UNIVERSIDADE DE BRASÍLIA - UnB

Fundamentos de Arquitetura de Computadores
Relatório Projeto 02

Professor Tiago Alves

Joberth Rogeres Tavares Costa - 160128013
Rafael Makaha Gomes Ferreira - 160142369

Introdução

O seguinte projeto tem por finalidade proporcionar um aprimoramento da experiência com a linguagem *Assembly* no ambiente MARS MIPS para os alunos da disciplina de Fundamentos de Arquitetura de Computadores ministrada pelo professor Tiago Alves na faculdade de Engenharias da Universidade de Brasília.

É por meio deste documento que será apresentado o desenvolvimento e o resultado do projeto apresentado aos alunos.

Objetivo

1. Exercitar conceitos de linguagem de montagem(MIPS), especialmente aqueles referentes à implementação de solução de problemas em aritmética inteira.
2. Interagir com ferramentas de desenvolvimento para criação, gerenciamento, depuração e testes de projeto e aplicações.

Materiais

1. Computador com Sistema Operacional programável
2. Ambiente de simulação para arquitetura MIPS: MARS

Questões

1. Escreva um programa em linguagem de montagem para MIPS usando, preferencialmente, o simulador MARS como plataforma de desenvolvimento e validação. A sua aplicação deverá calcular a **exponenciação modular**. Seguem os passos de implementação.

Sua aplicação deverá receber em entrada em console três números inteiros e imprimir, como resultado da operação, uma mensagem.

- O primeiro número será a base, ou seja, o número inteiro cuja potência (modular) será demandada.
- O segundo número será um inteiro não negativo que representará o expoente usado no cálculo da exponencial modular.
- O terceiro número inteiro será o **provável número primo** que definirá a classe de resíduos, ou seja, o **módulo**.
- A mensagem de saída poderá ser:
    - A exponencial modular AA elevado a BB (mod PP) eh ZZ.
    - O modulo nao eh primo.
- Na sua implementação, esperam-se encontrar as funções:
    - le_inteiro, que lerá um primo no console de entrada;
    - eh_primo, que testará se o inteiro indicado é, de fato, um número primo;
    - calc_exp, que calculará a exponencial modular;
    - imprime_erro, função que imprimirá o erro;
    - imprime_saida, função que imprimirá o resultado bem sucedido.
- Outras funções poderão ser criadas, ficando a critério da equipe de implementação.

Soluções

Questão 1-

trab2.asm

```
1         .data
2         .word
3  valor1:          .asciiz "Insira um valor para a base: "
4  valor2:          .asciiz "Insira um valor para o expoente: "
5  valor3:          .asciiz "Insira um valora supostamente primo: "
6  naoPrimo:        .asciiz "O valor inserido não é primo."
7  sucesso1:        .asciiz "A exponencial modular "
8  sucesso2:        .asciiz " elevado a "
9  sucesso3:        .asciiz " (mod "
10 sucesso4:        .asciiz ") eh "
11
```

A princípio, o programa cria *Words* para armazenar as mensagens que serão impressas no console.

trab2.asm

```
12        .text
13
14 le_inteiro:
15        la $a0, valor1          # Imprime mensagem de inserção
16        li $v0, 4
17        syscall
18
19        li $v0, 5               #Leitura do primeiro inteiro
20        syscall
21        move $t0, $v0           # Salva o primeiro inteiro em $t0
22
23        la $a0, valor2          # Imprime mensagem de inserção
24        li $v0, 4
25        syscall
26
27        li $v0, 5
28        syscall                 # Leitura do segundo inteiro
29        move $t1, $v0           # Salva o segundo inteiro em $  t1
30
31        la $a0, valor3          # Imprime mensagem de inserção
32        li $v0, 4
33        syscall
34
35        li $v0, 5
36        syscall                 # Leitura do terceiro inteiro
37        move $t2, $v0           # Passa o inteiro lido para o registrador $t2
38
```

Em seguida, é feita a requisição das entradas. Isso se faz com a impressão no console das mensagens pedindo ao usuário que insira o valor indicado. O terceiro valor inserido será testado para saber se é um número primo.

```
preparação:
        li $t3, 1               # $t3 == (cont)
        move $t6, $t2           # $t6 recebe as subtrações (total)
        li $t7, 1               # $t7 será o valor das subtrações (aux)
        jal sqrt
```

Há uma preparação de alguns registradores para a retirada da parte inteira da raiz quadrada do valor supostamente primo.

```
sqrt:                                   #encontrará a parte inteira da raiz quadrada
        slti $t4, $t6, 1                # ( aux < 1 )
        bne $t4, $zero, fim_sqrt                # if not( aux < 1 ), fim
        add $t7, $t7, 2                 # aux += 2
        add $t3, $t3, 1                 # cont++
        sub $t6, $t6, $t7               # total -= aux
        j sqrt
fim_sqrt:
        jr $ra
```

Para a captura da raíz quadrada do valor supostamente primo, é aplicado o método de Pell. Este algorítmo encontra a parte inteira da raíz quadrada de um valor por meio de sucessivas subtrações de valores ímpares até que o valor auxiliar seja igual ou menor do que zero. A raíz é encontrada pela contagem de iterações feitas ao longo deste processo.

```
        sub $t3, $t3, 1         # remove-se a contagem extra
        move $t7, $t3           # $t7 = sqrt($t2)
        move $t3, $zero         # i = 0
        move $t6, $t2           # copia o conteúdo da terceira entrada para $t6
        add $t6, $t6, 1         # $t6 = $t2 + 1
        j checa_primo
```

Após se obter o valor da raíz pelo algoritmo de Pell remove-se a contagem extra para se ter a raíz corretamente e faz-se alguns preparativos em registradores para se testar se o terceiro valor inserido pelo usuário é realmente um número primo.

```
checa_primo:
        beq $t3, $t7, fim_checagem      # while $t3 < $t7
        div $t2, $t3                    # $t2 / $t3
        mfhi $t4                        # $t4 = %t2 % $t3
        beq $t4, $zero, incrementa      # if ( $t2 % $t3 == 0 ), faz o incremento da quantidade de divisores
        add $t3, $t3, 1                 # i++
        j checa_primo

incrementa:
        add $t3, $t3, 1         # i++
        add $t5, $t5, 1         # divisores++
        j checa_primo

fim_checagem:                           # Fim da verificação de número primo
        li $t3, 0x02
        bne $t5, $t3, imprime_erro              # verifica se possui apenas dois divisores
        j continua

imprime_erro:
        la $a0, naoPrimo                # imprime mensagem de que não é primo
        li $v0, 4
        syscall

        li $v0, 10                      # finaliza o programa
        syscall
```

A checagem de valor primo é feita da maneira simples: serão verificados todos os valores de 1 a raíz quadrada do valor inserido para saber se cada um destes valores é um divisor dele. Para saber se é um divisor, é verificado o resto da divisão do valor inserido pelo valor do contador. Em seguida, é comparado a quantidade de divisores, caso seja igual a 2, o valor inserido é um primo. Caso o valor não seja primo, é impressa uma mensagem de erro e o programa é encerrado.

```
89  continua:
90          li $t3, 0x01            # Máscara para varredura dos bits
91          j msb
92
93  msb:                            # encontrando o bit mais significativo
94          beq $t3, $zero bit_maximo
95          sgt $t4, $t3, $t1       # if ( bit_atual > expoente )
96          beq $t4, 1, continua_2
97          sll $t3, $t3, 1         # Avança um bit com a máscara
98          j msb
99
100 bit_maximo:
101         li $t3, 10000000000000000000000000000000            # setagem da máscara caso ela tenha sido zerada
102         move $t4, $t0                                       # Iniciando $t4 para armazenar as multiplicações
103         srl $t3, $t3, 1                                     # shift devido a primeira iteração da exponenciaçãol
104         j exponenciacao
105
106 continua_2:
107         srl $t3, $t3, 1         # Posiciona corretamente a máscara com o bit mais significativo
108         move $t4, $t0           # Iniciando $t4 para armazenar as multiplicações
109         srl $t3, $t3, 1         # shift devido a primeira iteração da exponenciação
110         j exponenciacao
```

Após a verificação de número primo, encontra-se o bit mais significativo do expoente inserido como uma preparação para o processo de exponenciação modular - parte principal do algoritmo.

O bit mais significativo é encontrado com uma varredura dos bits do expoente inserido por meio de uma máscara que será shiftada para percorrer os bits e de utilização da função lógica *And*.

Caso o bit mais significativo do expoente inserido esteja no bit 32, a maścara tornar-se-á zero. Este caso é verificado e solucionado com a setagem da linha 101.

```
112 exponenciacao:
113         beq $t3, $zero, imprime_saida           # verifica se ainda há bits a serem utilizados
114         and $t5, $t3, $t1               # fazer o and para verificar o bit atual
115         bne $t5, $zero, sqr_mult        # If ( $t5 != 0 ) ; square and multiply
116         mult $t4, $t4                   # Else: square
117         mflo $t4
118         div $t4, $t2                    # divisão do resto atual pelo número primo
119         mfhi $t4                        # captura o resto da divisão em $t4
120         srl $t3, $t3, 1
121         j exponenciacao
122
123 sqr_mult:
124         mult $t4, $t4                   # $t4 ** 2
125         mflo $t4
126         mult $t4, $t0                   # $t4 * base
127         mflo $t4
128         div $t4, $t2                    # divisão do resto atual pelo número primo
129         mfhi $t4                        # captura o resto da divisão em $t4
130         srl $t3, $t3, 1                 # shift para o próximo bit do expoente
131         j exponenciacao
```

A exponencição modular fora feita, inicialmente, de uma maneira mais simples. Porém, tal implementação apresentava problemas para calcular valores de expoente extremos. Para contornar tal problema, fora pesquisado e implementado o algoritmo *Square multiply*. Este, se utiliza de simples multiplicações sucessivas para calcular o valor final da exponenciação desejada.

Para a solução do problema deste projeto, o *Square multiply* foi unido as operações de módulo, ou seja, a cada iteração é retirado o módulo do resultado atual para que se encontre o módulo do resultado final. Assim, as operações são feitas em cima do módulo do resultado anterior e não apenas em cima do resultado das multiplicações.

```
imprime_saida:
        la $a0, sucesso1            # mensagem de sucesso
        li $v0, 4
        syscall

        move $a0, $t0               # imprimindo inteiro
        li $v0, 1
        syscall

        la $a0, sucesso2            # mensagem de sucesso
        li $v0, 4
        syscall

        move $a0, $t1               # imprimindo inteiro
        li $v0, 1
        syscall

        la $a0, sucesso3            # mensagem de sucesso
        li $v0, 4
        syscall

        move $a0, $t2               # imprimindo inteiro
        li $v0, 1
        syscall

        la $a0, sucesso4            # mensagem de sucesso
        li $v0, 4
        syscall

        move $a0, $t4               # imprimindo inteiro
        li $v0, 1
        syscall

        li $v0, 10
        syscall
```

Por fim, são feitas as impressões no console com informações a respeito dos valores inseridos pelo usuário e o resultado obtido ao final das operações.

Conclusão

O projeto proporcionou o aprimoramento da familiarização com o ambiente de desenvolvimento e a linguagem de montagem. Isso proporciona uma facilidade no desenvolvimento e solução de futuros problemas de implementação e de otimização de códigos nessa linguagem.

Os resultados alcançados com a implementação da solução foram satisfatórios. As entradas utilizadas para alguns dos testes levaram em consideração a quantidade de bits que, sem a exponenciação modular, geraram Overflow em seus resultados. Essas entradas apresentaram o resultado matemático esperado no algoritmo desenvolvido.

Em vários websites, há a possibilidade de se verificar exponenciações modulares, porém, ao se inserir valores extremos para o expoente, o algoritmo não consegue computar o resultado final. O algoritmo acima implementado e apresentado se torna um ferramenta para contornar este problema.

Referência

David A. Patterson; John Hennessy, Organização e Projeto de Computadores, Campus, 3a Edição, 2005