Relatório MineSweeper

Arquitetura e Organização de Computadores – Ciência da Computação UFCA

Antônio José Monteiro Neto

Guilherme Viana Batista

●Função Play:

-Antes de qualquer coisa, foi usado a instrução ***save_context*** para poder usar os registradores em segurança. Após isso, foi feito o carregamento dos parâmetros através da instrução ***move***, onde em ***"$s0"*** está o endereço do tabuleiro, em ***"$s1"*** estão as linhas da matriz e em ***"$s2"*** estão as colunas.

```
save_context
move $s0, $a0
move $sl, $al
move $s2, $a2
```

- Após o carregamento dos parâmetros, devemos percorrer o tabuleiro. Para isso foram utilizadas três instruções em conjunto para descobrir a coordenada exata da posição do vetor, a ***sll***, ***add*** e ***lw***;

1- Utilizando sll: Para utilizar a ***sll*** tivemos que entender que ela funciona da seguinte maneira: ***sll destino, origem, quantidade***, onde ***quantidade*** é o número de posições que o valor será deslocado **para a esquerda**, ***origem*** é o registro que contém o valor a ser deslocado e ***destino*** é o registro onde o resultado do deslocamento será armazenado;

```
sll $tl, $sl, 5       #i*8*4
sll $t2, $s2, 2       #j*4
```

Assumindo que 1 "palavra" = 4 bits, e 1 linha = 8 "palavras", temos que na primeira linha do código ***"$s1"*** é deslocado 5 posições para a esquerda. Isso é feito para calcular o deslocamento correto da linha i no tabuleiro. Como cada palavra na memória é de 4 bits, e o tabuleiro tem 8 palavras por linha, multiplicar i por 8(tamanho de uma linha) e por 4 (tamanho de uma palavra) resulta na posição correta da linha em bits. Já ***"$s2"*** percorre as colunas da matriz;

2- Utilizando *add* e *lw*: O primeiro ***add*** foi utilizado para somar os valores do endereço armazenados em ***"$tl1"*** e ***"$tl2"*** e armazenar em ***"$t0".*** O segundo ***add*** foi utilizado para somar o endereço do tabuleiro com os endereços das linhas e colunas. Por fim, o ***lw*** foi utilizado para armazenar tudo em ***"$t4"***, a fim de liberar o espaço ocupado em ***"$t0";***

```
sll $t1, $s1, 5       #i*8*4
sll $t2, $s2, 2       #j*4

add $t0, $t1, $t2
add $t0, $t0, $s0
lw $t4, 0($t0)        #board[i][j]
```

-O próximo passo foi desenvolver o sistema de verificação, ou seja, se o jogador acertou uma bomba ou não. Para isso foram utilizadas as instruções ***beq (branch if equal)*** e ***bne (branch not equal),*** onde simulam os ifs do código em C, mas com a lógica inversa. Assim, foi feito o seguinte:

```
#Ifs aqui
beq $t4, -1, return0  #Se o player acertou uma bomba
bne $t4, -2, return1  #Se o player acertou uma casa já escolhida
```

Quando ***"$t4"*** for igual a –1 pula para a função ***return0*** e quando diferente de –2 ele pula para a função ***return1,*** onde quer dizer que todas as coordenadas já foram reveladas e, portanto, o jogador ganhou.

-Prosseguindo no código, chegamos num trecho para chamar a função **CountAdjacentBombs** através da instrução ***jal***:

```
addi $sp, $sp, -4  #Aloca espaço da pilha, assim liberando espaço para a função
sw $s0, 0 ($sp)  #Guarda o endereço do ponteiro $sp em $s0
move $a3, $t0         #Parametro para a função
jal countAdjacentBombs #Chamando a função
addi $sp, $sp, 4  #Libera o espaço alocado no início
sw $v0, 0($a3)        #Salva a quantidade de bombas encontradas no tabuleiro
```

-Após a chamada da função **CountAdjacentBombs,** o valor de ***"$v0"*** foi comparado a fim de verificar se é necessário chamar a função **revealNeighboringCells,** função essa que revela as coordenadas. Se for igual a 0 chama a função e se for diferente pula para função ***return1.***

```
bne $v0, $zero, return1 #Se tiver bomba perto ele retorna 1
addi $sp, $sp, -4 #Aloca espaço na memória para chamar a função
sw $s0, 0 ($sp) #Guarda o endereço do ponteiro $sp em $s0
jal revealNeighboringCells #Chama a função de revelar as casas
addi $sp, $sp, 4 #Libera o espaço alocado no início
```

-Por fim, as funções ***return1*** e ***return0***, que basicamente servem para alterar o valor que guardam em v0 que é o retorno da função play;

●Função CountAdjacentBombs:

-Primeiramente foi utilizado a instrução ***save_context*** e em seguida passado os parâmetros.

-Após isso, foi utilizada a instrução ***li*** para adicionar o valor imediato 0 no registrador ***"$v0".***

```
save_context
  move $s0, $a0
  move $s1, $a1 #Carregando parametros
  move $s2, $a2

  li $v0, 0
```

-Em seguida, antes de iniciar o laço de repetição, foi declarado um valor de i somando-se –1 no registrador que está guardado as linhas, ficando row – 1. Após declarar i, adicionamos 1 no registrador das linhas, ficando row + 1 e assim foi declarada mais uma variável para a condição do laço.

```
addi $s3, $s1, -1 # i = row - 1
addi $s4, $s1, 1 # row + 1
```

-Adiante, inicia-se o primeiro laço de repetição, denominado de **for_do_i.** No laço, através da instrução ***bgt (branch if greater than)*** foi verificado se i > row + 1, caso seja maior pula para a função **fim_for_do_i** e caso contrário ele executa o segundo laço denominado de **for_do_j.**

```
for_do_i:
bgt $s3, $s4, fim_for_do_i # Se o i > row+1

addi $s5, $s2, -1 # j = column - 1
addi $s6, $s2, 1 # column + 1
for_do_j:
bgt $s5, $s6, fim_for_do_j # Se o j > column + 1
```

-A primeira parte do **for_do_j** é semelhante ao **for_do_i**, mudando apenas a comparação. Caso j > column + 1 ele pula para função **fim_for_do_j** e caso contrário executa uma série de outras comparações.

```
for_do_j:
bgt $s5, $s6, fim_for_do_j # Se o j > column + 1

blt $s3, $zero, continue      # i>=0
bge $s3, SIZE, continue       # i<SIZE
blt $s5, $zero, continue      # j>=0
bge $s5, SIZE, continue       # j<SIZE

sll $t1, $s3, 5 #i*8*4
sll $t2, $s5, 2 #j*4

add $t0, $t1, $t2
add $t0, $t0, $s0
lw $s7, 0($t0)                # Carrega o elemento do board[i][j]

bne $s7, -1, continue         # board[i][j] == -1

addi $v0, $v0, 1              # count++
```

-Na imagem acima podemos ver diversas outras comparações que seguem uma lógica inversa das que estão no código em C. Caso todas sejam satisfeitas é utilizada a instrução ***addi*** como forma de cont++.

-Por último temos as duas últimas funções: **fim_for_do_i** e **fim_for_do_j.** Na função **continue** ela faz j++ e pula para o for do j, continuando o for. Já na função **fim_for_do_j** ela faz i++ e pula para o for o i. Finalmente, na função **fim_for_do_i** ela usa a função **restore_context** e sai da função **CountAdjacentBombs**.

```
continue:
addi $s5, $s5, 1              # j++
j for_do_j

fim_for_do_j:
addi $s3, $s3, 1              # i++
j for_do_i

fim_for_do_i:
restore_context
jr $ra
```

●Função RevealNeighboringCells:

-Antes de tudo **foi utilizado a instrução *save_context*** e em seguida passado os parâmetros.

-Após isso, foi utilizada a instrução ***li*** para adicionar o valor imediato 0 no registrador ***"$v0".***

```
save_context
move $s0, $a0
move $s1, $a1          # Carregando parametros
move $s2, $a2

li $v0, 0              # Iniciando V0 como 0
```

-Seguindo, são utilizadas as mesmas funções e argumentos que no **CountAdjacentBombs**, mas com o objetivo diferente. Objetivo é de, por meio da instrução ***bne (branch not equal)***, conferir se a coordenada adjacente foi realmente revelada.

```
addi $s3, $s1, -1 # i = row - 1
addi $s4, $s1, 1 # row + 1
for_do_i:
bgt $s3, $s4, fim_for_do_i # i > row + 1

addi $s5, $s2, -1 # j = column - 1
addi $s6, $s2, 1 # column + 1
for_do_j:
bgt $s5, $s6, fim_for_do_j # j > column + 1

blt $s3, $zero, continue       # i>=0
bge $s3, SIZE, continue        # i<SIZE
blt $s5, $zero, continue       # j>=0
bge $s5, SIZE, continue        # j<SIZE

sll $t1, $s3, 5 #i*8*4
sll $t2, $s5, 2 #j*4

add $t0, $t1, $t2
add $t0, $t0, $s0
lw $s7, 0($t0)                 # Carrega o elemento do board[i][j]

bne $s7, -2, continue          # Se o board[i][j] == -2
```

-Após conferir, é chamada a função **CountAdjacentBombs**. A coordenada só é revelada se a quantidade de bombas for 0. Então, em toda coordenada o count é acionado e confere se é > 0.

```
#Funcao countAdjacentBombs
move $a1, $s3
move $a2, $s5

addi $sp, $sp, -4
sw $s0, 0 ($sp)
move $a3, $t0
jal countAdjacentBombs         # Chama a função de contar os adjacentes
addi $sp, $sp, 4
sw $v0, 0($a3)                 # Coloca a quantidade de bombas na casa do tabuleiro
```

-Seguindo, temos uma parte responsável por verificar o valor retornado por o count. Se o valor for diferente de 0, existem bombas ao redor e não são reveladas coordenadas adjacentes. Se for igual a 0, significa que não existem bombas ao redor e coordenadas adjacentes são reveladas.

```
#Funcao de revelar as celulas
move $a1, $s3
move $a2, $s5

bne $v0, $zero, continue      # Se a casa tiver mais de 0 bombas ele não revela nada
addi $sp, $sp, -4
sw $s0, 0 ($sp)

jal revealNeighboringCells    # Senão revela os adjacentes
addi $sp, $sp, 4
```

-Por fim, temos a funções responsáveis por marcar o fim dos loopings internos.

```
continue:
addi $s5, $s5, 1 # j++
j for_do_j

fim_for_do_j:
addi $s3, $s3, 1 # i++
j for_do_i

fim_for_do_i:
restore_context
jr $ra
```

●Função CheckVictory:

-Para a **CheckVictory**, além do ***save_context,*** primeiro inicializamos um contador de coordenadas reveladas e um i para percorrer a matriz do tabuleiro.

```
save_context

# Inicializa o contador de coodenadas reveladas
  li $t0, 0

  # Percorre a matriz do tabuleiro
  li $t1, 0         # i = 0
```

-Após isso, abrimos o laço de repetição **i_for**, onde ele vai verificar, através do ***bge (branch if greater or equal)*** se o i >= SIZE. Caso a condição seja satisfeita, pula para **fim_for_do_i** e sai do laço. Caso a condição não seja satisfeita, é criado um j e o laço de repetição **j_for**.

```
i_for:
  bge $t1, SIZE, fim_for_do_i # Se i >= SIZE, sai do loop

  li $t2, 0          # j = 0
j_for:
  bge $t2, SIZE, j_for_end  # Se j >= SIZE, termina o loop interno
```

-Dentro do laço j_for é feita outra comparação: j >= SIZE. Caso a condição seja satisfeita, pula para **fim_for_do_j** e sai do laço. Caso a condição não seja satisfeita, começa a ser calculado o indicie da coordenada atual, com o intuito de verificar se ela já foi revelada ou não.

```
j_for:
  bge $t2, SIZE, j_for_end  # Se j >= SIZE, termina o loop interno

  # Calcula o índicie da coodenadas atual
  sll $t3, $t1, 5    # i * 8 * 4
  sll $t4, $t2, 2    # j * 4
  add $t5, $t3, $t4  # t5 = índice da coodenadas
  add $t5, $t5, $a0  # t5 = endereço da coodenadas

  lw $t6, 0($t5)     # Carrega o valor da coodenadas
  bge $t6, 0, coord_nao_revelada  # Se o valor da coodenadas < 0, continua a verificação

  # Incrementa o contador se a coodenadas foi revelada
  addi $t0, $t0, 1
```

-Após calcular o valor do indicie da coordenada atual, verifica, através do ***bge (branch if greater or equal)***, se o indicie >= 0. Caso a condição seja satisfeita, pula para a função **coord_nao_revelada**. Caso indicie < 0, continua a verificação, incrementando 1 ao contador criado no começo da **CheckVictory**.

-Entrando na função **coord_nao_revelada**, temos que será incrementado 1 ao j criado quando a condição da função **i_for** não foi satisfeita e pulando pro **j_for** com esse valor. Isso ocorre até satisfazer a condição do **j_for**.

```
coord_nao_revelada:
  addi $t2, $t2, 1  # Incrementa j
  j j_for       # Volta para o início do loop j
```

-Para finalizar, temos a funções responsáveis por marcar o fim dos loopings internos. Onde, caso seja retornado 1, o jogo é finalizado com a vitória do jogador.

```
j_for_end:
  addi $t1, $t1, 1  # Incrementa i
  j i_for       # Volta para o início do loop i

fim_for_do_i:

  beq $t0, BOMB_COUNT, return1

  li $v0, 0
  restore_context
  jr $ra

  return1:
  li $v0, 1
  restore_context
  jr $ra
```