ANDRÉ HENRIQUE MENDES - RA: 133576

GASPAR DE ANDRADE JUNIOR - RA: 133633

LUCAS MOREIRA DOS SANTOS - RA: 133682

TUTORIAL PARA CRIAÇÃO DE UM

ROBÔ SEGUIDOR DE LINHA

Arquitetura e Organização de Computadores

São José dos Campos - 2019

SUMÁRIO

1. **INTRODUÇÃO**

Assembly ou linguagem de montagem que é uma notação legível por humanos para o código de máquina que uma arquitetura de computador específica usa, utilizada para programar códigos entendidos por dispositivos computacionais, como microprocessadores e microcontroladores.

Tutorial é uma ferramenta de ensino/aprendizagem, que auxilia o processo de aprendizagem exibindo passo a passo o funcionamento de algo.
Utilizamos esses conceitos para demonstrar a criação de um robô segue linha no simulador MARS.

2. OBJETIVO

O objetivo deste tutorial é ensinar, de forma teórica e prática, a criação de um robô seguidor de linha utilizando a linguagem Assembly, a arquitetura MIPS, o simulador Mars e o Bitmap Display.

3. SOFTWARE/FERRAMENTAS UTILIZADAS:

3.1 Linguagem de programação Assembly: notação legível por humanos para o código de máquina que uma arquitetura de computador específica usa, utilizada para programar códigos entendidos por dispositivos computacionais, como microprocessadores e microcontroladores.

3.2 Arquitetura MIPS: arquitetura de processadores baseados no uso de registradores. As suas instruções tem à disposição um conjunto de 32 registradores para realizar as operações. Entretanto, alguns destes registradores não podem ser usados por programadores, pois são usados pela própria máquina para armazenar informações úteis. Processadores

MIPS são do tipo RISC (Reduced Instruction Set Computer - ou seja, Computadores com Conjunto de Instruções Reduzidas). Isso significa que existe um conjunto bastante pequeno de instruções que o processador sabe fazer. Combinando este pequeno número, podemos criar todas as demais operações.

3.3 Simulador Mars: ambiente de desenvolvimento interativo (IDE) para programação em linguagem assembly MIPS, destinada ao nível educacional.

3.4 Bitmap Display: recurso gráfico oferecido pelo simulador Mars baseada pixels. A próxima sessão esclarece melhor o funcionamento da ferramenta.

3.4.1 SOBRE O BITMAP DISPLAY:

O Bitmap Display é uma janela para exibição gráfica baseada em pontos. Suas principais características são as seguintes:

- Um ponto é conjunto de 1 ou mais pixels. O tamanho de um ponto é definido pelos parâmetros de *Unit Width in Pixels* e *Unit Height in Pixels* que definem, respectivamente, a largura do ponto em pixels e a altura do ponto em pixels.

- Resolução configurável, default 512 x 256 pixels.

- Cada ponto é representado por uma palavra de 32 bits (4 bytes), no formato RGB, um byte para cada cor. Red(vermelho) = 0x00FF0000, Green(verde) = 0x0000FF00, Blue(azul) = 0x000000FF. O byte mais significativo é ignorado.

- Essa ferramenta utiliza um display gráfico mapeado em memória, isto é, cada endereço de memória corresponde a um ponto.

- O endereço base de memória do Bitmap Display é configurável. Por exemplo, pode ser utilizado como endereço base de memória: global data, global pointer,static data, heap, memory map.

- Por default, o endereço base de memória aponta para o ponto superior esquerdo da tela.

- O desenho de um pixel na tela é realizado pela escrita de uma palavra contendo a descrição de sua cor RGB na posição de memória correspondente.

4. CONFIGURAÇÕES DO PROJETO

4.1 COMO CONFIGURAR O PROJETO:

- Após instalado o simulador MARS, abra-o, clique no menu *Tools* e selecione abra o *Bitmap Display.*

- A seguir clique em *Configuration* e defina as seguintes propriedades:
    - *Unidade de largura em pixels (Unit Width in Pixels):* 4
    - *Unidade de altura em pixels (Unit Height in Pixels):* 4
    - *Largura do display (Display Width in Pixels):* 256
    - *Altura do display (Display Height in Pixels):* 256
    - *Endereço base do display (Base address for display):* 0x10040000 (heap)

Figura 1 - Configurações do Bitmap

Utilizando essas configurações, o Bitmap Display será uma matriz de 512x512 pixels. Além disso, como cada ponto tem tamanho de tamanho 4 pixels de altura por 4 pixels de largura, o Bitmap Display será uma matriz de 64x64 pontos. Para cada escrita de um valor RGB (como descrito anteriormente) em um endereço de memória (começando no endereço de base 0x10040000), desenhamos um ponto no display.

4.2 FUNCIONAMENTO DO SIMULADOR

O robô é representado por um pixel vermelho que é posicionado aleatoriamente e deve procurar a linha para seguir, que também deve ser gerada aleatoriamente, sem posição fixa. Assim que o robô encontrar a linha, deve seguí-la até o final.

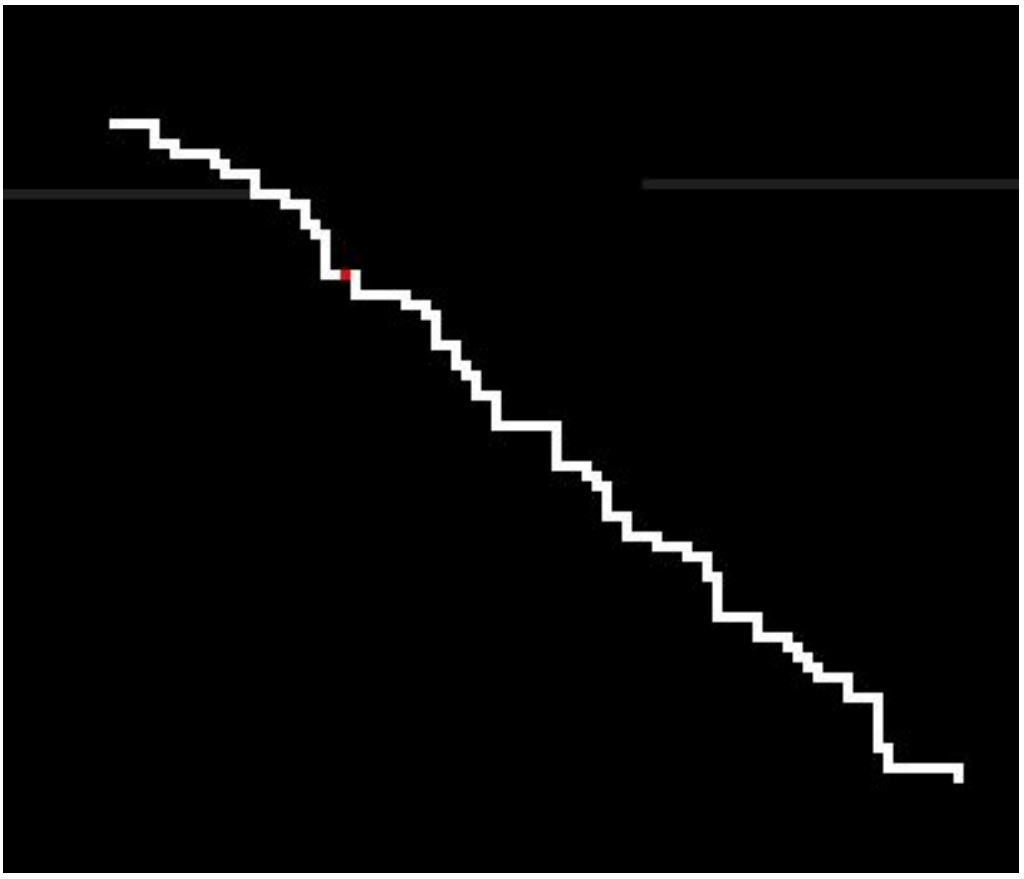

Figura 2 - Robô (vermelho), seguindo a linha branca

A distribuição de cores dos pixels é a seguinte:

5. CÓDIGO

O código-fonte completo do projeto está disponível em:
https://github.com/andrehenriquemendes/line-following-robot-assembly/blob/master/final-robot.asm
e também no final do tutorial.

Para o funcionamento correto e organização do programa, resumidamente, foram utilizadas as seguintes funções:

- ***set_tela*:** inicia todos os valores para o display.

- ***set_cores***: atribui cada cor a um registrador específico.

- ***set_linha***: gera uma linha em branco na tela, com formato aleatório.

- ***gera_posicao_inicia*l**: função que retorna um pixel aleatório na tela e atribui a um registrador específico.

- ***set_robo***: a função acessa o registrador da função *gera_posicao_inicial* e posiciona o robô inicialmente nesse pixel. A partir disso, a função faz o robô percorrer a tela até achar um pixel em branco (o que significa que achou a linha).

- ***segue_linha***: é chamada assim que o robô encontra a linha, e tem a função de seguir a linha em branco até o final.

- **the_end**: entra em ação assim que o robô chega ao final da linha, tem como função indicar que o simulador finalizou.

Definidas as funções, com o MARS aberto e o Bitmap display configurado começamos a programar

1) Inicialmente, atribuímos as cores a variáveis:

```
.data
        #Cores
        black:    .word 0x000000
        white:    .word 0xFFFFFF
        robo:    .word 0xFF0000
        rastro: .word 0x202020
        end: .word 0x9C9C9C
```

No código acima, atribuímos a variável *black* o valor 0x000000, que no padrão RGB de cores representa a cor preta. Isso se estende às demais variáveis.

2) Após inicializar os valores no segmento de dados, iniciamos o segmento de código:

```
.text
        j main
################################
# As funções entrarão aqui posteriormente
################################
        main:
```

- *.text*: as instruções ou dados inicializados depois dessa declaração serão colocados no segmento de código.
- *j main*: a instrução *j* (jump) pula incondicionalmente para a função *main.*

3) Dentro da função main podemos chamar as funções descritas (criaremos elas posteriormente, a medida que forem chamadas).

```
main:
        jal set_tela
        jal set_cores
        jal set_linha
        jal gera_posicao_inicial
        jal set_robo
        jal segue_linha
        jal the_end
```

- *jal*: instrução que chama a função. Por exemplo, *jal set_tela* chama a função *set_tela.*

4) Como a primeira função que é chamada em main é set_tela, criaremos essa função:

```
set_tela:
        addi $t0, $zero, 65536
        add $t1, $t0, $zero
        lui $t1, 0x1004
        j go_to_main
```

- *addi $t0, $zero, 65536*: atribui o inteiro 65536 ao registrador $t0.
  65536 = (512*512) / 4 pixels.
- *add $t1, $t0, $zero*: copia o conteúdo do registrador $t0 para o registrador $t1
- *lui $t1, 0x1004*: lui (Load upper immediate) o valor 0x1004 é deslocado 16 bits a esquerda e armazenado no registrador $t1
- *j go_to_main:* pula para a função go_to_main (porém ela ainda não está definida)

5) Como go_to_main ainda não está definida, vamos defini-la agora:

```
go_to_main:
        jr $ra
```

- Essa função simplesmente retorna para a função que a chamou.
- Relembrando: a função main chama a função *set_tela* através da instrução *jal set_tela*, essa, por sua vez, chama a função *go_to_main*, que simplesmente tem o trabalho de retornar para o main.

6) Seguindo no main, a função a ser chamada agora é set_cores. Vamos defini-la:

```
set_cores: # salva as cores nos registradores
        lw $s4, white
        lw $s5, robo
        lw $s6, rastro
        lw $s3, end
        j go_to_main
```

- Em *set_cores* simplesmente armazenamos cada cor em um registrador diferente. Por exemplo, a cor branca (white) é armazenada no registrador $s4.

7) Voltando ao main, a função *set_linha* agora é chamada, a implementação dela é a seguinte:

```
set_linha:
        add $t0, $zero, $t1 # Salva o endereço base da tela em $t0
        add $t0, $t0, 15500 # Indica o ponto de partida da linha

        addi $t2, $zero, 0 # Inicia em $t2 um contador

        loop_1:
                li $v0, 42 # Ler considerações abaixo
                li $a1, 2
                syscall
                move $s0, $a0

                beq $s0, 1, move_direita
                beq $s0, 0, move_baixo

                move_direita:
                        addi $t0, $t0, 4
                        sw $s4, ($t0)
                        addi $t2, $t2, 1
                        beq $t2, 150, go_to_main
                        j loop_1


                move_baixo:
                        addi $t0, $t0, 512
```

```
sw $s4, ($t0)
addi $t2, $t2, 1
beq $t2, 150, go_to_main
j loop_1
```

Considerações:

(1) li $v0, 42 *(2) li $a1, 2* *(3) syscall* *(4) move $s0, $a0*	Esse trecho de código representa um **syscall** (chamada de sistema) utilizada pelo MARS. Elas desempenham serviços específicos de entrada e saída.

A syscall acima gera um inteiro aleatório de 0 a 1 e armazena o valor em $a0.

- (1) 42 = gerar um inteiro aleatório
- (2) 2 = limite superior
- (4) Move o número gerado para o registrador $s0

Existem diversos tipos de chamadas de sistema, como printar um inteiro, abrir um arquivo, printar um inteiro etc.
Nesse link você pode ver algumas das mais importantes:
https://courses.missouristate.edu/KenVollmar/MARS/Help/SyscallHelp.html

Dentro do *loop_1*, com o número aleatório gerado, utilizamos para determinar uma direção para a linha, que pode ser para direita ou para baixo.

- *beq $s0, 1, move_direita*: Se o inteiro sorteado for igual a 1, então a linha se move para a direita
- *beq $s0, 0, move_baixo*: Se o inteiro sorteado for igual a 1, então a linha se move para baixo

Vamos entender como funciona a função *move_direita:*

- *addi $t0, $t0, 4:* Adiciona 4 bytes ao registrador base, isto é, como cada pixel tem 4 bytes, então se estamos por exemplo no endereço 0x10040000 e

somamos 4 bytes, então chegamos no endereço 0x10040004 (pixel que se localiza a **direita** do anterior).

- *sw $s4, ($t0)*: Acontece o store word, ou seja, o endereço do pixel armazenado em $t0 recebe o conteúdo do registrador $s4, que é o código RGB da cor branca (definimos isso na função set_cores)
- *addi $t2, $t2, 1:* adicionamos 1 ao contador
- *beq $t2, 150, go_to_main*: se número do contador for igual a 150, então encerramos por aqui o desenho da linha branca.
- *j loop_1:* Se o contador não for igual a 150, então voltamos para o loop_1, onde será novamente sorteado um número de 0 a 1, o qual indicará para qual sentido seguiremos, direita ou para baixo.

Para a função *move_baixo*, o raciocínio é o mesmo, com a diferença do deslocamento dos pixels. Ao invés de adicionarmos 4 bytes ao registrador, adicionaremos 512 bytes.

- *addi $t0, $t0, 512:* Como a largura do display é exatamente 512 bytes, ao adicionarmos 512 bytes, estaremos simplesmente pulando para a linha seguinte.

8) Após o contador chegar a 150, que indica o cumprimento final da linha, voltamos para o *main*, e então a função *gera_posicao_inicial* é chamada, que como o nome indica, gera a posição inicial para o robô iniciar o percurso.

```
gera_posicao_inicial:
        li $v0, 42 # gera um número aleatório
        li $a1, 10000 # de 0 a 9999
        syscall
        move $t8, $a0  # e atribui ao registrador $t8

        addi $t6, $zero, 4 # atribui o valor 4 ao registrador $t6
```

```
mul $t8, $t8,$t6 # multiplica o valor aleatório por 4
j go_to_main
```

O que acontece nessa função é o sorteio de um número de 0 a 9999, após o sorteio do número, este é multiplicado por 4, isto deve ser feito porque as posições de memória aumentam de 4 em 4 bytes. O valor é armazenado em $t8 e é utilizado na próxima função, set_robo.

9)

```
.set_robo:
        add $t0, $zero, $t1 # Salva o endereço base da tela em $t0

        add $t0, $t0, $t8 # Soma o valor sorteado ao endereço base para definir posição inicial do robô

        sw $s5, ($t0)  # O endereço do pixel $t0 recebe a cor do robô, $s5

        li $v0, 32 # Ler considerações abaixo
        add $a0, $a0, 1
        syscall
        add $a0, $zero, 0

        loop_direita:
                lw $t7, ($t0)
                beq $t7, $s4, go_to_main

                li $v0, 32
                add $a0, $a0, 100
                syscall
                add $a0, $zero, 0
```

```
sub $t5, $t0, 4

sw $s5, ($t0)
addi $t0, $t0, 4
sw $s6, ($t5)

j loop_direita
```

Considerações:

li $v0, 32 *add $a0, $a0, 1* *syscall* *add $a0, $zero, 0*	Novamente, se trata de uma syscall. Nesse caso, o número 32 indica um chamada de sistema para que o programa espere por um instante. O valor 1, atribuído ao registrador $a0, significa que esse instante é 1 milissegundo. Na última linha o registrador é zerado.

Após isso, o programa entra no *loop_direita*. Suas instruções são as seguintes:

- *lw $t7, ($t0)*: O pixel armazenado em $t0, que é o atual e também está na memória, é carregado para o registrador $t7.
- *beq $t7, $s4, go_to_main*: Se o pixel atual, armazenado em $t7, for igual ao pixel carregado em $s4, branco, então o robô achou a linha branca e deve voltar para o main. Se não, o programa segue para a instrução seguinte.
- As próximas 4 instruções fazem o programa esperar por 100 milissegundos

```
li $v0, 32
add $a0, $a0, 100
syscall
add $a0, $zero, 0
```

- *sub $t5, $t0, 4*: O registrador $t5 recebe ($t0-4) bytes, ou seja, a posição anterior (à esquerda) ao pixel atual. Essa informação nos interessa apenas

para formar o rastro do robô, ou seja, os pixels que já foram percorridos por ele na tela.

- *sw $s5, ($t0)*: O pixel atual recebe a cor $s5, vermelha, isto é, o robô.
- *addi $t0, $t0, 4*: O próximo pixel (à direita) é armazenado em $t0.
- *sw $s6, ($t5)*: A posição armazenada $t5, que é o pixel que acabou de ser percorrido pelo robô, recebe a cor $s6, cinza (cor do rastro).
- *j loop_direita*: volta ao *loop_direita*. Isso se repete até que a condição do *beq* (branch on equal) do loop seja satisfeita, isto é, quando o robô chegar na linha branca.

Assim que o robô encontra a linha, voltamos para a função *main* e então, jal *segue_linha* é chamada. Dividiremos essa função em duas partes

10) Relembrando: quando a função *segue_linha* é chamada, quer dizer que o robô encontrou a linha. Sendo assim, na primeira parte, o código deve determinar qual a direção que o robô deve seguir ao entrar na linha (para baixo ou para a direita). E na segunda parte, as funções *direta* e *baixo* devem ser chamadas para fazer o robô seguir nessa direção, com base na lógica desenvolvida na primeira parte.

Primeira parte

```
segue_linha:
        sub $t5, $t0, 4 # $t5 recebe o pixel anterior (rastro)
        sw $s6, ($t5) # $t5 recebe a cor do rastro
        sw $s5, ($t0) # $t0 pixel atual recebe o robô (nesse momento o robô entra na linha branca)

        segue_linha2:
        # Com o robô na linha branca, seguem as condições:
                addi $t0, $t0, 4 # $t0 recebe o pixel à direita
```

```
                lw $t5, ($t0) # $t5 recebe o valor de $t0 (pixel à direita)
                beq $t5, $s4, direita # se o pixel (à direita) for igual ao conteúdo do registrador $s4 (branco), então a função direita é chamada.

                sub $t0, $t0, 4 # $t0 recebe o valor anterior (posição atual do robô)
                addi $t0, $t0, 512 # $t0 recebe o pixel abaixo
                lw $t5, ($t0) # $t5 recebe o valor de $t0 (pixel abaixo)
                beq $t5, $s4, baixo # se o pixel (abaixo) for igual ao conteúdo do registrador $s4 (branco), então a função baixo é chamada.

                j go_to_main # Esse processo acontece até que nenhum dos beq's dessa função seja acionado, ou seja, não existe pixel branco nem à esquerda nem abaixo.
```

Resolvida a lógica de **qual** direção seguir, é hora de criarmos a lógica de **como** seguir a direção.

Segunda parte:

```
direita:
        li $v0, 32 # syscall para a função espere...
        add $a0, $a0, 100 #  ...por 100 milissegundos
        syscall
        add $a0, $zero, 0
        # Ler considerações abaixo
        sw $s5, ($t0)
        sub $t5, $t0, 4
        sw $s4, ($t5)

        j segue_linha2
```

```
baixo:
        li $v0, 32
        add $a0, $a0, 100
        syscall
        add $a0, $zero, 0

        sw $s5, ($t0)
        sub $t5, $t0, 512
        sw $s4, ($t5)

        j segue_linha2
```

Considerações:

Supondo que a função *segue_linha* determinou que o robô fosse para a direita, ou seja, o registrador $t0 andou 4 bytes (um pixel à direita) e a função direita foi chamada. Após a função esperar por 100 milissegundos, temos as seguintes instruções:

- *sw $s5, ($t0)*: A posição de memória armazenada no registrador $t0 recebe a cor do robô.
- *sub $t5, $t0, 4*: O registrador $t5 recebe o valor ($t0 - 4) bytes, ou seja, o pixel anterior (à esquerda).
- sw $s4, ($t5): A posição de memória armazenada no registrador $t5 recebe a cor branca (cor da linha), indicando graficamente que o pixel já foi percorrido pelo robô.
- *j segue_linha2*: Após isso, voltamos para *segue_linha2*, onde a regra de para qual direção seguir será novamente calculada.

Essa sequência de código é similar à função *baixo*, a única diferença é que o registrador $t5 armazenará o valor ($t0 - 512) bytes, ou seja, o pixel anterior (acima) quando o robô anda na vertical.

11) Após a função *segue_linha2* terminar sua execução e retornar para o main, quando o robô chegar ao final da linha, a função *the_end* é chamada.

```
the_end:
        add $t0, $zero, $t1
        addi $t0, $t0, 65536
        addi $t2, $zero, 0
        the_end2:
                sw $s3, ($t0)
                sub $t0, $t0, 4
                addi $t2, $t2, 4
                beq $t2, 65536, go_to_main
                j the_end2
```

- *add $t0, $zero, $t1*: $t0 recebe o endereço base do display (canto superior esquerdo).
- *addi $t0, $t0, 65536*: $t0 agora contém (endereço base + 65536) bytes, ou seja, $t0 está posicionado no último pixel do display (canto inferior direito).
- *addi $t2, $zero, 0*: Inicia um contador em 0.

Então, loop the_end2 entra em execução:

- *sw $s3, ($t0)*: O endereço do pixel atual, armazenado em $t0, recebe a cor $s4 (tela cinza que indica o final do programa).
- *sub $t0, $t0, 4*: $t0 armazena o endereço de memória anterior (à esquerda).
- *addi $t2, $t2, 1*: O contador $t2 é incrementado em 4 (bytes).
- *beq $t2, 65536, go_to_main*: Se o contador for igual a 65536, então voltamos para o *main*.
- *j the_end2*: Se não for igual a 65536, então voltamos para o laço *the_end2*. Nesse caso, a função inicialmente começa no final da tela, e decrementa de 4 em 4 bytes até chegar ao início da tela.

Assim que a condição *beq $t2, 65536, go_to_main* é satisfeita, voltamos ao main e o programa é então encerrado.

Relembrando, a ordem de chamada das funções dentro fica a seguinte:

```
main:
		jal set_tela
		jal set_cores
		jal set_linha
		jal gera_posicao_inicial
		jal set_robo
		jal segue_linha
		jal the_end
```

Portanto, o código-fonte do projeto do robô seguidor de linha fica o seguinte:

```
.data
	#Cores
	black:    .word 0x000000
	white:    .word 0xFFFFFF
	robo:   .word 0xFF0000
	rastro: .word 0x202020
	end: .word 0x9C9C9C

.text
	j main

	go_to_main
		jr $ra
```

```
set_tela: # Inicia todos os valores para a tela
        addi $t0, $zero, 65536
        add $t1, $t0, $zero
        lui $t1, 0x1004
        j go_to_main

set_cores:
        lw $s4, white
        lw $s5, robo
        lw $s6, rastro
        lw $s3, end
        jr $ra

set_linha:
        add $t0, $zero, $t1
        add $t0, $t0, 15500

        addi $t2, $zero, 0

        loop_1:
                li $v0, 42
                li $a1, 2
                syscall
                move $s0, $a0

                # faz a comparação
                beq $s0, 1, move_direita
                beq $s0, 0, move_baixo

                move_direita:
                        addi $t0, $t0, 4
```

```
                sw $s4, ($t0)
                addi $t2, $t2, 1
                beq $t2, 150, go_to_main
                j loop_1

        move_baixo:
                addi $t0, $t0, 512
                sw $s4, ($t0)
                addi $t2, $t2, 1
                beq $t2, 150, go_to_main
                j loop_1

gera_posicao_inicial:
        li $v0, 42
        li $a1, 10000
        syscall
        move $t8, $a0

        addi $t6, $zero, 4

        mul $t8, $t8,$t6
        j go_to_main



set_robo:
        add $t0, $zero, $t1

        add $t0, $t0, $t8

        sw $s5, ($t0)
```

```
        li $v0, 32
        add $a0, $a0, 1
        syscall
        add $a0, $zero, 0

        loop_direita:
                lw $t7, ($t0)
                beq $t7, $s4, go_to_main

                li $v0, 32
                add $a0, $a0, 100
                syscall
                add $a0, $zero, 0

                sub $t5, $t0, 4

                sw $s5, ($t0)
                addi $t0, $t0, 4
                sw $s6, ($t5)
                j loop_direita

segue_linha:
        sub $t5, $t0, 4
        sw $s6, ($t5)
        sw $s5, ($t0)

        segue_linha2:

                addi $t0, $t0, 4
                lw $t5, ($t0)
```

```
beq $t5, $s4, direita

sub $t0, $t0, 4

addi $t0, $t0, 512
lw $t5, ($t0)
beq $t5, $s4, baixo

j go_to_main

direita:
        li $v0, 32
        add $a0, $a0, 100
        syscall
        add $a0, $zero, 0

        sw $s5, ($t0)

        sub $t5, $t0, 4
        sw $s4, ($t5)

        j segue_linha2

baixo:
        li $v0, 32
        add $a0, $a0, 100
        syscall
        add $a0, $zero, 0

        sw $s5, ($t0)
```

```
                    sub $t5, $t0, 512
                    sw $s4, ($t5)
                    j segue_linha2

the_end:
        add $t0, $zero, $t1
        addi $t0, $t0, 65536
        addi $t2, $zero, 0
        the_end2:
                sw $s3, ($t0)
                sub $t0, $t0, 4
                addi $t2, $t2, 4
                beq $t2, 65536, go_to_main
                j the_end2

main:
        jal set_tela
        jal set_cores
        jal set_linha
        jal gera_posicao_inicial
        jal set_robo
        jal segue_linha
        jal the_end
```

6. CONCLUSÃO

Com o tutorial terminado podemos observar na prática como é feito e como funciona programas feitos em assembly, com essa visão prática conseguimos aprofundar os conhecimentos sobre a linguagem de programação assembly, a utilização do Software MARS e a ferramenta Bitmap..

7. REFERÊNCIAS

[1] https://pt.wikipedia.org/wiki/Linguagem_assembly

[2] https://courses.missouristate.edu/KenVollmar/MARS/Help/SyscallHelp.html

[3] https://courses.missouristate.edu/KenVollmar/MARS/

[4] https://pt.wikibooks.org/wiki/Programar_em_Assembly_com_GAS/Diretivas

[5] https://github.com/RaRoPe/Bitmap-Display-MARS/blob/master/Trabalho%201%20-%20Assembly%20MIPS.pdf

[6] https://pt.wikibooks.org/wiki/Introdu%C3%A7%C3%A3o_%C3%A0_Arquitetura_de_Computadores/O_que_%C3%A9_o_MIPS%3F