UNIVERSIDADE DE SÃO PAULO
ESCOLA DE ARTES, CIÊNCIAS E HUMANIDADES

Projeto da Disciplina
Organização de Computadores Digitais

GABRIEL RODRIGUES DE ALMEIDA RAMOS (1034629)
IVO DE ANDRADE DE DEUS (8075238)

São Paulo, 20 de Junho de 2018

Índice

Organização e Arquitetura MIPS

A arquitetura *MIPS* (*Microprocessor Without Interlocked Pipeline Stages*, ou Microprocessador Sem Estágios Intertravados de Pipeline em inglês) é uma arquitetura de conjunto de instruções do tipo *RISC* (*Reduced Instruction Set Computer*), ao qual permite um número menor de ciclos por instrução em comparação à uma arquitetura de conjunto de instruções do tipo *CISC* (*Complex Instruction Set Computer*), visando assim simplificar e generalizar as instruções computadas por uma máquina.

Uma característica desta abordagem de conjunto de instruções é o tratamento da arquitetura de carregar e armazenar dados onde, enquanto sistemas complexos tratam destes passos em conjunto a maioria das outras instruções, tais tarefas são reservadas para instruções específicas. Com isso, se reduz a quantidade de acessos a memória, os limitando a apenas os acessos estritamente requisitados pelo conjunto de instruções, assim melhorando a eficiência quanto ao tempo de processamento de dados.

Este grau de especificidade de instruções elimina o conceito de micro-programação de processadores baseados neste modelo de arquitetura, uma vez que todas as instruções são executadas diretamente pelo hardware, além de oferecer um baixo nível de complexidade as suas instruções.

Descrição do Problema

O problema apontado para o trabalho foi o exercício 11 da lista de problemas para o exercício programa relacionado a programação em MIPS fornecida pela disciplina. O exercício em si descreve duas fórmulas de recorrências *F* e *G*, as quais funcionam da seguinte forma:

$$\begin{cases} F_1 = 2; \\ F_2 = 1; \\ F_i = 2 \times F_{i-1} + G_{i-2}\ ,\ i \geq 3 \end{cases}$$

$$\begin{cases} G_1 = 1; \\ G_2 = 2; \\ G_i = G_{i-1} + 3 \times F_{i-2}\ ,\ i \geq 3 \end{cases}$$

A partir destas fórmulas, a seguinte tabela pode ser montada quanto a representação de seus valores:

i	**1**	**2**	**3**	**4**	**5**	**...**
$\mathbf{F_i}$	2	1	3	8	24	...
$\mathbf{G_i}$	1	2	8	11	20	...

Tabela 1. Valores iniciais das funções *F* e *G*

Dado as fórmulas de recorrência, as estipulações do exercício programa determinam para que se resolva o item (c) da questão, o qual se pede o desenvolvimento de um programa com base na arquitetura MIPS, em linguagem Assembly, que faça a leitura de um inteiro $n > 2$ e, por meio deste valor dado e com base nas fórmulas fornecidas, obter e imprimir o valor das expressões $F_{n-2} + G_{n+200}$ e $F_{n-200} - G_{n-2}$.

Código de Alto Nível da Solução

Para a formulação de uma lógica de código que fornece uma solução válida ao problema, optou-se pela construção de um algoritmo em linguagem C (construído e compilado utilizando o software Codeblocks 17.12) onde, por mais que seja uma linguagem de alto nível em relação à linguagem Assembly, sua proximidade a esta possibilita uma melhor adaptação de lógica de código de uma linguagem para outra.

De forma geral, foi determinado que o problema necessita a seguinte abordagem:

- Declaração de todas as variáveis pertinentes à lógica do código, o que inclui espaços para reservar os valores de *n,* de F_i e de G_i específicos, e de outras variáveis caso necessário;
- Leitura e armazenamento do inteiro *n*;
- Verificação de validade de entrada do valor *n*;
- Cálculo de F_{n-2} e G_{n-2} , com atenção aos casos específicos de *n - 2* ser igual à 1 ou 2;
- Cálculo de F_{n+200} e G_{n+200} , novamente prestando cautela aos casos específicos de *n - 2* ser igual à 1 ou 2;
- Cálculo das expressões $F_{n-2} + G_{n+200}$ e $F_{n-200} - G_{n-2}$, conforme valores calculados e salvos nos passos anteriores;
- Imprimir na saída os valores obtidos para estas expressões;
- Encerrar o programa.

Com isso, construiu-se o seguinte código com solução válida para o problema disposto ao grupo, com seu comentário em detalhe feito em partes conforme lê-se as linhas de código em ordem de execução:

```c
#include <stdio.h>
void main()
{

    int n = 0;
    int i = 0;
    double fi;
    double fimenos1 = 1;
    double fimenos2 = 2;
    double gi;
    double gimenos1 = 2;
    double gimenos2 = 1;
    double fnmenos2;
    double gnmenos2;
```

Figura 1. Código de Alto Nível da Solução - Inicialização

Primeiramente, a livraria de funções *stdio.h* é adicionada, a qual lida com entradas e saídas em C para, em seguida, dar-se início a função *main*, que irá trabalhar com o problema em questão e, para tal, foram determinadas as seguintes variáveis para o código:

- Uma variável do tipo inteiro para o valor de entrada *n*;
- Uma variável do tipo inteiro para um valor de iteração *i*, tendo em mente a sua necessidade no código em sua versão Assembly;
- Variáveis do tipo *double* para os valores de F_i , F_{i-1} e F_{i-2} , inicializadas para $i = 3$ nos dois últimos valores;
- De forma similar acima, variáveis do tipo *double* para os valores de G_i , G_{i-1} e G_{i-2} ;
- Por fim, variáveis do tipo *double* para salvar os valores de F_{n-2} e G_{n-2} .

```c

    printf("n = ");

    scanf("%d",&n);

    if(n <= 2){
        printf("Entrada invalida.");
        return;
    }

```

Figura 2. Código de Alto Nível da Solução - Validação de Entrada

Em seguida, rapidamente trabalha-se com a entrada do valor *n* para o cálculo das expressões pedidas pelo problema, além de verificar a sua validade de entrada (caso se trate de uma entrada inválida, ou seja, $n < 3$, o algoritmo termina sua execução após informar o usuário sobre esta).

```
25      if(n == 3){
26          fnmenos2 = fimenos2;
27          gnmenos2 = gimenos2;
28          i = 1;
29      }
30
31      else if(n == 4){
32          fnmenos2 = fimenos1;
33          gnmenos2 = gimenos1;
34          i = 2;
35      }
36
37      else{
38          i = 3;
39          for(int j = 3; j <= (n-2); j++){
40              fi = fimenos1*2 + gimenos2;
41              gi = gimenos1 + 3*fimenos2;
42              fimenos2 = fimenos1;
43              fimenos1 = fi;
44              gimenos2 = gimenos1;
45              gimenos1 = gi;
46              i++;
47          }
48          fnmenos2 = fimenos1;
49          gnmenos2 = gimenos1;
50      }
```

Figura 3. Código de Alto Nível da Solução - Cálculo de F_{n-2} e G_{n-2}

Caso se trate de uma entrada válida, o algoritmo inicializa sua execução para o cálculo de F_{n-2} e G_{n-2} , onde existem dois casos nos quais estes valores já são fornecidos nas definições de suas recorrências: para $n = 3$ e $n = 4$, uma vez que os valores para $n - 2$ destes são 1 e 2, respectivamente. Para estes casos em específico, carrega-se os valores predefinidos às variáveis *fnmenos2* e *gnmenos2* para o uso de valores em momento posterior.

Para valores $n > 4$, ou seja ($n - 2$) > 2, calculamos os valores de F_{n-2} e G_{n-2} por meio de um laço *for*, que por sua vez itera (por meio da variável *i*) do primeiro valor não definido de *F* e *G*, em 3, até $n - 2$, onde:

- Calcula-se F_i e G_i , na variáveis *fi* e *gi* respectivamente, utilizando os valores de F_{i-1} , F_{i-2} , G_{i-1} e G_{i-2} , previamente salvas em *fimenos1*, *fimenos2*, *gimenos2* e *gimenos2* respectivamente.
- Atualizam-se os valores de *fimenos1*, *fimenos2*, *gimenos2* e *gimenos2*, em caso de loops futuros.

Assim que o loop atinge sua condição de parada, ou seja, tenha calculado F_{n-2} e G_{n-2} , estes valores, presentes em *fimenos1* e *gimenos1*, são salvos em *fnmenos2* e *gnmenos2*.

```
52      if(i == 1){
53          n = 203;
54          i = 3;
55      }
56
57      else if(i == 2){
58          n = 204;
59          i = 3;
60      }
61
62      else n = n + 202;
63
64      for(int k = i ; k <= n; k++){
65          fi = fimenos1*2 + gimenos2;
66          gi = gimenos1 + 3*fimenos2;
67          fimenos2 = fimenos1;
68          fimenos1 = fi;
69          gimenos2 = gimenos1;
70          gimenos1 = gi;
71      }
72
```

Figura 4. Código de Alto Nível da Solução - Cálculo de F_{n+200} e G_{n+200}

De forma similar, o algoritmo calcula os valores de F_{n+200} e G_{n+200} , inicialmente tratando os casos específicos para *n* igual a 3 ou 4 (para corrigir a duração do segundo loop *for* uma vez que F_{n-2} e G_{n-2} foram dados pela definição da recorrência), e enfim percorrendo um laço *for* que calcula a recorrência até os valores de F_{n+200} e G_{n+200} , que ficam salvos em *fimenos2* e *gimenos2*.

```
73      double soma1 = fnmenos2 + gimenos1;
74      double soma2 = fimenos1 - gnmenos2;
75
76      printf("F(n-2) + G(n+200) = %.3e e ", soma1);
77      printf("F(n+200) - G(n-2) = %.3e\n", soma2);
78
79      return;
80
81  }
```

Figura 5. Código de Alto Nível da Solução - Cálculo e Impressão das Expressões $F_{n-2} + G_{n+200}$ e $F_{n+200} - G_{n-2}$

Por fim, com os valores de F_{n-2} , F_{n-200} ,G_{n-2} e G_{n+200} obtidos, basta apenas calcular os resultados das expressões requisitadas pelo problema e exibir estes resultados ao usuário, assim concluindo a execução do código.

Código Assembly Desenvolvido

Com a solução de código de alto nível em mente, agora é possível obter a resolução deste problema em linguagem Assembly, dado que haja os devidos tratamentos de conversão de código. Considerando as similaridades da linguagem C com a programação de baixo nível mencionadas anteriormente, as modificações necessárias à lógica de código são mais técnicas do que logística em si.

```
1  .data
2  fimenos2:   .double 2.0
3  fimenos1:   .double 1.0
4  gimenos2:   .double 1.0
5  gimenos1:   .double 2.0
6  fi:         .double 0.0
7  gi:         .double 0.0
8  fnmenos2:   .double 0.0
9  gnmenos2:   .double 0.0
10 n:          .word 0
11 entreN:     .asciiz "n = "
12 funcao1:    .asciiz "F(n-2) + G(n+200) = "
13 funcao2:    .asciiz " e F(n+200)- G(n-2) = "
14 invalida:   .asciiz "Entrada invalida!"
15
```

Figura 6. Código Assembly Desenvolvido - Inicialização

Em sua inicialização, declaram-se, de forma similar ao código desenvolvido em C:

- Uma variável do tipo *word* para o valor de entrada *n*;
- Variáveis do tipo *double* para F_i , F_{i-1} , F_{i-2} , F_{n-2} , G_i , G_{i-1} , G_{i-2} e G_{n-2} , para armazenar valores de interesse para *F* e *G*;
- Variáveis do tipo *asciiz* para salvar strings de interface de usuário.

Nota-se também a ausência do iterador que, por mais que não haja comandos diretos para laços *for* na linguagem Assembly, é possível utilizar *branches* para gerar laços, assim como usar endereços em memória para guardar uma variável de iteração.

```
16  .text
17  #imprime n =
18  li $v0, 4
19  la $a0, entreN
20  syscall
21
22  #obtem n
23  li $v0, 5
24  syscall
25  sw $v0, n
26
27  #verifica n > 2
28  lw $t0, n
29  bgt $t0, 2, passo1
30  j done
31
```

Figura 7. Código Assembly Desenvolvido - Validação de Entrada

Como visto no código de alto nível, também faz-se uma verificação da validade de entrada, onde o programa segue um *branch* para finalização do programa se $n < 2$, visível no final do código na Figura 12.

```
32  passo1:     #calcula F(n-2) e G(n-2)
33  beq $t0, 3, pega_n1
34  beq $t0, 4, pega_n2
35  j calcula1
36
37  pega_n1:    #caso n - 2 = 1
38  li $t1, 1
39  l.d $f2, fimenos2
40  s.d $f2, fnmenos2
41  l.d $f2, gimenos2
42  s.d $f2, gnmenos2
43  j passo2
44
45  pega_n2:    #caso n - 2 = 2
46  li $t1, 2
47  l.d $f2, fimenos1
48  s.d $f2, fnmenos2
49  l.d $f2, gimenos1
50  s.d $f2, gnmenos2
51  j passo2
52
```

Figura 8. Código Assembly Desenvolvido - Cálculo de F_{n-2} e G_{n-2} para n < 5

Caso contrário, o programa realiza um *jump* de instruções e inicializa a execução do seu algoritmo para, primeiramente, os casos de n igual a 3 ou 4, onde os valores de F e G para n - 2 estão definidos. Para tal, há um *branch* secundário para cada caso específico de $n < 5$ e, caso contrário, um *jump* para o cálculo dos valores de F_{n-2} e G_{n-2} para $n \geq 5$.

```
53  calcula1:   #caso n - 2 > 2
54  li $t1, 3   #for( int i = 3; ...
55  loop1:
56  bgt $t1, $t0, saida1    # i <= (n-2); ...
57
58  #calcula F(i)
59  l.d $f2, fimenos1
60  l.d $f4, gimenos2
61  add.d $f2, $f2, $f2
62  add.d $f2, $f2, $f4
63  s.d $f2, fi
64
65  #calcula G(i)
66  l.d $f2, gimenos1
67  l.d $f4, fimenos2
68  add.d $f2, $f2, $f4
69  add.d $f2, $f2, $f4
70  add.d $f2, $f2, $f4
71  s.d $f2, gi
72
73  #ajusta para proximo passo do for
74  l.d $f2, fimenos1
75  s.d $f2, fimenos2
76  l.d $f2, fi
77  s.d $f2, fimenos1
78  l.d $f2, gimenos1
79  s.d $f2, gimenos2
80  l.d $f2, gi
81  s.d $f2, gimenos1
82
83  addi $t1, $t1, 1    # i++){
84  j loop1
85  saida1:         # }
86  l.d $f2, fimenos1
87  s.d $f2, fnmenos2
88  l.d $f2, gimenos1
89  s.d $f2, gnmenos2
```

Figura 9. Código Assembly Desenvolvido - Cálculo de F_{n-2} e G_{n-2} para n ≥ 5

Como a linguagem Assembly trata-se de uma abordagem de baixo nível à programação, a estrutura de um laço *for* precisa ser adaptada via as instruções disponíveis para existir. Para tal objetivo, é carregado em memória um valor que servirá de iterador para a execução do laço. Em seguida, inicializa-se o conjunto de instruções que, num primeiro momento, passa por uma verificação de iterador onde, caso este não ative a condição de parada, permite a realização dos comandos do laço para o cálculo de valores das funções *F* e *G*, sequência finalizada por um *jump* de volta ao início do laço de instruções. Caso contrário, há um *branch* para as instruções após o laço, que salvam em memória os valores de F_{n-2} e G_{n-2} calculados pelo laço e salvos em memória também.

```
91  passo2:     #calcula F(n+200) e G(n+200)
92  beq $t1, 1, condicao1   #for( int i = 1,
93  beq $t1, 2, condicao2   #for( int i = 2,
94  addi $t0, $t0, 202  #for( int i de onde parou no
    ultimo for
95  j loop2
96
97  condicao1:
98  li $t0, 203
99  li $t1, 3
100 j loop2
101
102 condicao2:
103 li $t0, 204
104 li $t1, 3
105 j loop2
```

Figura 10. Código Assembly Desenvolvido - Correção da Condição de Parada do Laço Recursivo do Cálculo de F_{n+200} e G_{n+200} para n < 5

Com os valores de *F* e *G* para *n* - 2 salvos em memória, o algoritmo prepara os valores de *n* + 200 e do iterador na memória para o início do segundo laço de operações, realizando passos específicos para *n* < 5 por meio de *branches,* ou seguindo a execução do código para atualizar o valor de *n* - 2 para *n* + 200 e reutilizar o valor de *i* em memória, uma vez que basta dar continuação no laço para os casos onde *n* ≥ 5.

```
107 loop2:
108 bgt $t1, $t0, saida2    # i <= (n+200); ...
109
110 #calcula F(i)
111 l.d $f2, fimenos1
112 l.d $f4, gimenos2
113 add.d $f2, $f2, $f2
114 add.d $f2, $f2, $f4
115 s.d $f2, fi
116
117 #calcula G(i)
118 l.d $f2, gimenos1
119 l.d $f4, fimenos2
120 add.d $f2, $f2, $f4
121 add.d $f2, $f2, $f4
122 add.d $f2, $f2, $f4
123 s.d $f2, gi
124
125 #ajusta para proximo passo do for
126 l.d $f2, fimenos1
127 s.d $f2, fimenos2
128 l.d $f2, fi
129 s.d $f2, fimenos1
130 l.d $f2, gimenos1
131 s.d $f2, gimenos2
132 l.d $f2, gi
133 s.d $f2, gimenos1
134
135 addi $t1, $t1, 1    # i++){
136 j loop2
137 saida2:
```

Figura 11. Código Assembly Desenvolvido - Cálculo de F_{n+200} e G_{n+200}

```
#calcula F(n-2) + G(n+200)
l.d $f0, fnmenos2
l.d $f2, gimenos1
add.d $f0, $f0, $f2

#(e imprime)
li $v0, 4
la $a0, funcao1
syscall
li $v0, 3
mov.d $f12, $f0
syscall

# e F(n+200) - G(n-2)
l.d $f0, fimenos1
l.d $f2, gnmenos2
sub.d $f0, $f0, $f2

#(e imprime)
li $v0, 4
la $a0, funcao2
syscall
li $v0, 3
mov.d $f12, $f0
syscall

li $v0, 10
syscall

done:   #pulo pro final do programa
li $v0, 4
la $a0, invalida
syscall

li $v0, 10
svscall
```

Figura 12. Código Assembly Desenvolvido - Cálculo e Impressão das Expressões $F_{n-2} + G_{n+200}$ e $F_{n+200} - G_{n-2}$

Por fim, ao finalizar o segundo laço de recorrência para o cálculo dos valores de *F* e *G* para *n* + 200, basta utilizar os valores salvos em memória para os valores necessários para calcular as expressões e fazer chamadas de sistema para efetuar sua saída para o usuário. Concluído o processo das saídas, o código finaliza sua execução.

Descrição das Instruções Utilizadas

Nosso código utilizou-se de diversas instruções, cada uma representando um ciclo de operações (de busca, indireto, execução e interrupção), onde cada ciclo consiste de micro-operações, as quais realizam o tratamento de endereços e dados a fim de suprir os objetivos do algoritmo.

No ciclo inicial de busca, é obtida a instrução a ser realizada ao carrega-la da memória. Para tal, e nos ciclos a seguir, são utilizados quatro registradores: o registrador de endereço de memória (MAR), o registrador de armazenamento temporário de dados (MBR), o contador do programa (PC), e o registrador de instrução (IR).

```
t1: MAR <- (PC)
t2: MBR <- Memory
    PC <- (PC) + 1
t3: IR <- (MBR)
```

Tabela 2. Ciclo de Busca

Primeiramente, move-se o endereço presente no PC para o MAR, já que este é o único registrador conectado às linhas de endereço do barramento do sistema. Em seguida, adquire-se a instrução ao carrega-la da memória, com seu endereço salvo no MAR colocado no barramento para que a unidade de controle realize um comando de leitura, para o MBR por meio da cópia do resultado da busca efetuada pela unidade de controle disposta no barramento do sistema. Além disso, ao mesmo tempo atualiza-se o PC para que este esteja já pronto para efetuar a próxima instrução, uma vez que estas duas instruções não interferem uma com a outra. Por fim, move-se o conteúdo do MBR para o IR, que libera o espaço do primeiro para qualquer uso necessário deste em um possível ciclo de endereçamento direto à memória.

Terminado o ciclo de busca de instrução, agora busca-se os operandos fontes onde, caso a instrução específica um endereçamento indireto ao invés de direto, será necessário efetuar um ciclo indireto antes do ciclo de execução. Para tal, como visto na tabela abaixo, primeiro move-se o campo do endereço da instrução salvo no IR para o MAR, que obtém o endereço do operando posteriormente adicionado no MBR. Por fim, atualiza-se o campo de endereço em IR com o valor atualizado disposto no MBR, agora com um endereço direto pronto para o ciclo de execução.

```
t1: MAR <- (IR(Address))
t2: MBR <- Memory
t3: IR(Address) <- (MBR(Address))
```

Tabela 3. Ciclo Indireto

Concluídos os ciclos de busca e indiretos necessários para o tratamento da instrução, inicializa-se o ciclo de execução, onde este irá depender de cada instrução requisitada pelo algoritmo.

Tome, por exemplo, a instrução aritmética *addi* apresentada no problema, a qual atualiza o valor do contador sendo utilizado para calcular todos os valores necessários para a recorrência necessária para a obtenção de F_{n+200} e G_{n+200} após o cálculo de F_{n-2} e G_{n-2} para $n \geq 5$. Para tal, o código realiza a instrução *addi $t0, $t0, 200*, ou seja, some o valor localizado em *$t0* a 200 e salve-o no mesmo registrador *$t0*. Neste caso específico, o valor imediato 200 está salvo no registrador *$r4*, então basta realizar a operação e finalizar o processo deste ciclo de instrução.

```
t1: T0 <- (T0) + (R4)
```

Tabela 4. Ciclo de Execução para *addi $t0, $t0, 200*

De forma quase similar, as instruções do tipo *add.d* e *sub.d*, as quais realizam a soma (ou subtração) de dois valores do tipo *double* salvos em dois registradores de 32-bits cada e salvam o resultado em um dos registradores *double* indicados na operação, seguem a mesma lógica em um processador de ponto flutuante.

`t1: FD <- (FS) + (FT)`
`t1: FD <- (FS) - (FT)`

Tabela 5. Ciclo de Execução para *add.d $fd, $fs, $ft* e *sub.d $fd, $fs, $ft*

Esse processo também inclui as instruções *mov.d*, que transfere o valor de um registrador do tipo *double* para outro, e *li $v0, X* , que carrega um valor imediato *X* para um registrador (no caso específico do nosso algoritmo, ou para mandar um código de instrução para o registrador *$v0* ou atualizar nossos valores de *n* e do contador de recursão salvos em *$t0* e *$t1*, respectivamente), onde também estão envolvidas nestas instruções a movimentação de valores já presentes em registradores.

`t1: FD <- (FS)`
`t1: TD <- (R4)`

Tabela 6. Ciclo de Execução para *mov.d $fd, $fs* e *li $td, X*

Em um ciclo de execução mais complexo, comandos como *la*, que carrega um valor de memória em um registrador especificado, existe um maior número de instruções, dado que seja necessário realizar uma busca adicional à memória. Nessa busca, o endereço do conteúdo, localizado no IR, é carregado no MAR, que dispõe este no barramento do sistema para que a unidade de controle efetue a busca e, no segundo passo, carregue essa informação no MBR. Por fim, se repassa o valor de MBR para o registrador de destino especificado na linha de instrução. Esse conjunto de micro-operações é similar ao encontrado nos ciclos de instruções *lw* e *l.d*.

```
t1: MAR <- (IR(Address))
t2: MBR <- Memory
t3: TD <- (MBR)
```

Tabela 7. Ciclo de Execução para *la $td, address*

Para as operações de *store*, ao contrário das instruções *load* mencionadas acima, é salvo em memória o valor de um determinado registrador. Para cumprir tal tarefa, se repassa o conteúdo do registrador desejado a ser salvo para o MBR e o seu endereço para o MAR, o qual coloca este valor no barramento do sistema para que a unidade de controle salve este em memória. Este ciclo, por sua vez, é similar ao utilizado pela instrução *s.d*.

```
t1: MBR <- (TD)
    MAR <- Address
t2: Memory <- (MBR)
```

Tabela 8. Ciclo de Execução para *sw $td, address*

Por fim, instruções de desvio como as *branches* e os *jumps* terão os ciclos de execução mais complexos, uma vez que tratem de verificações de dados que possam definir que ciclo de instrução a ser tomado pelo algoritmo com base destes (no caso dos *branches*), ou na alteração do fluxo de instruções (no caso dos *jumps*).

Obtendo como exemplo a instrução *j label*, que atualiza o valor encontrado no PC para inicializar um processo de instruções indicado pelo *label*, que salva o endereço da próxima instrução a ser executada.

```
t1: MAR <- (IR(Address))
t2: MBR <- Memory
t3: PC <- (MBR)
```

Tabela 9. Ciclo de Execução para *j label*

Outro exemplo seria a pseudo-instrução *bgt $td1, $td2, label*, que verifica se o conteúdo do registrador *$td1* é maior que o do registrador *$td2* e, caso verdadeiro, realiza um desvio de instruções ao atualizar o valor encontrado em PC. Para tal, está primeiro verifica e determina se o valor em *$td2* é menor que o valor em *$td1* (por meio da instrução *slt $td0, $td2, $td1*), e logo em seguida verifica esta comparação e realiza o *branch* se esta for verdadeira (por meio da instrução *bne $td0, 0, label*). Para a instrução *beq $td1, $td2, label*, também utilizada no código, ela verifica a igualdade de dois valores e realiza um branch caso verdadeiro (de forma similar a instrução *bne*).

```
t1: TS0 <- (TD2) - (TD1)
    TS1 <- (Imediate 1)
 if(TS0 < 0) then (TD0 <- TS1)
```

```
t1: MAR <- (IR(Address))
t2: MBR <- Memory
 if(TD0 < 0) then (PC <- (MBR))
```

Tabela 10. Ciclo de Execução para *slt $td0, $td2, $td1* e *bne $td0, 0, label*

Com o fim do ciclo de execução, determina-se se existe a necessidade de um ciclo de interrupção, onde primeiramente é repassado o conteúdo do PC para o MBR e em seguida são salvos os endereços de armazenamento deste conteúdo para a memória no MAR e o endereço da rotina de interrupção a ser seguida no PC. No final, salva-se o conteúdo de MBR na memória e o sistema procede com as novas instruções salvas no PC. Porém, apesar de parte crucial de alguns algoritmos, este ciclo não se encontra presente no código apresentado neste trabalho.

```
t1: MBR <- (PC)
t2: MAR <- Store Address
    PC <- Routine Address
t3: Memory <- (MBR)
```

Tabela 11. Ciclo de Interrupção

Referências Bibliográficas

BACON, Jason W. **Chapter 5. The MIPS Architecture.** In: Computer Science 315 Lecture Notes. 2010-2011. Disponível em <http://www.cs.uwm.edu/classes/cs315/Bacon/Lecture/HTML/ch05.html>. Acesso em: 18 de junho de 2018.

FRENZEL, Jim. **MIPS Instruction Reference.** 10 de setembro de 1998. Disponível em <http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html>. Acesso em: 18 de junho de 2018.

NIKOLOPOULOS, Dimitris. **MIPS Assembly.** 2007. 9 slides. Disponível em <http://courses.cs.vt.edu/~cs2504/spring2007/lectures/lec10.pdf>. Acesso em: 18 de junho de 2018.

REED, Dale. **MIPS Architecture and Assembly Language Overview.** Disponível em <http://logos.cs.uic.edu/366/notes/mips%20quick%20tutorial.htm>. Acesso em: 18 de junho de 2018.

TUTORIALSPOINT. **Assembly Programming Tutorial.** Disponível em <https://www.tutorialspoint.com/assembly_programming/index.htm>. Acesso em: 18 de junho de 2018.

UNIVERSITY OF WASHINGTON. **MIPS Assembly Language Examples.** 2003. Disponível em <https://courses.cs.washington.edu/courses/cse378/03wi/lectures/mips-asm-examples.html>. Acesso em: 18 de junho de 2018.

WALKINGSHAW, Eric. **Introduction to MIPS Assembly Programming.** 23-25 de Janeiro de 2013. 26 slides. Disponível em <http://web.engr.oregonstate.edu/~walkiner/cs271-wi13/slides/04-IntroAssembly.pdf>. Acesso em: 18 de junho de 2018.