudio Jorge Lopes Filho - 120223
Israel da Rocha - 120432

Robótica no simulador MARS - Tutorial: Robô Seguidor de Linhas

28 de novembro de 2019

1. Objetivo

Este trabalho visa apresentar um tutorial passo-a-passo para que seja possível se desenvolver um simulador de robô no MARS (*MIPS Assembler and Runtime Simulator*). Juntamente ao tutorial, o código desenvolvido será apresentado para melhor entendimento do passo-a-passo.

Neste projeto, o robô é seguidor de linha, ou seja, ele deve encontrar uma linha apresentada no *Bitmap Display* e percorrê-la inteira.

O desenvolvimento segue as seguintes especificações:

- O simulador deve possibilitar diferentes desenhos de linhas;
- Inicialmente o robô é gerado em uma posição aleatória do *display* e deve encontrar a linha;
- Após encontrar a linha, ele deve segui-la até o final.

2. Simulador MARS

O simulador MARS (*MIPS Assembler and Runtime Simulator*) é um ambiente de desenvolvimento integrado (IDE) destinado para se estudar a arquitetura MIPS.

Para utilizá-lo, deve-se programar na linguagem Assembly, que é uma linguagem de montagem, ou seja, uma linguagem mais próxima aos códigos binários entendidos pelas máquinas, de modo que o processador não precisa recorrer ao compilador para decodificá-la.

Além da disponibilidade de diversas instruções e funções necessárias para a implementação do código, o simulador possui uma ferramenta de interface gráfica chamada de *Bitmap Display* para visualização do funcionamento do projeto. Essa ferramenta será utilizada neste trabalho.

Para utilizar o simulador, acesse o link a seguir e faça o download gratuito:

https://courses.missouristate.edu/KenVollmar/MARS/download.htm

3. Tutorial passo-a-passo

Para visualizar o funcionamento do programa, vá em *Tools* -> *Bitmap Display*. Clique em *Configuration* e configure os valores como vistos na Figura 1.

Inicialmente, após o *.text* utilizamos o operador *.eqv* que nos auxilia a alterar o nome dos registradores utilizados para facilitar a leitura e para tornar o código mais intuitivo.

```
8  # Display config:
9  #       Unit Width:     32
10 #       Unit Height:    32
11 #       Display Width:  512
12 #       Display Height: 512
13 #       Base Addres:    0x10040000 (heap)
14
15
16 .text
17         .eqv preto, $s7
18         .eqv pos_atual, $s0
19         .eqv pos_inicial, $s1
20         .eqv vermelho, $s2
21         .eqv azul, $s3
22         .eqv pilha1, $sp
23         .eqv pilha2, $fp
24
```

(Figura 1)

Para a execução do projeto seguimos os seguintes passos:

1. Decidir o início da linha.
2. Gerar a linha aleatoriamente.
    a. Verificar os limites do Bitmap Display para evitar inconsistências.
    b. Evitar que a linha seja adjacente a ela mesma.
3. Gerar o Robô em uma posição aleatória do Bitmap.
4. Fazer o Robô encontrar a linha.
5. Fazer o Robô percorrer a linha inteira.

Para compreender o passo a passo e poder reproduzir o projeto, é necessário saber as principais instruções em *Assembly* MIPS. Essas instruções são apresentadas na tabela da Figura 2 e utilizadas durante o código, onde suas utilidades serão melhor explicadas.

MIPS instructions

These are some of the most common MIPS instructions and pseudo-instructions, and should be all you need. However, you are free to use *any* valid MIPS instructions or pseudo-instruction in your programs.

Category	Example Instruction	Meaning
Arithmetic	add $t0, $t1, $t2	$t0 = $t1 + $t2
	sub $t0, $t1, $t2	$t0 = $t1 – $t2
	addi $t0, $t1, 100	$t0 = $t1 + 100
	mul $t0, $t1, $t2	$t0 = $t1 x $t2
	div $t0, $t1, $t2	$t0 = $t1 / $t2
Logical	and $t0, $t1, $t2	$t0 = $t1 & $t2 (Logical AND)
	or $t0, $t1, $t2	$t0 = $t1 | $t2 (Logical OR)
	sll $t0, $t1, $t2	$t0 = $t1 << $t2 (Shift Left Logical)
	srl $t0, $t1, $t2	$t0 = $t1 >> $t2 (Shift Right Logical)
Register Setting	move $t0, $t1	$t0 = $t1
	li $t0, 100	$t0 = 100
Data Transfer	lw $t0, 100($t1)	$t0 = Mem[100 + $t1] 4 bytes
	lb $t0, 100($t1)	$t0 = Mem[100 + $t1] 1 byte
	sw $t0, 100($t1)	Mem[100 + $t1] = $t0 4 bytes
	sb $t0, 100($t1)	Mem[100 + $t1] = $t0 1 byte
Branch	beq $t0, $t1, Label	if ($t0 = $t1) go to Label
	bne $t0, $t1, Label	if ($t0 ≠ $t1) go to Label
	bge $t0, $t1, Label	if ($t0 ≥ $t1) go to Label
	bgt $t0, $t1, Label	if ($t0 > $t1) go to Label
	ble $t0, $t1, Label	if ($t0 ≤ $t1) go to Label
	blt $t0, $t1, Label	if ($t0 < $t1) go to Label
Set	slt $t0, $t1, $t2	if ($t1 < $t2) then $t0 = 1 else $t0 = 0
	slti $t0, $t1, 100	if ($t1 < 100) then $t0 = 1 else $t0 = 0
Jump	j Label	go to Label
	jr $ra	go to address in $ra
	jal Label	$ra = PC + 4; go to Label

The second source operand of the arithmetic, logical, and branch instructions may be a constant.

Register Conventions

The *caller* is responsible for saving any of the following registers that it needs, before invoking a function.

$t0-$t9 $a0-$a3 $v0-$v1

The *callee* is responsible for saving and restoring any of the following registers that it uses.

$s0-$s7 $ra

Pointers in C:

Declarartion: either *char *char_ptr* -or- *char char_array[]* for *char c*
Dereference: *c = c_array[i]* -or- *c =*c_pointer*
Take address of: *c_pointer = &c*

1

(Figura 2. Fonte: https://www.slideserve.com/cady/mips-instructions)

PASSO 1 - Decidir o início da linha:

Como podemos ver na figura 3, nas linhas 26-31 inicializamos os principais registradores que precisaremos utilizar. Entre eles temos o $s1 que guarda a posição inicial do Bitmap, $s0 que guarda a posição atual (tanto da linha quanto do Robô futuramente) e

$s2 e $s3 que guardam as cores em hexadecimal para pintarmos o Bitmap (fique a vontade para escolher as cores do seu Robô).

As instruções utilizadas *li* e *lui* simplesmente atribuem o valor hexadecimal ao registrador correspondente. Já a instrução *addi* soma o valor que está no terceiro parâmetro ao valor que está no registrador do segundo parâmetro e guarda o resultado no primeiro registrador.

```
24
25  .main:
26          lui $s0, 0x1004         # Pos atual
27          lui $s1, 0x1004         # Pos inicial
28          li $s2, 0x00FF0000      # Cor Vermelha
29          li $s3, 0x7FFFD4        # Cor Azul - Robo
30          addi $sp, $s1, 1024     # Pos inicial pilha1
31          addi $fp, $sp, 1024     # Pos atual pilha 2
```

(Figura 3)

Na figura 4, mostra-se o primeiro passo para a criação da linha a ser seguida pelo robô. Para isso, utiliza-se o comando *syscall*, que é uma chamada para serviços do sistema para entrada e saída. O código 42 é o que gera um número aleatório em um intervalo inteiro, no caso, de 0 a 255. Com o comando *move* o número gerado é colocado no registrador $s4.

```
33          # Gera numero aleatorio de 0 a 255 e salva em pos_atual
34          li $v0, 42
35          li $a1, 255
36          syscall
37          move $s4, $a0
```

(Figura 4)

Esse número é multiplicado por 4, uma vez que cada “quadrado” do *Bitmap display* tem 4 bytes. A multiplicação por 4 não é trivial. Para isso duplicamos e depois duplicamos o número duplicado. Utilizamos essa sequência de instruções pois não podemos multiplicar por um imediato, como visto na figura 5.

```
40          add $s4, $s4, $s4
41          add $s4, $s4, $s4
42          add pos_atual, pos_atual, $s4
```

(Figura 5)

Como a linha a ser seguida é vermelha, essa posição é pintada utilizando-se o comando *sw* para inserir o novo valor na memória. Por fim, as pilhas utilizadas são atualizadas.

```
44          # Pinta de vermelho a posicao armazenada em $t0 (Bitmap Display)
45          sw vermelho, (pos_atual)
46
47          # Atualiza pilhas
48          addi pilha1, pos_atual, 1024
49          sw $zero, (pilha1)              #armazena indice na pilha1
50          sw pos_atual, (pilha2)          #armazena posicao na pilha2
51          addi pilha2, pilha2, 4
```

(Figura 6)

```
52
53          # Chama a funcao "labirinto" para criar a linha a ser seguida pelo robo
54          li $s4, 1
55          li $s5, 31
56
57          # Para criar uma linha de tamanho 30
58          laco:
59                  # Delay
60                  li $v0, 32
61                  li $a0, 40
62                  syscall
63                  beq $s4, $s5, continue
64                  jal labirinto
65                  addi $s4, $s4, 1
66                  #addi $s6, $s6, 1
67                  j laco
68
69          continue:
70
71                  #jal gera_robo
72                  jal pintaRobo
73                  jal procura
74                  jal percorre
75
76          # Encerra o programa
77          fim:
78                  li $v0, 10
79                  syscall
80
```

(Figura 7)

Na figura 7 conseguimos ver o esqueleto principal do programa que junta todos os passos. Vamos dissecá-los um a um.

A partir dessa posição inicial decidida, as posições restantes da linha são geradas utilizando-se de um laço de repetição. Os registradores $s4 e $s5 servem como controle para isso, pois o laço executará enquanto $s4 for diferente de 31. As linhas 60-62 geram um delay para que seja possível ver a linha sendo gerada.

Para isso, é necessário utilizar as instruções *beq, jal* e *j*. As instruções *jal* e *j* são utilizadas para que a execução do código vá para a linha que possui o nome colocado na instrução. A diferença entre as duas é que *jal* guarda a posição do *Program Counter* no registrador *$ra* fazendo com que atue similarmente a uma chamada de função como nas linguagens de programação de alto nível.

Já a instrução *beq* faz uma comparação entre o conteúdo dos registradores e se eles forem iguais a execução vai pular para a linha especificada.

Quando a linha está pronta, o comando *beq* faz com que a função *continue* seja executada, portanto, o robô é criado, busca a linha e a percorre.

Quando todas as funções tiverem sido executadas, a execução será redirecionada para fim e a *syscall* com código 10 encerrará o programa.

PASSO 2 - Gerar a linha aleatoriamente:

O funcionamento básico é este: cada posição pode ser gerada a partir da anterior em 4 posições, cima, baixo, direita e esquerda. Dessa forma, gera-se um número aleatório, da mesma maneira como geramos anteriormente, para decidir em qual dessas direções a nova posição se encontrará. Essa posição será avaliada pela função checa_loop para que garanta uma linha válida. Se a posição for válida, ela é pintada de vermelho, as pilhas são atualizadas e a execução volta para gerar a próxima posição. A implementação pode ser vista nas figuras 8, 9 e 10. Agora explicaremos cada pedaço.

```
81  labirinto:
82          li $t6, 0
83          li $t7, 0
84          li $t8, 0
85          li $t9, 0
86
87          # Gera um numero aleatorio de 0 a 3 e salva em pos_atual
88          # 4 posicoes possiveis de quadrados adjacentes ao que ja esta pintado
89          setup:
90                  li $v0, 42
91                  li $a1, 4
92                  syscall
93                  move $t0, $a0
94
95                  # Compara o numero gerado e vai para a funcao correspondente
96                  beq $t0, 0, pZero # Esquerda
97                  beq $t0, 1, pUm   # Direita
98                  beq $t0, 2, pDois # Baixo
99                  beq $t0, 3, pTres # Cima
100
101         pZero:
102                 j checa_loop # Verifica se ja existe linha nos pixels adjacentes
103                 voltaZero:
104                         sne $t6, $t2, 1
105                         bne $t2, 1, setup
106                         addi pos_atual, pos_atual, -4 # Subtrai 4 bits para ir para o quadrado da esquerda
107                         sw vermelho, (pos_atual)
108
109                         #Atualiza pilhas
110                         addi pilha1, pos_atual, 1024
111                         sw $s4, (pilha1) #armazena indice na pilha1
112                         sw pos_atual, (pilha2) #armazena posicao na pilha2
113                         addi pilha2, pilha2, 4
114
115                         j end
```

(Figura 8)

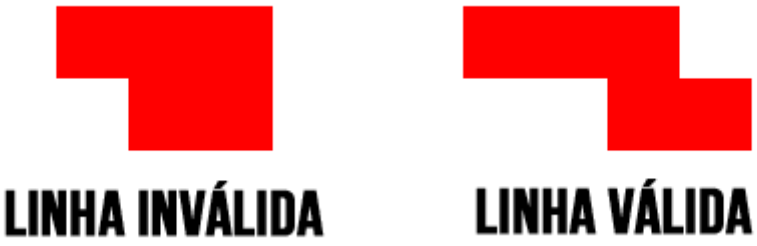
```
117         pUm:
118                 j checa_loop # Verifica se ja existe linha nos pixels adjacentes
119                 voltaUm:
120                         sne $t7, $t2, 1
121                         bne $t2, 1, setup
122                         addi pos_atual, pos_atual, 4 # Soma 4 bits para ir para o quadrado da direita
123                         sw vermelho, (pos_atual)
124 
125                         # Atualiza pilhas
126                         addi pilha1, pos_atual, 1024
127                         sw $s4, (pilha1) #armazena indice na pilha1
128                         sw pos_atual, (pilha2) #armazena posicao na pilha2
129                         addi pilha2, pilha2, 4
130 
131                         j end
132 
133         pDois:
134                 j checa_loop # Verifica se ja existe linha nos pixels adjacentes
135                 voltaDois:
136                         sne $t8, $t2, 1
137                         bne $t2, 1, setup
138                         addi pos_atual, pos_atual, 64 # Soma 64 bits para ir para o quadrado abaixo
139                         sw vermelho, (pos_atual)
140 
141                         # Atualiza pilhas
142                         addi pilha1, pos_atual, 1024
143                         sw $s4, (pilha1) #armazena indice na pilha1
144                         sw pos_atual, (pilha2) #armazena posicao na pilha2
145                         addi pilha2, pilha2, 4
146 
147                         j end
148 
```

(Figura 9)

```
148 
149         pTres:
150                 j checa_loop # Verifica se ja existe linha nos pixels adjacentes
151                 voltaTres:
152                         sne $t9, $t2, 1
153                         bne $t2, 1, setup
154                         addi pos_atual, pos_atual, -64 # Subtrai 64 bits para ir para o quadrado acima
155                         sw vermelho, (pos_atual)
156 
157                         #Atualiza pilhas
158                         addi pilha1, pos_atual, 1024
159                         sw $s4, (pilha1) #armazena indice na pilha1
160                         sw pos_atual, (pilha2) #armazena posicao na pilha2
161                         addi pilha2, pilha2, 4
162 
163                         j end
164         end:
165                 jr $ra # Retorna para a linha seguinte a que a funcao foi chamada
```

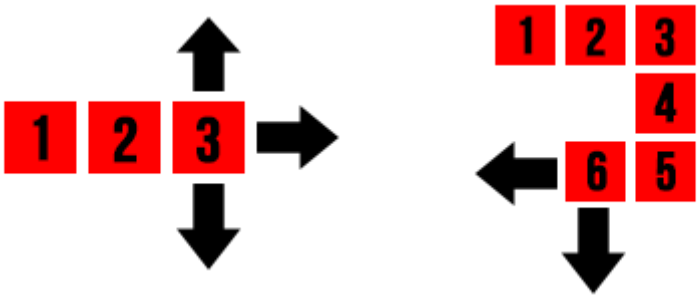
(Figura 10)

- O que é considerado uma linha válida?



(Figura 11)

- Como descobrir se a próxima posição gera uma linha válida?



(Figura 12)

Note que na linha da esquerda a próxima posição pode ser 3 entre as 4 possíveis. Por outro lado, a linha da direita só pode ir para 2 das 4 posições possíveis visto que a posição da direita já está ocupada e a posição de cima resultaria em uma linha inválida.

```
81  labirinto:
82          li $t6, 0
83          li $t7, 0
84          li $t8, 0
85          li $t9, 0
```

(Figura 13)

Como visto na figura 13, a função labirinto começa utilizando o comando *li* para inicializar quatro registradores com o valor zero. Eles servirão para checar se a posição é válida ou não durante a execução da função.

Quando os quatro estiverem com o valor 1 significa que nenhuma posição para qual a linha pode prosseguir é válida e portanto a geração da linha é encerrada. Isso pode causar variação no tamanho da linha já que essa ação depende dos números aleatórios gerados.

Em seguida temos o label *setup* que indica o início das ações da função labirinto.

Inicialmente geramos o número aleatório que indica a posição e o colocamos em $t0. Tmos logo abaixo quatro comandos *beq* que direcionam a execução de acordo com a posição para a qual queremos ir.

```
87          # Gera um numero aleatorio de 0 a 3 e salva em pos_atual
88          # 4 posicoes possiveis de quadrados adjacentes ao que ja esta pintado
89          setup:
90                  li $v0, 42
91                  li $a1, 4
92                  syscall
93                  move $t0, $a0
94
95                  # Compara o numero gerado e vai para a funcao correspondente
96                  beq $t0, 0, pZero # Esquerda
97                  beq $t0, 1, pUm   # Direita
98                  beq $t0, 2, pDois # Baixo
99                  beq $t0, 3, pTres # Cima
```

(Figura 14)

```
100
101         pZero:
102                 j checa_loop # Verifica se ja existe linha nos pixels adjacentes
103                 voltaZero:
104                         sne $t6, $t2, 1
105                         bne $t2, 1, setup
106                         addi pos_atual, pos_atual, -4 # Subtrai 4 bits para ir para o quadrado da esquerda
107                         sw vermelho, (pos_atual)
108
109                         #Atualiza pilhas
110                         addi pilha1, pos_atual, 1024
111                         sw $s4, (pilha1) #armazena indice na pilha1
112                         sw pos_atual, (pilha2) #armazena posicao na pilha2
113                         addi pilha2, pilha2, 4
114
115                         j end
```

(Figura 15)

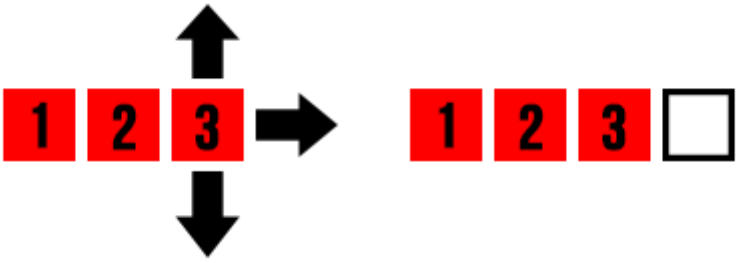
A figura 15 mostra um dos ramos de execução da função, nesse caso para quando a próxima posição é a da esquerda.

Como visto na figura 8, 9 e 10, temos quatro desses ramos e apenas pequenas mudanças existem entre eles.

Vamos explicar apenas um deles e consideraremos que o resto ficará trivial.

Logo no começo temos a chamada para a função *checa_loop*. Essa função verifica se a posição é válida em todos os sentidos.

- Como descobrir se uma posição é válida?



(Figura 16)

Considere a linha da figura 16. Suponha que o número aleatório gerado esteja tentando continuar a linha para a direita. Para descobrirmos se a posição é válida basta checar se já existe alguma posição da linha que está adjacente a posição que queremos gerar, em outros termos: calcular quantos vizinhos vermelhos ela tem.

O código da figura 17 inicia a função verificando se todas as posições são válidas. Para isso checamos os registradores $t6, $t7, $t8 e $t9 para ver se todos possuem o valor 1 ou não. Caso sim, interrompemos a geração da linha, caso não, executamos o código da *checa_loop*.

```
checa_loop:
        li $t1, 0 # posicao para verificacao
        li $t2, 0 # contador de numero de vizinhos

        bne $t6, 1, ok
        bne $t7, 1, ok
        bne $t8, 1, ok
        bne $t9, 1, ok
        add $s4, $zero, 30
        j end
```

(Figura 17)

Assim como a função *labirinto*, a função *checa_loop* possui casos diferentes dependendo da direção que está tentando tomar.

No código da figura 18 armazenamos em *$t3* a posição candidata a ser a próxima posição da linha. Em seguida carregamos o conteúdo da posição de memória no registrador *$t4*. Verificamos se *$t4* é igual a cor vermelho para checar se a posição já foi pintada anteriormente.

Caso não, pulamos para o Label *limites*.

```
        Zero:
                addi $t3, pos_atual, -4 # vai para a posicao gerada
                lw $t4, ($t3)
                beq $t4, vermelho, retorno # verifica se a posicao ja esa pintada
                j limites
        Um:
                addi $t3, pos_atual, 4 # vai para a posicao gerada
                lw $t4, ($t3)
                beq $t4, vermelho, retorno # verifica se a posicao ja esa pintada
                j limites
        Dois:
                addi $t3, pos_atual, 64 # vai para a posicao gerada
                lw $t4, ($t3)
                beq $t4, vermelho, retorno # verifica se a posicao ja esa pintada
                j limites
        _
```

(Figura 18)

Nas figuras 19 e 20 mostramos o código da label limites.

```
limites:
        sgt $t5, $t3, 0x100403FC # Limite Inferior do mapa
        beq $t5, 1, retorno

        slt $t5, $t3, pos_inicial # Limite Superior do mapa
        beq $t5, 1, retorno

        beq $t0, 0, limites_esquerda # se a proxima posicao for a esquerda, verifica os limites na esquerda
        beq $t0, 1, limites_direita # se a proxima posicao for a direita, verifica os limites na direita
        j cima

        # verifica se a posicao atual eh uma das bordas da direita
        limites_direita:
                beq pos_atual, 0x1004003C, retorno
                beq pos_atual, 0x1004007C, retorno
                beq pos_atual, 0x100400BC, retorno
                beq pos_atual, 0x100400FC, retorno
                beq pos_atual, 0x1004013C, retorno
                beq pos_atual, 0x1004017C, retorno
                beq pos_atual, 0x100401BC, retorno
                beq pos_atual, 0x100401FC, retorno
                beq pos_atual, 0x1004023C, retorno
                beq pos_atual, 0x1004027C, retorno
                beq pos_atual, 0x100402BC, retorno
                beq pos_atual, 0x100402FC, retorno
                beq pos_atual, 0x1004033C, retorno
                beq pos_atual, 0x1004037C, retorno
                beq pos_atual, 0x100403BC, retorno
                beq pos_atual, 0x100403FC, retorno
                j cima
```

(Figura 19)

```
236                 # verifica se a posicao atual eh uma das bordas da esquerda
237                 limites_esquerda:
238                         beq pos_atual, 0x10040000, retorno
239                         beq pos_atual, 0x10040040, retorno
240                         beq pos_atual, 0x10040080, retorno
241                         beq pos_atual, 0x100400C0, retorno
242                         beq pos_atual, 0x10040100, retorno
243                         beq pos_atual, 0x10040140, retorno
244                         beq pos_atual, 0x10040180, retorno
245                         beq pos_atual, 0x100401C0, retorno
246                         beq pos_atual, 0x10040200, retorno
247                         beq pos_atual, 0x10040240, retorno
248                         beq pos_atual, 0x10040280, retorno
249                         beq pos_atual, 0x100402C0, retorno
250                         beq pos_atual, 0x10040300, retorno
251                         beq pos_atual, 0x10040340, retorno
252                         beq pos_atual, 0x10040380, retorno
253                         beq pos_atual, 0x100403C0, retorno
```

(Figura 20)

O Bitmap do Mars é um vetor de posições de memória contíguas, ou seja, se estivermos na última posição de uma linha e somarmos 4 para acessar a próxima posição iremos para a primeira posição da linha de baixo. O código abaixo da label limites verifica os limites do Bitmap para que situações como a descrita acima não aconteçam.

Na figura 21 checamos se a posição atual é maior do que a última posição possível ou menor do que a posição inicial.

```
limites:
        sgt $t5, $t3, 0x100403FC # Limite Inferior do mapa
        beq $t5, 1, retorno

        slt $t5, $t3, pos_inicial # Limite Superior do mapa
        beq $t5, 1, retorno
```

(Figura 21)

Na figura 22 temos a verificação do limite lateral da direita. Todas as posições são múltiplas de 60 e caso estejamos em alguma delas não permitimos que a linha progrida para a direita. O mesmo vale para a figura 23 só que para o lado esquerdo, no caso as posições são múltiplas de 64.

```
216                 # verifica se a posicao atual eh uma das bordas da direita
217                 limites_direita:
218                         beq pos_atual, 0x1004003C, retorno
219                         beq pos_atual, 0x1004007C, retorno
220                         beq pos_atual, 0x100400BC, retorno
221                         beq pos_atual, 0x100400FC, retorno
222                         beq pos_atual, 0x1004013C, retorno
223                         beq pos_atual, 0x1004017C, retorno
224                         beq pos_atual, 0x100401BC, retorno
225                         beq pos_atual, 0x100401FC, retorno
226                         beq pos_atual, 0x1004023C, retorno
227                         beq pos_atual, 0x1004027C, retorno
228                         beq pos_atual, 0x100402BC, retorno
229                         beq pos_atual, 0x100402FC, retorno
230                         beq pos_atual, 0x1004033C, retorno
231                         beq pos_atual, 0x1004037C, retorno
232                         beq pos_atual, 0x100403BC, retorno
233                         beq pos_atual, 0x100403FC, retorno
234                         j cima
```

(Figura 22)

```
236                 # verifica se a posicao atual eh uma das bordas da esquerda
237                 limites_esquerda:
238                         beq pos_atual, 0x10040000, retorno
239                         beq pos_atual, 0x10040040, retorno
240                         beq pos_atual, 0x10040080, retorno
241                         beq pos_atual, 0x100400C0, retorno
242                         beq pos_atual, 0x10040100, retorno
243                         beq pos_atual, 0x10040140, retorno
244                         beq pos_atual, 0x10040180, retorno
245                         beq pos_atual, 0x100401C0, retorno
246                         beq pos_atual, 0x10040200, retorno
247                         beq pos_atual, 0x10040240, retorno
248                         beq pos_atual, 0x10040280, retorno
249                         beq pos_atual, 0x100402C0, retorno
250                         beq pos_atual, 0x10040300, retorno
251                         beq pos_atual, 0x10040340, retorno
252                         beq pos_atual, 0x10040380, retorno
253                         beq pos_atual, 0x100403C0, retorno
```

(Figura 23)

Após essa bateria de verificações iniciamos mais uma vista na figura 24. Dessa vez checamos a quantidade de vizinhos vermelhos da posição candidata. Lembrando que só pintaremos essa posição caso o número de vizinhos seja exatamente 1.

```
255         # verifica se ja existe linha acima da proxima posicao
256         cima:
257                 addi $t1, $t3, -64
258                 lw $t5, ($t1) # carrega o valor contido na posicao de memoria
259                 bne $t5, vermelho, baixo
260                 addi $t2, $t2, 1 # incrementa 1 no numero de vizinhos
261
262         # verifica se ja existe linha abaixo da proxima posicao
263         baixo:
264                 li $t1, 0
265                 addi $t1, $t3, 64
266                 lw $t5, ($t1) # carrega o valor contido na posicao de memoria
267                 bne $t5, vermelho, esquerda
268                 addi $t2, $t2, 1 # incrementa 1 no numero de vizinhos
269
270         # verifica se ja existe linha a esquerda da proxima posicao
271         esquerda:
272                 li $t1, 0
273                 addi $t1, $t3, -4
274                 lw $t5, ($t1) # carrega o valor contido na posicao de memoria
275                 bne $t5, vermelho, direita
276                 addi $t2, $t2, 1 # incrementa 1 no numero de vizinhos
277
278         # verifica se ja existe linha a direita da proxima posicao
279         direita:
280                 li $t1, 0
281                 addi $t1, $t3, 4
282                 lw $t5, ($t1) # carrega o valor contido na posicao de memoria
283                 bne $t5, vermelho, retorno
284                 addi $t2, $t2, 1 # incrementa 1 no numero de vizinhos
285
```

(Figura 24)

A figura 25 mostra a label *retorno*. Essas 4 linhas de código fazem o retorno para a função labirinto no mesmo ponto onde a label *checa_loop* foi chamada dependendo de qual posição estamos tentando gerar.

```
286         # retorna para a funcao labirinto
287         retorno:
288                 beq $t0, 0, voltaZero # Esquerda
289                 beq $t0, 1, voltaUm   # Direita
290                 beq $t0, 2, voltaDois # Baixo
291                 beq $t0, 3, voltaTres # Cima
```

(Figura 25)

Volta pra labirinto.

Na figura 26 verificamos se o registrador *$t2*, que guarda a quantidade de vizinhos vermelhos, é igual a 1. Caso sim, pintamos, finalmente, a posição de vermelho e executamos novamente para encontrar a próxima posição.

```
sne $t6, $t2, 1
bne $t2, 1, setup
addi pos_atual, pos_atual, -4 # Subtrai 4 bits para ir para o quadrado da esquerda
sw vermelho, (pos_atual)
```

(Figura 26)

PASSO 3 - Gerar o Robô em uma posição aleatória do Bitmap:

Como já explicado anteriormente, a chamada do comando *syscall* com o código 42 gera um número aleatório, nesse caso de 0-255. Esse valor é armazenado em *$t2* e multiplicado por 4 nas linhas 427-428. Em seguida, na linha 429, é atualizado a posição atual somando o novo valor calculado com a posição inicial do Bitmap.

Para esse projeto, optou-se por eliminar as possibilidades do robô ser gerado dentro já no labirinto. Para impedir isso, é armazenado a cor da posição calculada anteriormente em *$t3* e comparada com a cor do labirinto. Caso seja o labirinto, é feito todo o procedimento anterior novamente, caso contrário, geramos o robô naquela posição.

```
414 pintaRobo:
415         #Procura posicao vazia
416         posicao:
417                 # Gera numero aleatorio de 0 a 255 e salva em pos_atual
418                 li $v0, 42
419                 li $a1, 256
420                 syscall
421                 move $t2, $a0
422
423                 # Atualiza posicao inicial de t0
424                 move $t0, $t1
425
426                 # Multiplica o numero por 4 (bits)
427                 add $t2, $t2, $t2
428                 add $t2, $t2, $t2
429                 add pos_atual, $t2, pos_inicial
430
431                 # Carrega valor da posicao
432                 lw $t3, (pos_atual)
433                 beq $t3, vermelho, posicao
434
435         sw azul, (pos_atual) # Pinta robo no mapa
436
437         jr $ra
```

(Figura 27)

PASSO 4 - Fazer o Robô encontrar a linha:

Com o mapa já montado e o robô gerado aleatoriamente, precisamos fazer com que ele procure o labirinto. Construímos a função *procura*, onde vai conter o código necessário para mover o robô até encontrar uma posição qualquer do labirinto.

Para garantir que o labirinto seja sempre encontrado foram definidos as seguintes regras de operação:

- Para cada posição, é verificado se existe alguma parte da linha na parte superior ou inferior da coluna.
- Inicialmente o robô sempre andará para a direita.
- Quando estiver na borda direita, o robô muda sua direção para a esquerda.
- Para cada passo que der é verificado se aquela posição é parte da linha.

As linhas 294-296 da figura 28 inicializam alguns registradores a serem utilizados, *$s5* irá pintar o mapa de preto para limpar os rastros do robô enquanto *$s4* armazena a última posição do mapa e *$t2* será usado depois para definir a direção horizontal do robô.

```
293  procura:
294          li $s5, 0x000000        # Cor Preta
295          li $t2, 0
296          addi $s4, $s1, 1023     # Ultima pos do mapa
297
298
299          veCol:                  # Verifica se tem labirinto na coluna
300          move $t0, $s0           # Posicao a ser verificada
301                  # Ve coluna acima
302                  veCima:
303                          addi $t0, $t0, -64      # Pos acima
304                          lw $t3, ($t0)           # Cor da pos
305                          # Condicoes de busca
306                          blt $t0, $s1, veBaixo   # Se esta na ultima posicao da coluna (Break)
307                          bne $t3, $s2, veCima    # Se n?o achar labirinto (Continua)
308                          j andaCima              # Achou labirinto
309
310                  # Ve coluna abaixo
311                  veBaixo:
312                          addi $t0, $t0, 64       # Pos abaixo
313                          lw $t3, ($t0)           # Cor da pos
314                          # Condicoes de busca
315                          bgt $t0, $s4, veLinha   # Se esta na ultima pos da coluna (Break)
316                          bne $t3, $s2, veBaixo   # Se nao achar labirinto (Continua)
317                          j andaBaixo             # Achou labirinto
318
```

(Figura 28)

A primeira regra de operação, que diz respeito a verificação da coluna, foi implementada como mostra na figura 28, usando duas funções: *veCima* e *veBaixo*. Ambas funções são semelhantes. Primeiro é calculado a posição acima/abaixo e carregado a cor correspondente, em seguida verifica se a posição existe dentro do Bitmap, linhas 306 e 315, caso exista, verifica se continua a busca ou se já encontrou o labirinto, linhas 307 e 316, caso não exista, termina a função.

Ao encontra o labirinto acima ou abaixo, é acionado a função *andaCima*/*andaBaixo*, que será explicado mais para frente.

A segunda e terceira regra são referentes as operações na horizontal e estão implementadas nas figuras 29 e 30. *$t2* indica o sentido do movimento (0 = direita, 1 = esquerda). Como *$t2* foi inicializado com 0 (ver linha 295, figura 28), a função *proxLinha* será chamada.

É feito uma verificação para garantir se o robô está ou não na borda. Todas as posições da borda direita são múltiplas de 64, logo basta verificar se o mod entre a posição e 64 é igual a zero. Isso pode ser visto nas linhas 329-337. Caso não seja borda, o robô anda e verifica se a posição é ou não um labirinto. Caso a posição seja borda, é chamado a função *voltaLinha*.

```
319         veLinha:
320                 beq $t2, 0, proxLinha
321                 beq $t2, 1, voltaLinha
322                 proxLinha:
323                         # Delay
324                         li $v0, 32
325                         li $a0, 500
326                         syscall
327
328                         # Calculando mod
329                         sub $t4, $s0, $s1       # indice da pos
330                         addi $t4, $t4, 4        # Ajuste
331                         div $t1, $t4, 64        # quociente
332                         mul $t1, $t1, 64        # multiplica
333                         sub $t1, $t4, $t1       # mod
334
335
336                         # Se estiver na borda
337                         beqz $t1, voltaLinha
338
339                         # Se nao for borda
340
341                         # Anda pra direita
342                         sw $s5, ($s0)           # Pinta de preto
343                         addi $s0, $s0, 4        # Prox pos
344                         lw $t3, ($s0)           # Carrega cor da prox pos
345                         sw $s3, ($s0)           # Pinta robo
346
347                         # Achou labirinto
348                         beq $t3, $s2, Saida
349
350                         # Nao achou labirinto
351                         j veCol
352
```

(Figura 29)

Ao iniciar a função *voltaLinha* o valor do registrador *$t2* é alterado para 1, indicando que a direção será sempre para a esquerda. Fora essa alteração, o funcionamento de *voltaLinha* ocore do mesmo modo que a função *veLinha* como mostra a figura 30.

```
353             voltaLinha:
354                     # Delay
355                     li $v0, 32
356                     li $a0, 500
357                     syscall
358                     addi $t2, $zero, 1
359
360                     # Anda para esquerda
361                     sw $s5, ($s0)           # Pinta de preto
362                     addi $s0, $s0, -4       # Prox pos
363                     lw $t3, ($s0)           # Carrega cor da prox pos
364                     sw $s3, ($s0)           # Pinta robo
365
366                     # Achou labirinto
367                     beq $t3, $s2, Saida
368
369                     # Nao achou labirinto
370                     j veCol
371
```

(Figura 30)

Voltando para a movimentação na vertical, a figura 31 mostra o código das funções *andaCima* e *andaBaixo,* ambas ocorrem após a certeza de que existe um pedaço do labirinto na direção correspondente. Desse modo, basta apenas atualizar a posição do robô para a posição adjacente, e verificar se já está no labirinto. Se não estiver a função é executada novamente. Se estiver a função de busca é encerrada, e o objetivo de encontrar o labirinto é alcançado.

```
372                 # Anda para cima ate achar labirinto
373                 andaCima:
374                         move $t0, $s0
375                         # Delay
376                         li $v0, 32
377                         li $a0, 500
378                         syscall
379
380                         # Anda para cima
381                         sw $s5, ($s0)           # Pinta de preto
382                         addi $s0, $s0, -64      # Prox pos
383                         lw $t3, ($s0)           # Carrega cor da prox pos
384                         sw $s3, ($s0)           # Pinta robo
385
386                         #Achou labirinto
387                         beq $t3, $s2, Saida
388                         j andaCima
389
390                 # Anda para baixo ate achar labirinto
391                 andaBaixo:
392                         move $t0, $s0
393                         # Delay
394                         li $v0, 32
395                         li $a0, 500
396                         syscall
397
398                         # Anda para baixo
399                         sw $s5, ($s0)           # Pinta de preto
400                         addi $s0, $s0, 64       # Prox pos
401                         lw $t3, ($s0)           # Carrega cor da prox pos
402                         sw $s3, ($s0)           # Pinta robo
403
404                         # Achou labirinto
405                         beq $t3, $s2, Saida
406                         j andaBaixo
```

(Figura 31)

PASSO 5 - Fazer o Robô percorre uma linha inteira:

Para o robô percorrer a linha, foram usados duas estruturas de memória, chamadas de *pilha1* e *pilha2*.

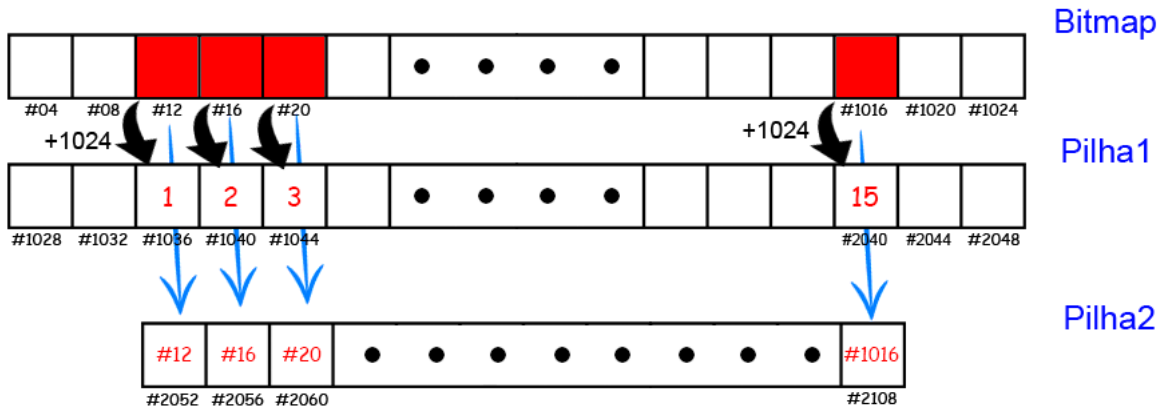
Considere como posição de memória números com hashtag ao lado, a figura 32 auxiliará na explicação do próximo passo. Além disso, por conveniência o *Bitmap* será mostrado em forma de vetor.

Como o *Bitmap* possui 256 quadrados, e cada um tem 4 bytes de tamanho, o início da *pilha1* se dá na posição inicial do *Bitmap* somado com (256x4 = 1024).

A estrutura *pilha1* possui o mesmo tamanho do Bitmap, logo para cada quadrado no Bitmap existe 1 quadrado na *pilha1* a uma distância de 1024 bytes, chamaremos esse par de quadrados correspondentes. Sempre que um quadrado do labirinto é pintado no Bitmap, o quadrado correspondente na *pilha1* armazena um índice indicando a ordem em que foi pintado.

O exemplo da figura 33, considera uma linha de tamanho 15, logo a posição #1016 do *Bitmap* foi o último bloco a ser gerado. O bloco da posição #2040 da *pilha1* armazena esse índice.

Por fim a *pilha2* contém os endereços ordenados do labirinto, logo o tamanho da *pilha2* será o mesmo que o labirinto.



(figura 32)

Já entendido a funcionalidade das estruturas de memória, veremos como elas foram utilizadas no código.

Ao iniciar a função percorre, o registrador *pilha2* contém a posição final da *pilha2*. Na figura 33 é calculado o tamanho da *pilha2* (posição final - posição incial + 4 bytes) e armazenado em $t4.

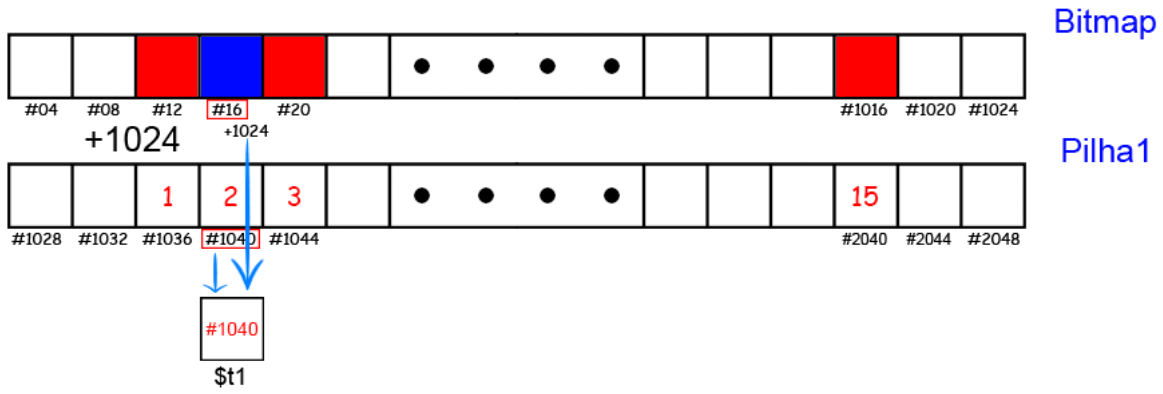
```
439  percorre:
440          move $s4, $ra
441
442          # Tamanho do labirinto x4
443          addi $t4, $s1, 1024     # Pos inicial pilha1
444          addi $t4, $t4, 1028     # Pos inicial pilha2
445          sub  $t4, pilha2, $t4
```

(figura 33)

Em seguida, a figura 34 mostra alguns parâmetros sendo ajustados. O registrador *$t0* armazena a posição final da *pilha2.* Já o registrador *$t1* na linha 450 recebe a posição atual do robô, em seguida na linha 451 é obtido a posição correspondente na *pilha1* e armazenado em *$t1.* Dessa forma é possível saber qual índice o robô se encontra. A figura 35 ilustra esse procedimento.

```
448         addi pilha2, pos_inicial, 2048  #Pos atual da pilha2
449         add $t0, pilha2, $t4            #Pos final da pilha2
450         move $t1, pos_atual
451         addi $t1, $t1, 1024             #Pos atual da Pilha1
```

(figura 34)

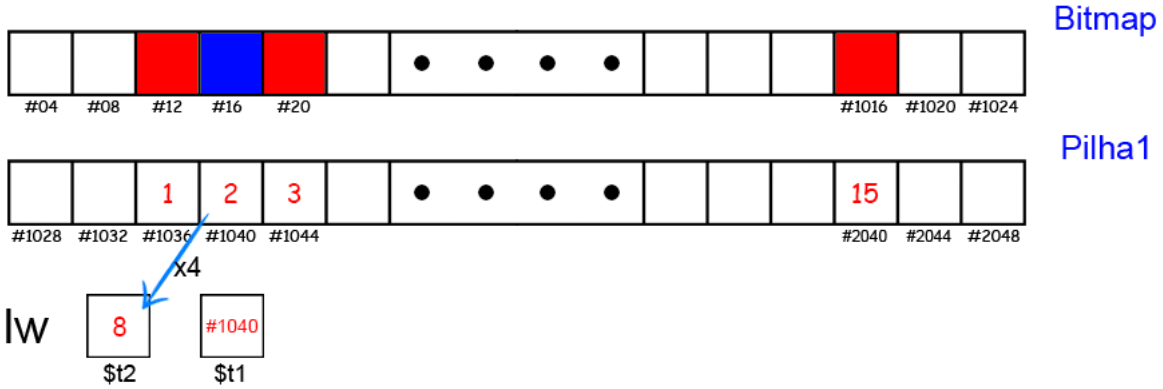


(figura 35)

Para carregar o índice basta pegar a informação da posição de memória armazenada em *$t1*. O número é ajustado com base nos 4 bytes. Esse procedimento ocorre na figura 36 e está exemplificado na figura 37.

```
453         #Calcula posicao da Pilha1
454         lw $t2, ($t1)               #Recupera o indice (dado da pilha1)
455         add $t2, $t2, $t2
456         add $t2, $t2, $t2
```

(figura 36)

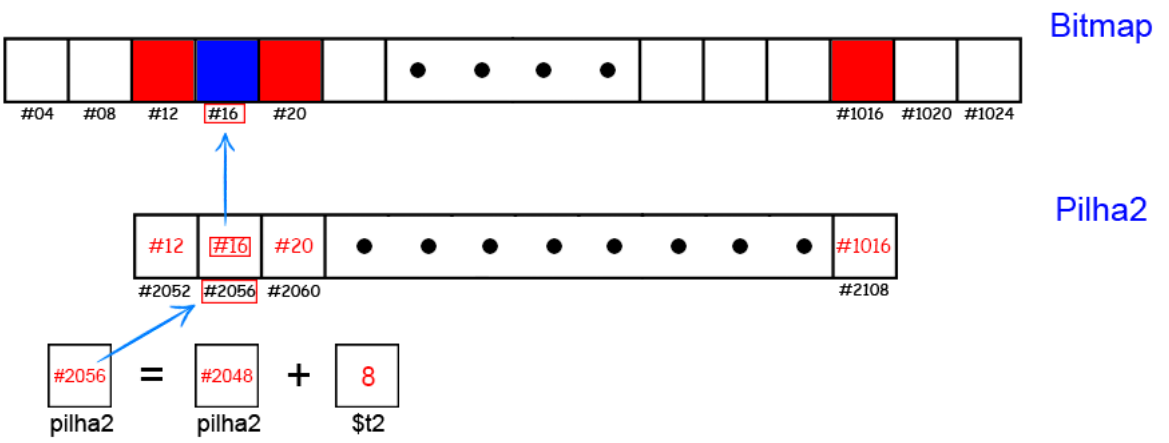


(figura 37)

O valor de $t2 somado ao valor inicial da *pilha2,* linha 459 da figura 38, indica a posição da *pilha2* em que o robô se encontra. A figura 39 mostra essa relação.

```
458         #Calcula posicao do Robo pela pilha 2
459         add pilha2, $t2, pilha2
```

(figura 38)



(figura 39)

Em seguida, o código da figura 40 chama a função *andaRobo* percorrendo a *pilha2* em um sentido, após isso, verifica se já está na última posição do labirinto, o que coincide com a última posição da *pilha2*, e por fim, chama a função andaRobo percorrendo a *pilha2* no sentido contrário. O registrador $t3 contém o dado que faz percorrer a pilha em um sentido ou em outro.

```
461         # Se estiver na ultima pos s faz camin de volta
462         beq pilha2, $t0, caminVolta
463         # Anda primeira metade
464         addi $t3, $zero, 4
465         jal andaRobo
466
467         caminVolta:
468         # Anda caminho de volta
469         addi $t3, $zero, -4
470         sub $t0, $t0, $t4
471         jal andaRobo
472
```

(figura 40)

Ao entrar na função *andaRobo,* mostrada na figura 41, é importante lembrar que $t0 contém a posição atual, e *$t3* possui o sentido que a pilha está sendo percorrida (4 para um sentido e -4 para o sentido oposto).

A função atualiza a posição do robô com os dados da *pilha2,* e anda uma quantia armazenada em *$t3* para a posição da pilha. Como *$t3* contém 4 ou -4, a próxima posição sempre será uma posição de memória adjacente. Isso ocorrerá em loop, até que se esteja em um dos limites da *pilha2*.

```
474         move $ra, $s4
475         jr $ra
476
477 # Robo anda quando encontra o labirinto
478 andaRobo:
479
480         # Calcula proxima posicao
481         add pilha2, pilha2, $t3
482
483
484         # Delay
485         li $v0, 32
486         li $a0, 500
487         syscall
488
489         # Anda
490         lw $t2,(pilha2)                 # Recupera posicao na pilha2
491         sw azul, ($t2)                  # pinta robo
492         sw vermelho, (pos_atual)        # pinta o mapa
493         move pos_atual, $t2             # atualiza pos atual
494         bne pilha2, $t0, andaRobo       # Verifica se terminou caminho
495         jr $ra
```

(Figura 41)

O código completo pode ser acessado no GitHub a partir deste link: https://github.com/BeatrizMartinsA/AOC/blob/master/Rob%C3%B4SeguidordeLinhaAssemblyMIPS.asm