UNIVERSIDADE DE SÃO PAULO
ESCOLA DE ARTES, CIÊNCIAS E HUMANIDADES

AMAURI DE MELO JUNIOR
BRENNO CREMOINI

Organização de Computadores Digitais
Relatório do Exercício-Programa

Profa. Dra. Gisele da Silva Craveiro

São Paulo
2017

Sumário

1 Introdução

O problema sorteado para a realização deste exercício-programa foi a obtenção de máximos divisores comuns (MDC) seguindo o algoritmo de Euclides. O problema divide-se em duas partes:

- Escrever uma função que receba dois números a e b como parâmetros e retorne o MDC deles;
- Escrever um programa que leia da entrada padrão um inteiro positivo n e uma sequência de n inteiros não-negativos e imprime o MDC de todos os números dados.

O algoritmo de Euclides para determinação de MDC é um dos algoritmos mais antigos que se tem conhecimento (data de cerca de 300 anos A.C) e pode ser encontrado no Livro VII da obra Os Elementos, de Euclides. Ainda nos dias de hoje, o Algoritmo de Euclides é uma das maneiras mais simples e eficientes de se calcular MDC. O processo é também conhecido como processo das divisões sucessivas, pois é a partir de sucessivas divisões que ele é executado, e baseia-se no seguinte resultado:

> Sejam a e b números naturais que, para evitar aborrecimentos desnecessários, suporemos $a > b > 0$. Se q e r são, respectivamente, o quociente e resto da divisão de a por b, então $mdc(a, b) = mdc(b, r)$.

Baseado na descrição acima, foi criado o programa em C que atenda as condições exigidas para a solução do problema. Em seguida, baseando-se no código em C, foi criado o programa em *Assembly* correspondente seguindo o Conjunto de Instruções MIPS. O Conjunto de Instruções MIPS será descrito a seguir.

Para desenvolvimento, testes e execução, usamos o simulador MARS, um versátil e leve ambiente de desenvolvimento interativo (IDE, no original) para a elaboração de programas seguindo a arquitetura MIPS. O *software* foi desenvolvido pela Universidade Missouri State e disponibilizado gratuitamente para uso estudantil.

2 O Conjunto de Instruções MIPS

O Conjunto de Instruções MIPS prevê três formatos de instruções, e as instruções podem receber até três operandos. Na arquitetura MIPS, os operandos das instruções são registradores de 32 bits de memória ou palavras endereçadas na memória principal ou pilha de dados. No total, 32 registradores compõem a arquitetura, porém há usos convencionados para cada conjunto que veremos adiante.

2.1 Formatos de Instruções

Os três formatos de instruções previstos são estes que se seguem:

- Tipo R:

  São instruções para as quais todos os operandos se tratam de registradores. Todas as instruções do tipo R possuem o seguinte formato:

  Tabela 1 – Instruções tipo R

OP	rd	rs	rt

  Onde OP corresponde ao código mnemônico da instrução. rs e rt são registradores de origem, e rd é o registrador de destino. Podemos tomar como exemplo a operação *add*:

  ```
  add $s1, $s2, $s3
  ```

- Tipo I:
  São instruções que operam sobre um valor "imediato" e um operando. "Valores imediatos" podem ter no máximo 16 bits de comprimento. Números grandes não devem ser manipulados por instruções imediatas. Todas as instruções do tipo I possuem o seguinte formato:

  Onde OP corresponde ao código mnemônico da instrução. rs é o registrador de origem e rt o de destino. IMM corresponde ao valor imediato. Podemos tomar como

Tabela 2 – Instruções tipo I

OP	rt	rs	IMM

exemplo a operação *addi*:

```
addi $s1, $s2, 100
```

- Tipo J:
  São instruções usadas para realizar saltos. Instruções deste tipo possuem o maior espaço para um valor imediato já que endereços são sempre números grandes. Todas as instruções do tipo J possuem o seguinte formato:

Tabela 3 – Instruções tipo J

OP	LABEL

Onde OP corresponde ao código mnemônico da instrução e $LABEL$ é o endereço alvo para onde deve ocorrer o salto. Podemos tomar como exemplo a operação j:

```
j FIM
```

2.2 Registradores disponíveis

Os 32 registradores são implementados como de uso geral, ou seja, podem ser usados à vontade pelo programador. Os registradores são precedidos pelo caractere $ nas instruções em *assembly*. Podem ser endereçados pelo seu endereço numérico (de $0 a $31) ou pelos seus nomes ($\$t1$, $\$sp$, $\$ra$). Dois registradores especiais são reservados e não podem ser diretamente acessados: Lo e Hi, usados para operações com pontos flutuantes e de multiplicação e divisão.

Convencionalmente, os registradores são organizados por uso e por comportamento esperado conforme o quadro abaixo:

Tabela 4 – Registradores

Número do Registrador	**Nome Alternativo**	**Descrição**
0	$\$zero$	O valor 0
1	$\$at$	(temporário assembler) – reservado pelo assembler
2-3	$\$v0 - \$v1$	(valores) – Resultados de expressões e retornos de funções
4-7	$\$a0 - \$a3$	(argumentos) – Primeiros quatro parâmetros de uma subrotina. Não são preservados entre chamadas de funções.
8-15	$\$t0 - \$t7$	(temporários) – Salvos pelo escopo invocador, se necessário. Subrotinas podem usar sem salvar previamente. Não são preservados entre chamadas de funções.
16-23	$\$s0 - \$s7$	(valores salvos) – Salvos pelo escopo invocado. Uma subrotina que use um destes registradores deve salvar o valor original e restaurá-lo antes de retornar. São preservados entre chamadas de funções.
24-25	$\$t8 - \$t9$	(temporários) – Salvos pelo escopo invocador, se necessário. Subrotinas podem usar sem salvar previamente. Não são preservados entre chamadas de funções.
26-27	$\$k0 - \$k1$	Reservados para uso dos tratamentos de interrupções/exceções.
28	$\$gp$	Ponteiro global. Aponta para o meio do bloco (64kb) de memória no segmento de memória estática.
29	$\$sp$	Ponteiro de pilha. Aponta para a última posição em uso da pilha.

Número do Registrador	Nome Alternativo	Descrição
30	$\$s8/\$fp$	Valor salvo/Ponteiro da moldura. É preservado entre chamadas de funções.
31	$\$ra$	Endereço de retorno.

2.3 Chamadas ao Sistema Operacional

O conjunto de instruções MIPS também prevê um conjunto de chamadas ao sistema operacional que podem ser invocadas através do comando *syscall*. Ao invocar o comando *syscall*, deve-se preencher alguns registradores de modo a indicar qual ação deve ser executada e quais são os argumentos para a ação. O registrador $\$v0$ indicará qual ação deve ser executada e possíveis argumentos são armazenados nos registradores $\$a0 - \$a3$. Quando o sistema operacional executa a ação invocada pelo *syscall*, dependendo da ação, se gerar retorno, este pode ser armazenado nos registradores $\$v0$ e $\$v1$, dedicados a isso.

As chamadas suportadas pelo simulador MARS, bem como quais argumentos são esperados e quais retornos são devolvidos estão listadas abaixo:

Tabela 5 – Chamadas ao Sistema Operacional suportadas pelo MARS

Função	Código em $v0	Argumentos	Resultado
Imprimir inteiro (integer)	1	$a0 = inteiro a ser impresso	
Imprimir ponto flutuante (float)	2	$f12 = ponto flutuante a ser impresso	
Imprimir ponto flutuante de precisão dupla (double)	3	$f12 = ponto flutuante a ser impresso	
Imprimir cadeia de caracteres (string)	4	$a0 = endereço da cadeia de caracteres terminada em null a ser impressa	
Ler inteiro da entrada padrão	5		$v0 contém o inteiro lido

Função	Código em $v0	Argumentos	Resultado
Ler ponto flutuante da entrada padrão	6		$f0 contém o ponto flutuante lido
Ler ponto flutuante de dupla precisão da entrada padrão	7		$f0 contém o ponto flutuante lido
Ler cadeia de caracteres da entrada padrão	8	$a0 = endereço do buffer de entrada $a1 = quantidade máxima de caracteres a serem lidos	Ver notas abaixo
sbrk (alocar memória da pilha)	9	$a0 = quantidade de bytes a serem alocados	$v0 contém o endereço da memória alocada
Sair (terminar execução)	10		
Imprimir caractere (char)	11	$a0 = caractere a ser impresso	Veja notas abaixo
Ler caractere da entrada padrão	12		$v0 contém o caractere lido
Abrir arquivo	13	$a0 = endereço da cadeia de caracteres terminadas em null contendo o caminho do arquivo $a1 = flags $a2 = modo de abertura	$v0 contém o descritor do arquivo (ponteiro –negativo se falhar). Veja notas abaixo
Ler do arquivo	14	$a0 = descritor do arquivo $a1 = endereço do buffer de entrada $a2 = quantidade máxima de caracteres a serem lidos	$v0 contém a quantidade de caracteres lidos (0 se final de arquivo; negativo se falhar)
Escrever em arquivo	15	$a0 = descritor do arquivo $a1 = endereço do buffer de entrada$a2 = quantidade máxima de caracteres a serem lidos	$v0 contém a quantidade de caracteres lidos (negativo se falhar)

Função	Código em $v0	Argumentos	Resultado
Fechar arquivo	16	$a0 = descritor do arquivo	
Sair (terminar com valor)	17	$a0 = resultado do término	Veja notas abaixo

Notas:

Função 8 – Segue a semântica da função $fgets$ do UNIX. Para um determinado tamanho n, a cadeia de caracteres não pode ser superior a $n - 1$. Se menos do que isso, adiciona uma quebra de linha ao fim. Em qualquer caso, preenche com o *bytenull*. Se $n = 1$, a entrada é ignorada e apenas o *bytenull* é armazenado no $buffer$. Se $n < 1$, a entrada é ignorada e nada é escrito no $buffer$.

Função 11 – Imprime o caractere ASCII correspondente ao *byte* de menor ordem.

Função 13 – O MARS implementa três valores de flags: 0 para 'apenas leitura', 1 para 'apenas escrita com criação' e 9 para 'apenas escrita com criação e junção'. Essa função ignora o valor do modo. O descritor de arquivo retornado será negativo se a operação falhar. A implementação de entrada e saída de arquivos por baixo usa os métodos $java.io.FileInputStream.read()$ para ler e $java.io.FileOutputStream.write()$ para escrever. O MARS mantém descritores de arquivos internamente e os aloca começando com 3. Descritores 0, 1 e 2 estão sempre abertos para: ler da entrada padrão, escrever para a saída padrão, e escrever para a saída de erros padrão.

Função 17 – Se o programa MIPS é executado sob controle da interface gráfica MARS (GUI), o código de saída em $\$a0$ é ignorado.

3 Implementação

A parte A do problema foi resolvida implementando a função recursiva abaixo em C, que reproduz o algoritmo de Euclides:

```c
int mdc(int x, int y) {
    if (y == 0) {
        return x;
    }

    if (x == y) {
        return x;
    }

    if (x < y) {
        return mdc(y,x);
    }

    int r = x % y;
    if (r == 0) {
        return y;
    }

    return mdc(y, r);
}
```

E seu código correspondente comentado em *Assembly* MIPS:

```
mdc:
# primeiro de tudo, salvamos o valor de ra na pilha (para saber de onde
↪  viemos)
    addiu    $sp, $sp, -4
    sw       $ra, 0($sp)
# agora podemos começar
#   o segundo parametro é zero? se sim, pula para o trecho que retorna o
↪  primeiro parametro
    beqz     $a1, mdc_retorna_a0
```

```
8  #   os parametros são iguais? se sim, pula para o trecho que retorna o primeiro
   ↪  parametro
9      beq      $a0, $a1, mdc_retorna_a0
10 #   o primeiro parametro é menor do que o segundo? se sim, pula para o trecho
   ↪  que os inverte e chama mdc recursivamente
11     blt      $a0, $a1, mdc_retorna_mdc_invertido
12 #   divide o primeiro parametro pelo segundo
13     div      $a0, $a1
14 #   copia o resto da divisão para a variável temporária t0
15     mfhi     $t0
16 #   o valor de t0 é zero? se sim, significa que são múltiplos entre si,
   ↪  portanto pula para o trecho que retorna o segundo parametro
17     beqz     $t0, mdc_retorna_a1
18 #   se não pulamos até aqui, o que resta é chamar mdc recursivamente, para
   ↪  isso, colocamos os novos parametros nas posicoes a0 e a1 e pulamos para o
   ↪  começo de mdc
19     la       $a0, ($a1)
20     la       $a1, ($t0)
21     jal      mdc
22 # quando voltarmos, precisamos desempilhar o endereço e voltar, por isso,
   ↪  copiamos o que está no topo da pilha nesse instante para ra, movemos o
   ↪  ponteiro da pilha e pulamos para o ponto de ra
23     lw       $ra, 0($sp)
24     addiu    $sp, $sp, 4
25     jr       $ra
26
27 mdc_retorna_mdc_invertido:
28 # copia o valor do primeiro parametro para a variável temporária t0
29     la       $t0, ($a0)
30 # copia o valor do segundo parametro para a variável de parameto a0
31     la       $a0, ($a1)
32 # copia o valor da variável temporária t0 para a variável de parametro a1
33     la       $a1, ($t0)
34 # pulamos para o começo de mdc
35     jal      mdc
36 # quando voltarmos, precisamos desempilhar o endereço e voltar, por isso,
   ↪  copiamos o que está no topo da pilha nesse instante para ra, movemos o
   ↪  ponteiro da pilha e pulamos para o ponto de ra
37     lw       $ra, 0($sp)
38     addiu    $sp, $sp, 4
39     jr       $ra
40 mdc_retorna_a0:
```

```
41  # armazenamos em v0 o valor do primeiro parametro, armazenado até então na
    ↪  variável de parametro a0
42      la      $v0, ($a0)
43  # desempilhamos o endereço para voltar e pulamos
44      lw      $ra, 0($sp)
45      addiu   $sp, $sp, 4
46      jr      $ra
47  mdc_retorna_a1:
48  # armazenamos em v0 o valor do primeiro parametro, armazenado até então na
    ↪  variável de parametro a0
49      la      $v0, ($a1)
50  # desempilhamos o endereço para voltar e pulamos
51      lw      $ra, 0($sp)
52      addiu   $sp, $sp, 4
53      jr      $ra
```

A parte B do problema foi resolvida implementando a função main, ponto de entrada do programa, definida abaixo em C, que invoca a função MDC criada na parte A para obter o MDC dos números informados pelo usuário através da entrada padrão:

```
1  int main(const int argc, const char argv[])
2  {
3      int n, i, current_mdc = 0;
4      int * numbers;
5
6      printf("Entre com a quantidade de números:\n");
7      scanf("%u", &n);
8
9      numbers = (int *) malloc(n * sizeof(int));
10
11     printf("Digite os números, um de cada vez:\n");
12     for (i = 0; i < n; i++) {
13         scanf("%u", &numbers[i]);
14         current_mdc = mdc(numbers[i], current_mdc);
15     }
16
17     printf("O MDC é %d\n", current_mdc);
18 }
```

E seu código correspondente comentado em *Assembly* MIPS:

```
1  .data
2  quantity_message: .asciiz "Entre com a quantidade de numeros:\n"
3  number_message:   .asciiz "Digite os numeros, um de cada vez:\n"
4  result_message:      .asciiz "O resultado do MDC dos numeros informados é: "
5
6  .text
7
8  main:
9  # carrega $v0 com o valor 4 para indicar ao syscall que a chamada ao SO
   ↪  desejada é imprimir string
10     li     $v0, 4
11 # carrega $a0 com o endereço do buffer da mensagem a ser exibida
12     la     $a0, quantity_message
13 # invoca syscall: imprimir a string contida no label 'quantity_message'
14     syscall
15 # carrega $v0 com o valor 5 para indicar ao syscall que a chamada ao SO
   ↪  desejada é ler inteiro da entrada padrão
16     li     $v0, 5
17 # invoca syscall: lê um inteiro da entrada padrão
18     syscall
19 # copia o inteiro informado pelo usuário para o registrador $s0
20     la     $s0, ($v0)
21 # carrega $s1 com o valor 0 -- $s1 será o registrador onde manteremos o MDC
   ↪  conforme for calculado a cada iteração
22     li     $s1, 0
23 # carrega $v0 com o valor 4 para indicar ao syscall que a chamada ao SO
   ↪  desejada é imprimir string
24     li     $v0, 4
25 # carrega $a0 com o endereço do buffer da mensagem a ser exibida
26     la     $a0, number_message
27 # invoca syscall: imprimir a string contida no label 'number_message'
28     syscall
29 # loop: equivalente a um for
30 main_loop:
31 #   primeiro verificamos se o conteúdo de $s0 é zero, se for, pulamos para o
   ↪  fim do programa
32     beqz     $s0, main_end
33 #   carrega $v0 com o valor 5 para indicar ao syscall que a chamada ao SO
   ↪  desejada é ler inteiro da entrada padrão
```

```
34      li    $v0, 5
35  #   invoca syscall: lê um inteiro da entrada padrão
36      syscall
37  #   copia o conteúdo de $s1 (nosso MDC atual) para o registrador $a0 (arguemtno
    ↪   da subrotina mdc)
38      la    $a0, ($s1)
39  #   copia o inteiro informado pelo usuário para o registrador $a1 (argumento da
    ↪   subrotina mdc)
40      la    $a1, ($v0)
41  #   armazena nossa posição atual em $ra e salta para mdc para a execução da
    ↪   subrotina
42      jal    mdc
43  #   copia o conteúdo de $v0 (resultado da subrotina mdc) para $s1 substituindo
    ↪   nosso MDC atual pelo novo calculado
44      la    $s1, ($v0)
45  #   decrementa $s0 para representar o passo do loop
46      addi    $s0, $s0, -1
47  #   salta para o início do loop para nova iteração
48      j    main_loop
49  # fim: onde chegamos depois de ler e processar todos os números informados pelo
    ↪   usuário
50  main_end:
51  # carrega $v0 com o valor 4 para indicar ao syscall que a chamada ao SO
    ↪   desejada é imprimir string
52      li    $v0, 4
53  # carrega $a0 com o endereço do buffer da mensagem a ser exibida
54      la    $a0, result_message
55  # invoca syscall: imprimir a string contida no label 'result_message'
56      syscall
57  # carrega $v0 com o valor 1 para indicar ao syscall que a chamada ao  SO
    ↪   desejada é imprimir inteiro
58      li    $v0, 1
59  # carrega $a0 com o valor de $s1, ou seja, o resultado calculado do MDC
60      la    $a0, ($s1)
61  # invoca syscall: imprimir o número que representa o MDC calculado
62      syscall
63  # carrega $v0 com o valor 10 para indicar ao syscall que a chamada ao SO
    ↪   desejada é fim de execução
64      li    $v0, 10
65  # invoca syscall: fim
66      syscall
```