UNIVERSIDADE FEDERAL DE SÃO PAULO
INSTITUTO DE CIÊNCIA E TECNOLOGIA
UC ARQUITETURA E ORGANIZAÇÃO DE COMPUTADORES
Profª. Dra. Denise Stringhini
2º semestre de 2019

Robô Aspirador de Pó

Tutorial

Nome: Tales Cabral Nogueira **RA:133811**

Nome: Thiago da Silva Moreira **RA: 114834**

UNIVERSIDADE FEDERAL DE SÃO PAULO
INSTITUTO DE CIÊNCIA E TECNOLOGIA
UC ARQUITETURA E ORGANIZAÇÃO DE COMPUTADORES
Profª. Dra. Denise Stringhini
2º semestre de 2019

1. Descrição do jogo

Este trabalho tem por objetivo a apresentar em forma de tutorial como pode ser implementado um robô aspirador de pó automático em assembly MIPS utilizando a IDE MARS (MIPS Assembly and Runtime Simulator). A fim de simular como seria o real funcionamento do aspirador de pó automático em uma casa, o jogo se passa na planta de uma casa onde há móveis que são espalhados de maneira aleatória(a cada execução do programa), além disso, assim como no mundo real, essa planta possui paredes e móveis, desta maneira o robô deve ser capaz de passar por baixo dos móveis, mas não por baixo das paredes.

Para a implementação do jogo, um conjunto de regras devem ser seguidas:

A. Toda vez que o programa for executado, tanto os móveis quanto as paredes devem ser gerados de maneira aleatória.
B. Da mesma forma que A, sujeiras aleatórias devem ser colocadas na casa.
C. O deve robô andar de maneira aleatória respeitando os limites tanto das bordas quanto das paredes que são geradas
D. A execução deve terminar quando todas as sujeiras forem recolhidas

Essas condições servirão de base para a implementação do projeto. Vale ressaltar que o jogo terá uma interface gráfica que será feita também no MARS com o auxílio do Bitmap Display.

Nas próximas seções serão mostradas as ferramentas que serão utilizadas para a execução do projeto.

2. MIPS Assembly

O MIPS é uma Arquitetura de Conjunto de Instruções (Instruction Set Architecture - ISA), desenvolvida pela empresa MIPS Computer Systems, que hoje é chamada de MIPS Technologies. MIPS significa Microprocessor Without Interlocked Pipeline Stages (Microprocessador Sem Estágios Intertravados de Pipeline).

O Conjunto de Instruções é um dos elementos desse grande sistema, e é de extrema importância para a construção de qualquer sistema computacional.

É importante ressaltar que diferente das linguagens de alto nível que C, Java, Python entre outras, onde vários comandos(instruções) podem ser dados em um mesma linha, quando

se trabalha com linguagens da mais baixo nível como assembly MIPS por exemplo, deve-se ter em mente que os comandos que fazem muitas coisas de um única vez, quando traduzidos para assembly, devem ser fragmentados e instruções únicas, onde cada linha de código pode executar uma instrução.

A imagem abaixo mostra a diferença de implementação de um simples algoritmo que calcula a soma entre dois vetores *a[i]* e *b[i]* e armazena em *c[i]* implementado primeiro em C e depois em assembly MIPS. Mesmo sem saber o funcionamento de cada uma das instruções usadas, é possível notar que o aumento na quantidade de linha do código se dá devido a necessidade de fragmentar comandos em pequenas instruções. Ter isso em mente é de grande importância para poder trabalhar com linguagens de baixo nível.

```
void sumarray(int a[], int b[], int c[]) {
   int i;
   for(i = 0; i < 100; i = i + 1)
         c[i] = a[i] + b[i];
}
```

```
        addi    $t0,$a0,400     # beyond end of a[]
Loop:   beq     $a0,$t0,Exit
        lw      $t1, 0($a0)     # $t1=a[i]
        lw      $t2, 0($a1)     # $t2=b[i]
        add     $t1,$t1,$t2     # $t1=a[i] + b[i]
        sw      $t1, 0($a2)     # c[i]=a[i] + b[i]
        addi    $a0,$a0,4       # $a0++
        addi    $a1,$a1,4       # $a1++
        addi    $a2,$a2,4       # $a2++
        j       Loop
Exit:   jr      $ra
```

Para iniciar o projeto é necessário ter o conhecimento de algumas instruções e registradores que serão utilizados durante a implementação.

UNIFESP
UNIVERSIDADE FEDERAL DE SÃO PAULO
1933

UNIVERSIDADE FEDERAL DE SÃO PAULO
INSTITUTO DE CIÊNCIA E TECNOLOGIA
UC ARQUITETURA E ORGANIZAÇÃO DE COMPUTADORES
Profª. Dra. Denise Stringhini
2º semestre de 2019

A imagem abaixo mostra um tabela com algumas das diversas instruções que existem em assembly MIPS. As colunas mostram, respectivamente qual a categoria da instrução, o seu nome, um exemplo (com parâmetros), o seu significado e um comentário.

MIPS assembly language

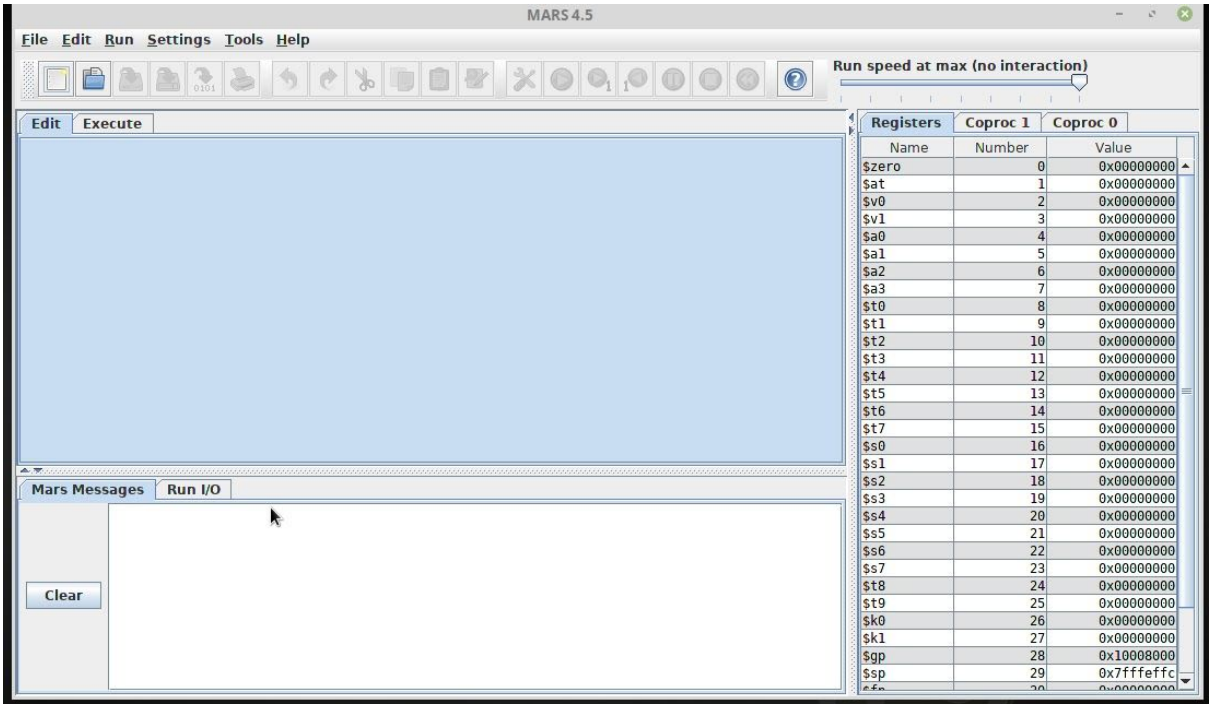
Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $1,$2,$3	$1 = $2 + $3	3 operands; exception possible
	subtract	sub $1,$2,$3	$1 = $2 – $3	3 operands; exception possible
	add immediate	addi $1,$2,100	$1 = $2 + 100	+ constant; exception possible
	add unsigned	addu $1,$2,$3	$1 = $2 + $3	3 operands; no exceptions
	subtract unsigned	subu $1,$2,$3	$1 = $2 – $3	3 operands; no exceptions
	add imm. unsign.	addiu $1,$2,100	$1 = $2 + 100	+ constant; no exceptions
	Move fr. copr. reg.	mfc0 $1,$epc	$1 = $epc	Used to get exception PC
	multiply	mult $2,$3	Hi, Lo = $2 ¥ $3	64-bit signed product in Hi, Lo
	multiply unsigned	multu $2,$3	Hi, Lo = $2 ¥ $3	64-bit unsigned product in Hi, Lo
	divide	div $2,$3	Lo = $2 ÷ $3, Hi = $2 mod $3	Lo = quotient, Hi = remainder
	divide unsigned	divu $2,$3	Lo = $2 ÷ $3, Hi = $2 mod $3	Unsigned quotient and remainder
	Move from Hi	mfhi $1	$1 = Hi	Used to get copy of Hi
	Move from Lo	mflo $1	$1 = Lo	Use to get copy of Lo
Logical	and	and $1,$2,$3	$1 = $2 & $3	3 register operands; logical AND
	or	or $1,$2,$3	$1 = $2 | $3	3 register operands; logical OR
	and immediate	andi $1,$2,100	$1 = $2 & 100	Logical AND register, constant
	or immediate	ori $1,$2,100	$1 = $2 | 100	Logical OR register, constant
	shift left logical	sll $1,$2,10	$1 = $2 << 10	Shift left by constant
	shift right logical	srl $1,$2,10	$1 = $2 >> 10	Shift right by constant
Data transfer	load word	lw $1,100($2)	$1 = Memory[$2+100]	Data from memory to register
	store word	sw $1,100($2)	Memory[$2+100] = $1	Data from register to memory
	load upper imm.	lui $1,100	$1 = 100 x 2^{16}	Loads constant in upper 16 bits
Conditional branch	branch on equal	beq $1,$2,100	if ($1 = = $2) go to PC+4+100	Equal test; PC relative branch
	branch on not eq.	bne $1,$2,100	if ($1!= $2) go to PC+4+100	Not equal test; PC relative
	set on less than	slt $1,$2,$3	if ($2 < $3) $1=1; else $1=0	Compare less than; 2's complement
	set less than imm.	slti $1,$2,100	if ($2 < 100) $1=1; else $1=0	Compare < constant; 2's comp.
	set less than uns.	sltu $1,$2,$3	if ($2 < $3) $1=1; else $1=0	Compare less than; natural number
	set l.t. imm. uns.	sltiu $1,$2,100	if ($2 < 100) $1=1; else $1=0	Compare < constant; natural
Unconditional jump	jump	j 10000	go to 10000	Jump to target address
	jump register	jr $31	go to $31	For switch, procedure return
	jump and link	jal 10000	$31 = PC + 4; go to 10000	For procedure call

Visto qual sucintamente como funciona um linguagem de baixo nível, podemos prosseguir para a próxima seção e conhecer um pouco sobre o ambiente onde o jogo será desenvolvido.

3. MARS

MARS é um ambiente de desenvolvimento integrado (IDE) destinado a estudos da arquitetura MIPS. A ferramenta MARS é um programa desenvolvido em Java, de código livre e que contém 155 instruções básicas da arquitetura MIPS, aproximadamente 370 pseudo-instruções, 17 funções *syscalls* para o console e entrada e saída de dados, outras 22 funções *syscalls* para outros usos.

A imagem abaixo mostra como a tela inicial do MARS



Nesta IDE, temos além dos campos que são comuns em qual em qualquer ambiente de programação, os campos para os registradores que permitem identificar o que cada registrador está armazenando em cada operação. Além disso, é importante mencionar ícone **(?) Help** que encontra-se no canto superior direito. Dúvidas com relação a instruções,

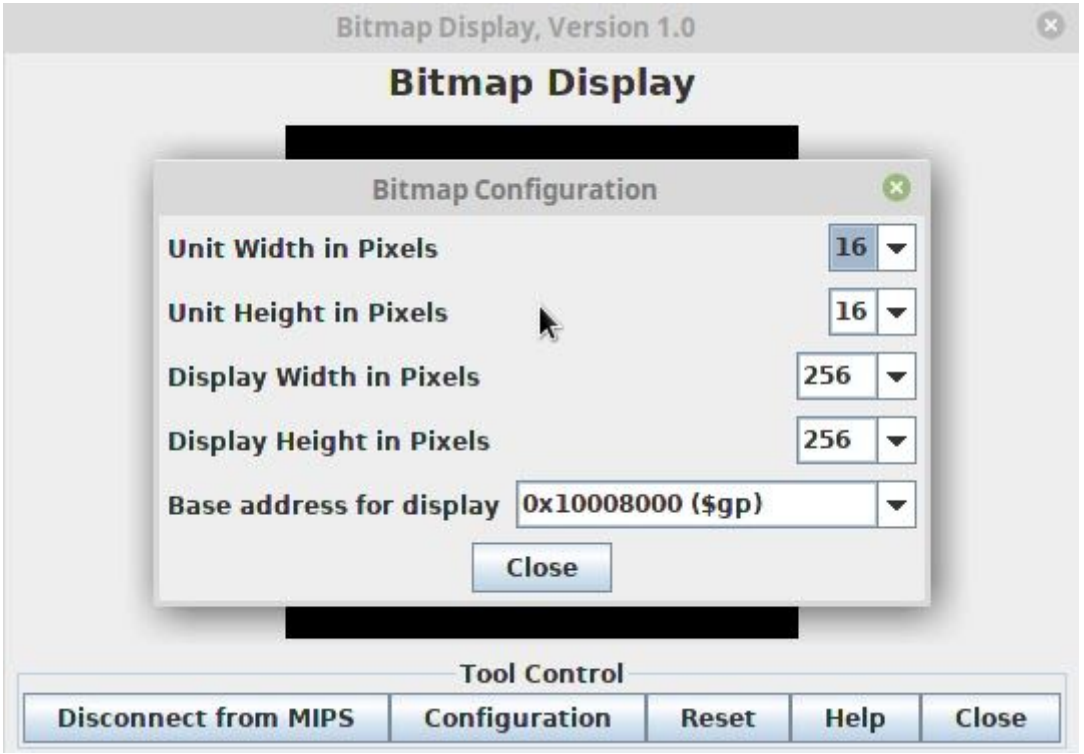
syscalls, exceções, macros e diretivas podem ser facilmente solucionadas com através desta ferramenta.

4. Bitmap Display

Agora que já temos o que é necessário saber para iniciar a implementação do jogo e qual será o ambiente utilizado para o seu desenvolvimento, é necessário definir quais serão as configurações do Bitmap Display(tela) onde o jogo será mostrado. É importante lembrar que o Bitmap Display é um grande vetor, e não uma matriz como muitos podem pensar.

Para configurar o Bitmap Display, basta ir em Tools → Bitmap Display → Configuration.

Para essa implementação as configurações escolhidas foram as seguintes:



Ou seja, 16x16 tamanho do pixel, tela com tamanho 256x256 e registrador base da tela ($gp).

Observação: como já mencionado, a tela não é uma matriz, mas sim um vetor, ou seja, quando dizemos que o tamanho é de 256x256, o que teremos na verdade é um vetor com

65.536(256x256) posições. Saber disso é de suma importância, uma vez que para desenhar na tela é necessário saber qual é a posição do registrador.

5. Implementação do jogo

Está seção mostra na prática como o jogo foi implementado, passo a passo. Os trechos de código que serão mostrados a podem ser copiados e rodados no MARS desde que as configurações do Bitmap sejam definidas.

A princípio, armazenamos as cores(representadas em hexadecimal) que serão utilizadas em todo o jogo. Os registradores $s0 a $s3 guardam essas cores.

```
#cores
li $s0, 0x00008b8b      #dark cyan - cenario
li $s1, 0x00ffd700      #gold - mobilias
li $s2, 0x00cd2626      #firebrick - aspirador
li $s3, 0x0087ceff      #skyblue - area limpa
```

Após salvar as cores nos seus respectivos registradores, atribuímos zero as registradores que serão utilizados para percorrer a tela.

```
li $s4, 0               #contador de areas vazias
li $s6, 0               #contador para as linhas
li $s7, 0               #contador para as colunas
```

A integração entre o código e a tela do Bitmap Display é feita através do registrador base já definido. Lembrando que o registrador base escolhido foi o ($gp)

```
move $s5, $gp           #move para o inicio do cenario
```

Após feitas as definições básicas, podemos iniciar a construção do cenário do jogo. O código abaixo mostra como os pixels são pintados para gerar o cenário do jogo.

```
cenario:
        up:
                sw $s0, 0($gp)    #memoria[0+$gp] = $s0 - o local da memoria recebe a cor dark cyan
                addi $gp,$gp,4    #locomove um bit a direita
                addi $s7,$s7,1    #adiciona uma coluna no contador
                bne $s7,16,up     #repete o procedimento enquanto a coluna for diferente de 16
        li $s7,0                  #zera o contador de colunas para o procedimento abaixo
        left:
                addi $s6,$s6,1    #adiciona uma linha no contador
                sw $s0,0($gp)     #memoria[0+$gp] = $s0 - o local da memria recebe a cor do cenario
                addi $gp,$gp,64   #locomove pela coluna esquerda como se percorresse
                bne $s6,15,left   #repete o procedimento ate atingir a ultima linha

        addi $gp,$gp,-60          #traz o ponto para a posicao
        down:
                addi $s7,$s7,1    #adiciona uma coluna no contador
                sw $s0,0($gp)     #memoria[0+$gp] = $s0 - local da memoria recebe a cor do cenerio
                addi $gp,$gp,4    #locomove um bit a direita
                bne $s7,15,down   #repete o procedimento enquanto a coluna for diferente de 15

        addi $gp,$gp,-4           #traz o ponto para a posicao
        right:
                addi $s6,$s6,-1   #remove uma linha no contador
                addi $gp,$gp,-64  #locomove pela coluna direita
                sw $s0,0($gp)     #memoria[0+$gp] = $s0 - local da memoria recebe a cor do cenario
                bne $s6,0,right   #repete o procedimento enquanto a coluna for diferente de 0

        addi $gp,$gp,-28          #move o gp para o meio
        li $v0,42                 #gera um numero aleatorio
        li $a1,12                 #ate 12
        syscall                   #faz a chamado do syscall para 42 para gerar um numero aleatorio
        add $a0,$a0,1             #adiciona +1 no numero aleatorio
        move $t0,$a0              #move o valor aleatorio para $t0

          divisoria:
                  addi $gp,$gp,64         #move para a linha de baixo
                  addi $s6,$s6,1          #adiciona mais uma linha no contador
                  beq $s6,$t0,divisoria   #caso a linha seja igual a $t0 retorna a divisoria
                  sw $s0,0($gp)           #pinta o bit com cyan
                  bne $s6,15,divisoria    #repete o procedimento ate 15 vezes
```

Logo após fazer o cenário e as divisórias que representam as paredes da planta da casa, é necessário gerar os móveis que serão colocados na casa, além disso, precisa-se verificar se esse móvel de pode de fato ser colocar no local que foi gerado de maneira aleatória.

O código abaixo mostra tanto como desenha os móveis na label *furniture* quanto como verificar se o móvel pode ser colocado no lugar na label *contar*.

```
#Moveis/Mobilia
furniture:
        lw $t1,0($gp)                   #$t1 = memoria[0+$gp]
        addi $gp,$gp,4                  #move um bit a direita
        beq $t1,$s0,furniture           #caso o t1 = $s0 (cenario) volta nos moveis

        li $v0,42                       #gera o numero aleatorio
        li $a1,10                       #ate 10
        syscall
        add $a0,$a0,1                   #chao 1
        move $t0,$a0                    #move o valor aleatorio para t0
        bne $t0,10,furniture            #caso o valor seja diferente de 15 retorna para moveis

        sw $s1,-4($gp)                  #pinta o bit
        ble $gp,0x10008400,furniture    #caso o valor seja menor volta para mobilia

move $gp,$s5                            #move para o inicio
#Verificar posicao
contar:
        lw $t1,0($gp)                   #$t1 = memoria[0+$gp]
        addi $gp,$gp,4                  #move um bit a direita
        beq $t1,$s0,contar              #caso o valor seja igual a parade retorna contar
        beq $t1,$s1,contar              #caso o valor seja igual a movel retorna contar
        addi $s4,$s4,1                  #adiciona mais um no contador
        ble $gp,0x10008400,contar       #caso o valor seja menor retorna para contar
```

Em sequência temos como é feito o robô que se move de aleatoriamente, respeitando as regras definidas inicialmente.

```
robot:
        move $gp,$s5                    #move o robo para o inicio
        li $v0, 42                      #gera um numero aleatorio
        li $a1, 255                     #ate 255
        syscall

        move $t0, $a0                   #move o valor para $t0
        mul $t0, $t0, 4                 #multiplica esse valor por 4
        add $gp, $gp, $t0               #move o bit para t0

        lw $t1,0($gp)                   #$t1 = memoria[0+$gp]

        bge $gp,0x10008400,robot        #caso o valor seja maior que o cenario retorna para o robo
        beq $t1,$s0,robot               #caso o valor seja igual ao cenario retorna
        beq $t1,$s1,robot               #caso o valor seja igual aos movel retorna
        sw $s2,0($gp)                   #colore o robo na posicao resultante
```

Por fim são definidas as direções nas quais o robô aspirador pode andar, este último pode andar para as seguintes direções: Norte, Sul, Leste, Oeste, Sudoeste, Sudeste, Nordeste e Noroeste. A ideia aplicada para decidir em qual direção o robô irá se mover é simples, basta verificar o valor armazenado no registrador *$t2* e decidir com *beq* qual caminho tomar.

```
direction:
        ble $s4,0,fim                   #caso valor do contador seja 0 ele encerra o programa
        li $v0,42                       #gera um num aleatorio
        li $a1,8                        #ate 8
        syscall
        add $a0,$a0,1                   #adiciona mais um no numero aleatorio
        move $t2,$a0                    #move o valor para $t2

        beq $t2,1,north                 #caso o valor seja 1 vai para o Norte
        beq $t2,2,northeast             #caso o valor seja 2 vai para o Nordeste
        beq $t2,3,east                  #caso o valor seja 3 vai para o oeste
        beq $t2,4,southeast             #e assim por diante
        beq $t2,5,south
        beq $t2,6,southwest
        beq $t2,7,west
        beq $t2,8,northwest
```

Abaixo temos o que ocorre quando a direção é escolhida pelo robô

```
north:
        move $t3,$gp            #move o valor de gp para t3 (aux)
        addi $t3,$t3,-64        #move o aux para cima
        lw $t1,0($t3)           #t1 = memoria[0+$t3]
        beq $t1,$s0,direction   #caso o valor bata num cenario retorna uma nova direcao
        beq $t1,$s1,direction   #caso o valor bata num imovel retorna uma nova direcao
        move $gp,$t3            #senao $gp recebe o valor de $t3
        beq $t1,$s3,N           #caso valor bata com uma area limpa entao vai para N
        addi $s4,$s4,-1         #tira um da area vazia
        N:
        sw $s2,0($gp)           #colore o aspirador na area do bit
        sw $s3,64($gp)          #colore a area que passou como limpa
        j north
```

```
    northeast:
            move $t3,$gp
            addi $t3,$t3,-60
            lw $t1,0($t3)
            beq $t1,$s0,direction
            beq $t1,$s1,direction
            move $gp,$t3
            beq $t1,$s3,printaNE
            addi $s4,$s4,-1
            printaNE:
            sw $s2,0($gp)
            sw $s3,60($gp)
            j northeast

    east:
            move $t3,$gp
            addi $t3,$t3,4
            lw $t1,0($t3)
            beq $t1,$s0,direction
            beq $t1,$s1,direction
```

```
southeast:
        move $t3,$gp
        addi $t3,$t3,68
        lw $t1,0($t3)
        beq $t1,$s0,direction
        beq $t1,$s1,direction
        move $gp,$t3
        beq $t1,$s3,printaSE
        addi $s4,$s4,-1
        printaSE:
        sw $s2,0($gp)
        sw $s3,-68($gp)
        j southeast

south:
        move $t3,$gp
        addi $t3,$t3,64
        lw $t1,0($t3)
        beq $t1,$s0,direction
        beq $t1,$s1,direction
        move $gp,$t3
        beq $t1,$s3,printaS
        addi $s4,$s4,-1
        printaS:
        sw $s2,0($gp)
        sw $s3,-64($gp)
        j south
```

```
southwest:
        move $t3,$gp
        addi $t3,$t3,60
        lw $t1,0($t3)
        beq $t1,$s0,direction
        beq $t1,$s1,direction
        move $gp,$t3
        beq $t1,$s3,printaSW
        addi $s4,$s4,-1
        printaSW:
        sw $s2,0($gp)
        sw $s3,-60($gp)
        j southwest

west:
        move $t3,$gp
        addi $t3,$t3,-4
        lw $t1,0($t3)
        beq $t1,$s0,direction
        beq $t1,$s1,direction
        move $gp,$t3
        beq $t1,$s3,printaW
        addi $s4,$s4,-1
        printaW:
        sw $s2,0($gp)
        sw $s3,4($gp)
        j west
```

```
northwest:
        move $t3,$gp
        addi $t3,$t3,-68
        lw $t1,0($t3)
        beq $t1,$s0,direction
        beq $t1,$s1,direction
        move $gp,$t3
        beq $t1,$s3,printaNW
        addi $s4,$s4,-1
        printaNW:
        sw $s2,0($gp)
        sw $s3,68($gp)
        j northwest
```

Por fim temos um comando para encerrar o programa

```
li $v0,10                    #comando que encerra o programa
syscall
```