

Relatório de OCD

Data de Entrega: 16/07/2020

Mikael Gi Sung Shin, 10843441

1. Entendendo MARS (MIPS Assembler and Runtime Simulator)

1.1 Introdução

Como o próprio nome já diz, MARS é um simulador e montador (do inglês “assembler”, responsável por converter determinadas instruções em linguagem de montagem para a linguagem equivalente da máquina) de MIPS, uma arquitetura de processadores baseado na utilização de registradores. Esse programa permite simular o MIPS executando o programa implementado em assembly, que é linguagem de montagem.

1.2 Estrutura do código:

A estrutura é basicamente definida por 2 seções, onde a primeira, `.data`, recebe as declarações das variáveis e a segunda, `.text`, contém as instruções do programa, e por “labels” (rótulos) que são pedaços de código, com intuito de utilizá-los em algum momento da execução do programa, através da chamada pelo nome ou rótulo que foi dado pelo programador. Segue abaixo, um molde de como a estrutura é:

```
.data
```

```
<nome da variável>: <tipo (directive)> <dado>
```

```
.text
```

```
<instrução> <register A>, <register B>, <serviço/label/op imediato>
```

Nota: O trecho de código acima, exemplifica como a linha de instrução é escrita. Como existem várias instruções, cada uma delas define o que será inserido ou não junto com ela (se devemos inserir ou não mais de um registrador, uma label, entre outros).

```
.text
```

```
nomeLabel:
```

```
<instrução 1> <register A> <dado>
```

```
<instrução 2> <register B> <nº serviço>
```

```
<instrução 3> <outraLabel>
```

1.3 Exemplos

1.3.1 Impressão de dados

```
.data
    nome: .ascii "Mikael Gi Sung Shin "
    nUsp: .word 10843441

.text

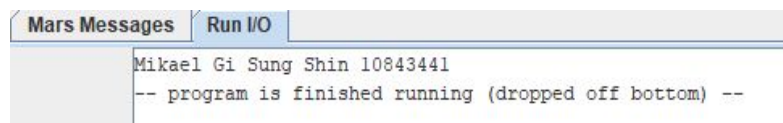
    li $v0, 4
    la $a0, nome
    syscall

    li $v0, 1
    lw $a0, nUsp
    syscall
```

O código acima declara variáveis, mais comumente conhecidas como String (.ascii) e inteiro (.word) no campo .data. E no campo .text, contém as instruções de “printar” essas duas variáveis.

Explicando com mais detalhes, no primeiro bloco da instrução, o li (load immediate) armazena o serviço de número 4, “printar” uma String, no registrador \$v0. O la (load address) armazena a variável nome dentro do registrador \$a0. Com a chamada do syscall, o controle é transferido para o sistema operacional, que executa o serviço solicitado. No segundo bloco, as instruções ocorrem de modo similar, porém, agora, como se trata de um inteiro, e não mais de uma String, faremos as devidas alterações. O número do serviço passa a ser 1 (“printar” inteiro), e, a instrução agora é carregar, não mais um endereço, mas sim um inteiro (word), por isso altera-se para lw.

Apertando a tecla F3, a aba de execução é aberta e o nosso programa é montado, e com F5, ele será executado:



```
Mars Messages Run I/O
Mikael Gi Sung Shin 10843441
-- program is finished running (dropped off bottom) --
```

Uma funcionalidade interessante é o uso de F7, pois nos mostra as instruções e suas alterações nos registradores do começo ao fim, uma por uma. No “Source” da tabela abaixo, mostra as linhas de código do programa em pseudo-instruções, que é maneira mais compacta do programador se referir às instruções. Já no “Basic”, ele retrata as instruções de uma forma mais completa em relação ao pseudo, podendo gerar outras instruções básicas (como é o caso do la e lw, que existem 2 para cada uma). O “Bkpt” se refere ao “breakpoint” que é o ponto onde o programador não quer executar. No “Code” representa a determinada instrução em MIPS num número binário de 32-bit. Por fim, percebe que em “Address” os números em hexadecimal são incrementados de 4 em 4, isso se deve ao fato do contador de programa atribuir um endereço de 4 bytes para cada linha de instrução.

Edit		Execute		
Text Segment				
Bkpt	Address	Code	Basic	Source
<input type="checkbox"/>	0x00400000	0x24020004	addiu \$2,\$0,0x00000004	8: li \$v0, 4
<input type="checkbox"/>	0x00400004	0x3c011001	lui \$1,0x00001001	9: la \$a0, nome
<input type="checkbox"/>	0x00400008	0x34240000	ori \$4,\$1,0x00000000	
<input type="checkbox"/>	0x0040000c	0x0000000c	syscall	10: syscall
<input type="checkbox"/>	0x00400010	0x24020001	addiu \$2,\$0,0x00000001	12: li \$v0, 1
<input type="checkbox"/>	0x00400014	0x3c011001	lui \$1,0x00001001	13: lw \$a0, nUsp
<input type="checkbox"/>	0x00400018	0x8c240018	lw \$4,0x00000018(\$1)	
<input type="checkbox"/>	0x0040001c	0x0000000c	svscall	14: svscall

Alterações de cada linha de instrução:

0) Nenhuma instrução

Registers		
Name	Number	Value
\$zero	0	0x00000000
\$at	1	0x00000000
\$v0	2	0x00000000
\$v1	3	0x00000000
\$a0	4	0x00000000
pc		0x00400000

1) Instrução básica: `addiu $2, $0, 0x00000004`

- “setar” em \$2 (o mesmo que \$v0), o \$zero (que não armazena nada) mais o serviço que foi delegado

\$zero	0	0x00000000
\$at	1	0x00000000
\$v0	2	0x00000004
\$v1	3	0x00000000
\$a0	4	0x00000000
pc		0x00400004

2) Instrução básica: `lui $1, 0x00001001`

- “setar” em \$1 (o mesmo que \$at), registrador reservado pelo montador, esse último número binário em ordem decrescente (inverter a ordem de cada bit)

\$zero	0	0x00000000
\$at	1	0x10010000
\$v0	2	0x00000004
\$v1	3	0x00000000
\$a0	4	0x00000000
pc		0x00400008

3) Instrução básica: `ori $4, $1, 0x00000000`

- “setar” em \$4 (o mesmo que \$a0), o número correspondente no registrador \$at.

\$zero	0	0x00000000
\$at	1	0x10010000
\$v0	2	0x00000004
\$v1	3	0x00000000
\$a0	4	0x10010000
pc		0x0040000c

4) Instrução básica: `syscall`

- executando o serviço (“printando” a variável nome)

\$zero	0	0x00000000
\$at	1	0x10010000
\$v0	2	0x00000004
\$v1	3	0x00000000
\$a0	4	0x10010000
pc		0x00400010

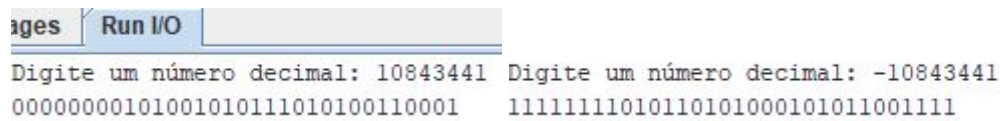
Notas: Em cada instrução, o pc (program counter ou contador de programa) é incrementado de 4 em 4. Ocorre de maneira similar no segundo bloco de código, com algumas alterações devido à utilização de um outro tipo de dado.

1.3.2 Conversor de decimal para binário

O programa abaixo converte o número decimal escolhido pelo usuário para binário com o máximo de 32 bits, sendo 31 deles alocados para o número propriamente dito ($2^{31} - 1$ em decimal) e 1 referindo-se ao seu sinal. Caso o valor “inputado” seja negativo, a saída fornecerá o binário em complemento de 2. Cada linha ou bloco de instrução estão comentados (as palavras em verde localizadas depois do “#”), para facilitar o entendimento do código.

```
1  .data
2      msg: .asciiz "Digite um número decimal: "
3      num: .asciiz ""
4  .align 2
5
6  .text
7
8  .globl start
9  start:
10     li $v0, 4          #
11     la $a0, msg        #
12     syscall            ## imprimindo a msg e
13     li $v0, 5          ## armazenando o valor
14     syscall            #
15     move $a0, $v0      #
16
17     jal init           # salta para a label init, guardando o link
18
19     li $v0, 11         #
20     li $a0, 10         ## gerando uma nova linha
21     syscall            #
22
23     j start            # chamada ao start, reiniciando todo o processo
24
25  init:
26     add $t0, $zero, $a0 # $t0 recebe o valor decimal
27     add $t1, $zero, $zero # $t1 recebe 0
28     addi $t3, $zero, 1    # $t3 recebe 1
29     sll $t3, $t3, 31      # desloca essa lógica para esquerda
30     addi $t4, $zero, 32   # $t4 recebe 32 (contador do loop)
31  loop:
32     and $t1, $t0, $t3     # $t1 recebe o resultado de ($t0 e $t3)
33     beq $t1, $zero, print # se $t1 = 0, salta para a label print A
34
35     add $t1, $zero, $zero #
36     addi $t1, $zero, 1    ## $t1 recebe 1
37
38     j print               # chama a label print
39  print:
40     li $v0, 1            #
41     move $a0, $t1        ## printa o inteiro guardado (1 ou 0)
42     syscall              #
43
44     srl $t3, $t3, 1      # descoloca em 1 a lógica
45     addi $t4, $t4, -1    # decrementa em 1 o contador
46     bne $t4, $zero, loop # se o contador NÃO for 0 continua o loop,
47                          # senão retorna e salta para a linha 21
48     jr $ra               # retorna
```

Ao executarmos, a sua saída será (exemplo da direita, nº positivo, esquerda, nº negativo):



```
pages Run I/O
Digite um número decimal: 10843441 Digite um número decimal: -10843441
00000000101001010111010100110001 1111111101011010101000101011001111
```

2. Relacionando a linguagem C com Assembly

2.1 Introdução

Antes de entrar nos comandos equivalentes das duas linguagens, é importante ressaltar que elas não possuem o mesmo nível. Enquanto que Assembly é uma linguagem de baixo nível, isto é, que se assemelha mais com a linguagem da máquina cujo os símbolos são uma representação direta do código de máquina, C é uma linguagem de médio nível, que não só possuem símbolos que são uma representação direta do código de máquina, como também símbolos complexos que são convertidos por um compilador, tornando mais fácil a sua compreensão.

2.2 Comandos

Apresentação de comandos similares entre as duas linguagens

- Instruções aritméticas:

Descrição	Assembly	C
Adicionar	add	+
Subtrair	sub	-
Definir em menos que	slt	(n1 > n2), devolvendo true (1) ou false (2)
AND	and	&
OR	or	
XOR	xor	^

- Instrução de deslocamento

Descrição	Assembly	C
Deslocamento à direita aritmético	sra	>>

- Instruções de multiplicação e divisão

Descrição	Assembly	C
Multiplicar	mult	*
Dividir	div	/

- Instruções de salto e desvio

Descrição	Assembly	C
Saltar com ligação	jal	soma(int n1, int n2)
Desviar quando igual	beq	if/while(n == 0)
Desviar quando não igual	bne	if/while(n != 0)
Desviar quando menor ou igual a zero	blez	if/while(i <= 0)
Desviar quando maior que zero	bgtz	if/while(i > 0)
Desviar quando menor que zero	bltz	if/while(i < 0)
Desviar quando maior ou igual a zero	bgez	if/while(i >= 0)

- Instrução de armazenamento

Descrição	Assembly	C
Armazenar inteiro	sw	int n = 0
Armazenar byte	sb	char c = "c";

2.3 Exemplos

Comparando exemplos de código em C e Assembly.

2.3.1 Salto, armazenamento e comparação

Em C:

```

1  if ( a < b )
2  n = n1 + n2;
3  else
4  n = n1 - n2;
```

Em Assembly:

```
# Considerando:  n = $s0,  n1 = $s1,  n2 = $s2,  a = $s3,  b = $s4.

1      slt $t0, $s3, $s4      # guarda em $t0, 1 (true) se $s3 < $s4
2      bne $t0, $zero, true    # se ($t0 != 0), executar o label "true",
3                                # senão, executar a linha seguinte.
4      add $s0, $s1, $s2      # n = n1 + n2; (se $t0 == 0)
5      j    jump              # salto para jump
6      true: sub $s0, $s3, $s4 # n = n1 - n2; (se $t0 != 0)
7      jump: jr $ra
```

2.3.2 Laços

Em C:

```
1      void naUspCincoEhDez(int[] notas, int nAlunos){
2          for(cont = 0; cont < nAlunos; cont++){
3              if(notas[cont] >= 5)
4                  notas[cont] = 10;
5          }
6      }
7      // um pouco de humor no trabalho :)
```

Em Assembly:

```
1      naUspCincoEhDez:
2          addi $s0, $zero, 0      # $0, que é o cont, = 0
3          addi $s1, $zero, 5      # $s1 = 5
4          addi $s2, $zero, 10     # $s2 = 10
5      loop:
6          slt $t0, $s0, $a1      # $t0 = 1, se ($s0 i) < ($a1 nAlunos)
7          beq $t0, $zero, exit    # chama label exit, caso $t0 == 0
8          add $t0, $a0, $s0      # $t0 = notas[0] + cont
9          lw  $t1, 0($t0)        # $t1 = notas[cont]
10         slt $t2, $s1, $t1      # $t2 = 1, se 5 < notas[i]
11         bne $t2, $zero, salto   # se $t2 != 0, salta para a label else
12         sw  $s2, 0($t0)        # notas[cont] = 10
13         j    salto
14     salto:
15         addi $s0, $s0, 1        # cont++
16         j    loop              # salta de volta para o loop
17     exit:
18         jr   $ra               # return
```

2.3.3 Procedimentos: Sequência de Fibonacci

A sequência de Fibonacci é infinita e tem dois elementos iniciais que são o zero, depois o um. Os números seguintes são sempre a soma dos dois números anteriores. Portanto, depois de 0 e 1, vem 1, 2, 3, 5, 8, 13, 21, 34, 45, e assim por diante. Os algoritmos abaixo iniciam com o elemento 1, ao invés do 0.

Em C (recursivamente):

```
1  #include <stdio.h>
2
3  int fibo(int n) {
4
5      if (n == 1) return 1;
6      else
7          if (n == 2) return 1;
8          else return fib(n - 1) + fib(n - 2);
9  }
10
11 int main() {
12
13     int n = 20;
14     printf("Os 20 primeiros elementos de Fibonacci são:\n");
15     printf("%d\n ", fibo(n));
16     return 0;
17 }
```

Em Assembly:

```
1  .data
2      fibo: .word 0 : 20      # vetor que conterá os valores da sequência
3      tam:  .word 20          # tamanho do vetor
4  .text
5      la    $t0, fibo         # carregamento de endereço do vetor
6      la    $t5, tam          # carregamento do endereço da variável tam (tamanho)
7      lw    $t5, 0($t5)       # carregamento do tamanho do vetor
8      li    $t2, 1            # o 1 é o 1º e o 2º elemento do arranjo
9      sw    $t2, 0($t0)       # fibo[0] = 1
10     sw    $t2, 4($t0)        # fibo[1] = fibo[0] = 1
11     addi   $t1, $t5, -2      # $t1 armazena o contador para o loop, executará (tam
12                                # - 2) vezes, pois o 1º e o 2º já foram definidos
13 loop:
14     lw     $t3, 0($t0)       # retorna o valor de fibo[n]
15     lw     $t4, 4($t0)       # retorna o valor de fibo[n+1]
16     add    $t2, $t3, $t4     # $t2 = fibo[n] + fibo[n+1]
17     sw     $t2, 8($t0)       # armazena o fibo[n+2] = fibo[n] + fibo[n+1] no vetor
18     addi   $t0, $t0, 4       # incrementa em 4 (bytes) o endereço de fibo
19                                # (para alocarmos espaço para o próximo elemento)
20     addi   $t1, $t1, -1      # decrementa o contador do loop
21     bgtz   $t1, loop         # repete, enquanto o contador for maior que 0
22
23     la     $a0, fibo         # adiciona o primeiro argumento para a
24                                # label print (endereço do vetor)
25     add    $a1, $zero, $t5   # adiciona o segundo argumento para a
26                                # label print(variável "tam")
27     jal    print             # chamando a label "print"
28     li     $v0, 10           # chamando o sistema (syscall)
29                                # para terminar a execução
30     syscall
31
```



```

32  # label para printar os elementos
33
34  .data
35      espaco: .asciiz  " "  # espaço para inseri-lo entre os números
36      frase: .asciiz  "Os 20 primeiros elementos de Fibonacci são:\n"
37  .text
38
39  print:
40      add  $t0, $zero, $a0  # $t0 recebe o endereço do vetor
41      add  $t1, $zero, $a1  # $t1 recebe o contador para o loop (out)
42      la   $a0, frase      # carregamento de endereço a fim de imprimir a "frase"
43      li   $v0, 4          # especifica o serviço de printar uma String (nº 4)
44      syscall              # printando a frase
45  out:
46      lw   $a0, 0($t0)     # carrega o número de fibo
47      li   $v0, 1          # especifica o serviço de printar um inteiro (nº 1)
48      syscall              # printa o número especificado
49
50      la   $a0, espaco     # carregamento do endereço de espaço
51      li   $v0, 4          # especifica o serviço de printar uma String (nº 4)
52      syscall              # printa a determinada String
53
54      addi $t0, $t0, 4      # incrementa o endereço para o próximo print
55      addi $t1, $t1, -1     # decrementa o contador do loop
56      bgtz $t1, out        # repete, enquanto o contador for maior que 0
57      jr   $ra             # retorna

```

3. Observações pessoais

Apesar das desvantagens nítidas que as linguagens de baixo nível apresentam, que é a complexidade de compreensão e o aumento de linha de códigos, os benefícios suprem, e suprem muito bem, esses problemas. Durante a execução desse trabalho e “brincando” com o MARS, pude perceber que a superioridade do assembly, em relação a uma linguagem de mais alto nível, é evidente em alguns pontos, como: maior rapidez em processar dados e compilar, maior controle de armazenamento e manipulação de dados, uma vez que podemos indicar em qual registrador iremos armazenar cada dado, possuem instruções úteis, dependendo do objetivo do código, que as linguagens de alto nível não tem.

Foi bem interessante entender, estudar e ter esse primeiro contato, não só com assembly, mas com uma linguagem desse nível. Apesar de ser mais difícil ler e entender o código, tendo que recorrer muitas vezes ao “help” para identificar uma instrução ou serviço de “syscall”, suas grandes vantagens fizeram puxar a minha atenção e explorar mais essa linguagem.