Desenvolvimento de Aplicações em Assembly MIPS

6 de outubro de 2020
Organização e Arquitetura de Computadores – Turma B

Aécio Fernandes Galiza Magalhães
Departamento de Ciência da Computação - UnB
15/0115121
150115121@aluno.unb.br

Abstract

Este relatório tem como objetivo apresentar conceitualmente o tema proposto, bem como análise de simulações computacionais e desempenho relacionados à aplicação desenvolvida, para que seja possível a verificação experimental do projeto.

1. Objetivos

O projeto do Laboratório 01 tem como objetivo a aplicação prática dos conceitos estudados na disciplina referentes à arquitetura de computadores e sua organização, especificamente abordando a arquitetura MIPS. Será trabalhada a capacidade de entender a relação entre um algoritmo em Assembly MIPS e a montagem de seu arquivo código objeto, bem como análises de desempenho modulares. O projeto do laboratório 01 consiste em, através do desenvolvimento de uma aplicação na linguagem de programação *Python*, realizar o trabalho do montador, gerando dois arquivos separados no formato *.MIF*, um para a segmentação *.data* e outro para *.text*.

2. Introdução

2.1. Arquitetura

Segundo Patterson [1, p. A-45], a arquitetura de um processador MIPS R2000 consiste em uma Unidade Principal de Processamento, contendo 32 registradores para uso geral registrados de 0-31 [1, p. A-22], e dois Coprocessadores, responsáveis por gerenciar *traps* e memória virtual, bem como operações com tipos de dados como ponto flutuante. A importância do entendimento dessa arquitetura se dá pelo fato de que, com o MIPS, trabalhamos exatamente com esses registradores. Nela, temos um conjunto de instruções que utilizam esses registradores como operandos, bem como valores imediatos, e apenas instruções do tipo *load* e *store* para acessar a memória. Em nosso código Assembly, temos duas principais áreas de informação: a área definida pelo termo *.data*, diretiva responsável por informar ao montador que os presentes a seguir devem ser interpretados como dados [1, p. A-48], e *.text*, responsável por informar que os seguintes são instruções [1, p. A-49].

2.2. Montador

Para entender essa formatação do código Assembly, devemos entender o montador. Este é o responsável por transformar o código com dados e instruções Assembly em código de máquina, chamado arquivo objeto [1, p. A-10], contendo a tradução das instruções e dados de forma que o processador possa entender para executar. Neste projeto, esse código objeto montado será representados por arquivos *.MIF* [2], um arquivo de texto ASCII que poderá ser utilizado como *input* em um simulador a fim de representar-se a interpretação do processador frente a este arquivo.

2.3. Código Objeto

O arquivo de saída referente ao segmento *.text* engloba a tradução das instruções em Assembly para seu referente em código de máquina. Segundo o manual [2], tem-se a premissa de que, dentro deste arquivo, as instruções serão representadas no formato "*Address : Data*", onde "*Address*" representa, sequencialmente, os endereços das instruções do programa em Assembly e "*Data*" representa a instrução em si, traduzida para linguagem de máquina.

Para realizar essa tradução, deve-se seguir o padrão utilizado no Manual do Set de Instruções da arquitetura MIPS [3].

Tipo R	opcode (6)	rs (5)	rt (5)	rd (5)	shamt (5)	func (6)
add	000000	rs	rt	rd	00000	100000
sub	000000	rs	rt	rd	00000	100010
and	000000	rs	rt	rd	00000	100100
or	000000	rs	rt	rd	00000	100101
nor	000000	rs	rt	rd	00000	100111
xor	000000	rs	rt	rd	00000	100110
slt	000000	rs	rt	rd	00000	101010
addu	000000	rs	rt	rd	00000	100001
subu	000000	rs	rt	rd	00000	100011
sll	000000	00000	rt	rd	shift amount	000000
srl	000000	00000	rt	rd	shift amount	000010
mult	000000	rs	rt	00000	00000	011000
div	000000	rs	rt	00000	00000	011010
mfhi	000000	00000	00000	rd	00000	010000
mflo	000000	00000	00000	rd	00000	010010
clo	000000	rs	00000	rd	00001	010001
srav	000000	rs	rt	rd	00000	000111
sra	000000	00000	rt	rd	shift amount	000011
jalr	000000	rs	00000	11111	hint (00000)	001001
jr	000000	rs	00000	00000	hint (00000)	001000
madd[1]	011100	rs	rt	00000	00000	000000
msubu[1]	011100	rs	rt	00000	00000	000101

Table 1. Tabela de instruções do tipo R.

Tipo I	opcode (6)	rs (5)	rt (5)	imm (16)
lw	100011	rs	rt	offset
sw	101011	rs	rt	offset
beq	000100	rs	rt	offset
bne	000101	rs	rt	offset
bgez	000001	rs	00001	offset
bgezal	000001	rs	10001	offset
lui	001111	00000	rt	imm
addi	001000	rs	rt	imm
andi	001100	rs	rt	imm
ori	001101	rs	rt	imm
xori	001110	rs	rt	imm

Table 2. Tabela de instruções do tipo I.

3. Materiais e Métodos

3.1. Ambiente e Ferramentas

Durante o desenvolvimento do projeto, foi utilizado o Sistema Operacional *Ubuntu* em sua versão **16.04**. Neste, foi instalado o *Python* em sua versão **3.5**, para o ambiente de desenvolvimento. O código do projeto foi escrito através do editor de texto *Atom*, na versão **1.51**.

Por questões de verificação da proposta e validação da aplicação criada, foi utilizado, também, o software *MARS*, na versão **4.5**, cuja utilidade estará descrita na subseção "Métodos".

3.2. Metodologia

A partir da lista de instruções solicitadas no roteiro do projeto, criou-se tabelas para facilitar o monitoramento de instruções implementadas e instruções a implementar durante o desenvolvimento, classificadas de acordo com seu tipo.

Além do mapeamento das instruções, também foi necessário, no desenvolvimento da aplicação, representar um dicionário para o mapeamento dos registradores, de acordo com a tabela presente no manual [4].

Tipo J	opcode (6)	instr$_i ndex(16)$
j	000010	instr$_i ndex$
jal	000011	instr$_i ndex$

Table 3. Tabela de instruções do tipo J.

opcode (6)	rs (5)	rt (5)	rd (5)	shamt (5)	func (6)

Table 4. Formatação do código-objeto para instruções do tipo R.

A partir disso, o código é iniciado com uma solicitação de um arquivo de *input* do usuário, que será um arquivo de texto ASCII com um algoritmo implementado em Assembly MIPS, com a extensão ".asm".

Com esse arquivo, o código analisa separadamente suas duas seções, *.text* e *.data*

3.3. Área *.text*

Para a área *.text*, que é composta pelas instruções do código, foi necessário, primeiramente, identificar o tipo de instrução que se tratava, pois o código-objeto gerado na saída depende diretamente disso. O arquivo é lido linha por linha e, de acordo com a instrução declarada no começo da linha, ela é separada e processada em três funções distintas para cada tipo: *transforma_tipoR* para instruções do tipo R, *transforma_tipoI* para instruções do tipo I, com exceção dos *branches*, que exigem um tratamento especial, e *transforma_jump_branch*, que trata as instruções do tipo J e também as instruções *beq*, *bne*, *bgez* e *bgezal*.

3.3.1 Função *transforma_tipoR*

Em se tratando de instruções do tipo R, temos, por definição, que o código-objeto gerado para estas é no formato apresentado na Tabela 4, de acordo com a documentação [3]. Os números em parênteses representam a quantidade de bits para cada parcela. Nesta função, a metodologia utilizada foi a construção de uma variável para retorno denominada *machineCode*, composta pela concatenação das *strings* referentes a cada componente da instrução. Por padrão, durante a execução desta, é atribuído o valor "000000" para o campo *opcode* das instruções, exceto *madd* e *msubu*.

Dentro das instruções do tipo R, possuímos algumas instruções que recebem 3 registradores como operandos, outras que recebem 2 e mais algumas que recebem apenas 1 registrador. Para definir os valores de **rs**, **rt** e **rd**, é necessário verificar se seu valor é referente ao registrador naquela posição ou, para casos com menos de 3 registradores operandos, se é um valor tabelado. Assim, para a contagem de registradores em cada instrução (já

classificada como tipo R), foi necessária a utilização de outra função que procura por símbolos "$", que definem a utilização de um registrador na instrução. Tendo em vista que o arquivo é lido linha por linha, na chamada da função *transforma_tipoR* foi passado como argumento a linha que está sendo analisada e, dentro desta função, foi chamado o método para identificar a quantidade daqueles caracteres identificadores de um registrador, a partir da leitura individual de cada caractere enquanto houvessem caracteres na linha. Como retorno desta função, há um vetor de inteiros "i", que identifica as posições onde há ocorrência do caractere, e, para saber a quantidade de vezes que um registrador qualquer aparece na função, basta utilizar do método **len(i)**, que retorna o tamanho do vetor de ocorrências.

A partir desse tamanho, analisa-se primeiro o grupo com 3 registradores operandos. Com exceção do registrador *$zero*, todos os demais são representados pelo símbolo "$", seguido de dois caracteres, segundo o manual [4]. Assim, possuindo um vetor com as posições dos caracteres dado pela função citada anteriormente, basta guardar em **rs**, **rt** e **rd** os cararacteres que vão da posição do "$" até os dois próximos algarismos, extaindo-se, assim, o valor literal destes.

Contudo, para a transformação dos registradores em um código binário, devemos extrair seu valor numérico. Assim, basta chamar a função com o dicionário de registradores, onde, passando-se seu nome literal, é feita uma comparação até que se encontre sua representação nos valores tabelados, retornando o inteiro referente à ele, na faixa de 0 a 31.

Com este valor traduzido, temos os valores para **rs**, **rt** e **rd**. Com relação ao campo **shamt**, nenhuma das instruções com 3 operandos o utiliza, então, por padrão, seu valor é setado em "00000". Por fim, é feito um tratamento para associar a operação ao seu valor de **func**, onde é identificada a instrução a partir do começo da linha e traduzida para seu valor tabelado.

Concluídos os cálculos dos campos acima citados, o código objeto nada mais é do que a concatenação dos valores obtidos na forma de uma string, traduzida posteriormente para um inteiro a partir da base 2 (*num = int(machineCode, 2)*) e traduzida novamente para o tipo string, agora convertido para a base 16 (*format(num, '08X')*).

Os cálculos para as demais instruções tipo R seguem o mesmo raciocínio, atentando-se ao fato de que, por possuírem menos registradores operandos, os valores dos registradores que não estão presentes na operação também são tabelados no manual [3].

O caso especial nos itens acima está presente nas operações *sll*, *srl* e *sra*, onde precisamos extrair, também, o campo **shamt**. Para tal, utiliza-se uma função auxiliar denominada *localiza_imediato*, que busca por todas as aparições de vírgula na instrução e, sabendo que o formato das instruções citadas segue "instr $reg1, $reg2, imediato", basta buscar tudo que aparece após a segunda vírgula, com o auxílio do vetor de posições que é retornado da função citada.

opcode (6)	rs (5)	rt (5)	imediato (16)

Table 5. Formatação do código-objeto para instruções do tipo I.

Por fim, a última exceção está relacionada com as instruções *madd* e *msubu*, que possuem **opcode** diferentes. Para resolver isso, basta atribuir o valor de seu opcode dentro da condição que lê o começo da linha e associa à instrução, anteriormente utilizada para diferenciar o campo **func** das operações. Em seguida, trataremos as instruções do tipo I, com a função *transforma_tipoI*.

3.3.2 Função *transforma_tipoI*

Da mesma forma descrita anteriormente, durante a leitura do arquivo linha por linha, as instruções identificadas como instruções do tipo I através das tabelas criadas chamam a função *transforma_tipoI*, com exceção dos **branches**, que serão tratados na próxima função junto com os **branches**.

Existem, dentro dessas instruções, algumas que utilizam dois registradores como operando e outras que utilizam apenas um. O tratamento para isso é semelhante ao da função anterior, onde localizamos os registradores e utilizamos o tamanho do vetor de posições para definir a quantidade. Dentro das instruções de 2 registradores, para completar os parâmetros que definem o código-objeto representado na Tabela 5, basta identificar o valor do **imediato**, da mesma forma que identificamos o **shamt** na função anterior, atentando-se para o fato de que agora, este valor deve ser formatado para 16 bits.

Nas instruções com apenas 1 registrador como operando, basta alterar diretamente o valor de **rs** ou **rt** diretamente para seu valor tabelado, dependendo de qual instrução estamos lidando.

No caso da pseudoinstrução **li**, falaremos dela um pouco mais a frente, pois foi criada uma metodologia para lidar com ela dependendo do valor do imediato passado. Agora, veremos como funciona a instrução para **jumps** e **branches**.

3.3.3 Função *transforma_jump_branch*

Para diferenciar esses dois tipos de operação mecanicamente, temos que **branches** utilizam pelo menos um registrador como operando e **jumps** não utilizam, dentro das instruções solicitadas para implementação. Assim, ao atribuir o vetor de posições, temos que, caso seu tamanho seja maior do que 0, estamos tratando do primeiro caso e, caso seja igual a zero, estamos tratando do segundo.

Dentro das possibilidades de branch, temos operações com 2 registradores (beq e bne) e operações com 1 registrador (bgez e bgezal). A identificação dos registradores é feita da mesma forma que calculamos anteriormente para instruções do tipo R e do tipo I; contudo, temos um novo fator a trabalhar, onde iremos utilizar, para o cálculo do campo de imediato do código objeto, a posição relativa entre a chamada da instrução de branch e a posição da primeira instrução da **label** a qual ela se refere.

Para identificar a label dentro da instrução de branch, utilizamos a mesma ferramenta que criamos para localizar o imediato, com a diferença de que agora estamos guardando, qualitativamente, uma *string*, e não um inteiro. Para não precisar procurar por exatamente 1 ou 2 aparições de vírgula (em se tratando com instruções com 2 ou 3 operandos), utilizamos o valor relativo do total de aparições no vetor para pegar a posição da última vírgula, e assim extrair o que vem após ela (label).

Ao identificar o valor da label, precisamos percorrer novamente o arquivo desde o início para localizar em qual linha está localizada, procurando-se por linhas que começam exatamente com o mesmo valor guardado em **label**. Uma informação importante é que o valor das linhas é incrementalmente contado a cada leitura de linha no arquivo dentro do loop principal da função *main*, então, outro parâmetro passado para essa função *transforma_jump_branch* é o valor da linha em que a função está.

Logo, ao localizar a aparição da **label** na abertura do arquivo dentro desta função, temos que o *offset* a ser utilizado como último parâmetro do código-objeto dessas instruções é dado pela distância entre a linha da **label** e a linha da instrução que a convoca. Assim, ainda de acordo com a Tabela 5 (pois ainda estamos tratando de instruções do tipo I), temos que a composição do código objeto está completa.

Agora, trataremos das instruções de **jump**. Nesta, temos a facilidade de que o único operando da instrução é a própria label. Assim, como já precisaríamos separar as instruções pelo nome para a atribuição do opcode, utilizamos essa mesma condicional para buscar pela label, onde, na instrução **j** ela pode ser encontrada a partir da leitura da linha a partir de seu segundo caractere (em uma linguagem de alto nível como **Python**, temos a facilidade de lidar com a string como sendo um vetor de caracteres, onde podemos acessar esses algarismos a partir da atribuição de um *index* ao vetor) e em **jal**, a partir do quarto.

Após identificar esse operando, buscamos o endereço onde a primeira instrução da label chamada está, da mesma forma que fizemos nas operações de branch. Contudo, para instruções de jump, temos que o imediato é calculado a partir da posição absoluta da label (endereço), com um *shift* de dois bits para a direita, segundo o manual [3]. Logo, ao identificar a linha em que a label está, calculamos o endereço dela a partir do endereço da instrução que a chama, com adição da diferença de linhas de código entre ambas multiplicado por 4, pois os endereços para instruções incrementam em 4 unidades a cada linha.

opcode (6)	imediato (26)

Table 6. Formatação do código-objeto para instruções do tipo J.

Por fim, formatamos esse endereço para 26 bits e concatenamos com o opcode da instrução referente, gerando, assim, o código objeto para instruções do tipo J, segundo a Tabela 6.

3.3.4 Função para a instrução *li*

Neste caso em específico, lidamos da mesma forma que o software MARS lida com a pseudoinstrução *li*: caso o imediato solicitado tenha uma magnitude superior a 16 bits, essa instrução é traduzida para as instruções que a compõem - *lui* e *ori*; caso tenha uma magnitude inferior, traduzimos para a instruição *addiu*, com o auxílio do registrador $zero. O segundo caso nada mais é do que a formação do código objeto da mesma forma de uma instrução do tipo R, solicitando apenas o registrador que estava na instrução *li* e o imediato, preenchendo o outro registrador (**rs**) com $zero, logo, segue a metodologia utilizada na função *transforma_tipoR*.

Agora, para o primeiro caso, temos que utilizar da função *insere_li*. Para isso, devemos verificar que, para lidar com os endereços de forma correta, no caso da substituição de uma instrução por outras duas, temos que abrir espaço para dois endereços, mantendo a continuidade do programa. Assim, a incrementação pré-função realizada recorrentemente, agora, passa a ser em +8 unidades para casos em que há uma função *li* com um imediato de uma magnitude superior a 16 bits. Assim, passamos para a função *insere_li* também o endereço atual incrementado em 8.

Dentro desta função, basicamente calculamos o código objeto para a função *lui*, utilizando, segundo o MARS, no registrador **rt** o $at e no registrador **rs** o $zero, e no valor do imediato, utilizamos os 16 bits mais significativos do imediato original da instrução *li*, através de um **shift** para a direita em 16 bits, formatado para 16 bits. O endereço para esta instrução, visto que incrementamos anteriormente em 8 o endereço da instrução anterior a *li* será o endereço atual subtraido de 4 unidades.

Em seguida, calculamos o código objeto para a função *ori*. Nela, utilizamos em **rt** o registrador operando de *li*, em **rs** o registrador $at e no imediato o valor dos 16 bits menos

significativos do número inicial de *li*, calculado a partir deste e uma operação **and** com a máscara 0x0000FFFF, formatado para 16 bits. O endereço para essa instrução, então, é o próprio endereço passado como parâmetro, visto que está 4 unidades a frente do endereço de *lui*, como esperado.

3.4. Área *.text*

3.5. Formatações para o arquivo .MIF

Para a formatação do arquivo .MIF segundo a referência [2], alguns pontos devem ser verificados. Em primeiro lugar, os arquivos de saída, dentro desse formato, são compostos pelo cabeçalho (onde, para o projeto, foi solicitada a utilização dos valores padrões do **MARS**, então não houveram necessidades de cálculos, apenas sua inserção) e seu conteúdo, classificado na forma "ADDRESS : DATA". Durante a execução do programa, ambos os valores são conhecidos e calculados a partir dos endereços iniciais para os campos *.data* e **.text** do MARS. A cada novo dado ou instrução (uma nova leitura de linha do arquivo que contivesse algum desses conteúdos em seus campos separados), o valor do endereço fora incrementado em 4 unidades. Assim, a cada laço dentro da leitura do arquivo de entrada, possuímos endereço e dado referentes. Logo, assim que obtemos o código objeto referente à instrução da linha, já escrevemos diretamente no arquivo de saída referente, junto do endereço relativo a este, incrementado em 4 anteriormente ao cálculo do código.

Seguindo o exemplo proposto, foi adicionado, também, uma parcela para um comentário, incluindo a instrução que foi utilizada para gerar o código-objeto.

É importante ressaltar que, durante a execução do código, é checado se a primeira área a ocorrer no arquivo de entrada é a área *.data* ou a área *.text*, pois exigem metodologias diferentes para a transformação de seus dados em código objeto. Para fazer isso, foi, a partir de uma leitura do arquivo de entrada, identificadas as linhas onde essas chamadas de áreas acontecem e, a partir da comparação do número da sua linha de ocorrência, verificado qual ocorre primeiro e rearranjada a forma de processamento do arquivo, onde, caso ocorra *.data* primeiro, ao encontrar uma linha escrita *.text* a execução passa a ser referente ao campo de instruções, e vice-versa.

4. Resultados

O resultado do programa segue as diretrizes solicitadas e as respostas esperadas. Utilizando como base o arquivo exemplo passado junto do roteiro do projeto, temos a seguinte resposta na execução do programa:

4.1. Arquivo .MIF para o campo text

```
DEPTH = 4096;
WIDTH = 32;
ADDRESS_RADIX = HEX;
DATA_RADIX = HEX;
CONTENT
BEGIN

00400000 : 3C011001;  % 1: lui $at, 0001000000000001 %
00400004 : 34280000;
% 1: ori $t0, $at, 0000000000000000 %
00400008 : 8D090000;  % 2: lw $t1, 0($t0) %
0040000C : 8D0A0004;  % 3: lw $t2, 4($t0) %
00400010 : 8D0B0008;  % 4: lw $t3, 8($t0) %
00400014 : 71404821;  % 5: clo $t1, $t2 %
00400018 : 014B4820;  % 6: add $t1, $t2, $t3 %
0040001C : 012A6026;  % 7: xor $t4, $t1, $t2 %
00400020 : 218D000A;  % 8: addi $t5, $t4, 10 %
00400024 : 39AE0014;  % 9: xori $t6, $t5, 20 %
00400028 : AD0C0000;  % 10: sw $t4, 0($t0) %
0040002C : AD0D0004;  % 11: sw $t5, 4($t0) %
00400030 : AD0E0008;  % 12: sw $t6, 8($t0) %

END;
```

No arquivo de saída, os comentários estão sempre na mesma linha do código objeto, porém na disponibilização no relatório, alguns casos excederam o tamanho da listagem para código, por isso houve a quebra de linha.

4.2. Arquivo .MIF para o campo .data

```
DEPTH = 16384;
WIDTH = 32;
ADDRESS_RADIX = HEX;
DATA_RADIX = HEX;
CONTENT
BEGIN

10010000 : 00000001;
10010004 : 00000002;
10010008 : 00000003;

END;
```

Alguns outros códigos foram testados para englobar outras instruções e possíveis desvios de comportamento do arquivo de entrada, como, por exemplo, o seguinte código:

```
.text

beq $t0, $t1, label
li $t0, 0x10010000
lw $t1, 0($t0)
lw $t2, 4($t0)
label1:
lw $t3, 8($t0)
clo $t1, $t2
label:
add $t1, $t2, $t3
li $t2, 0xFF02
xor $t4, $t1, $t2
bgezal $gp, label1
addi $t5, $t4, 10
li $t2, 4
xori $t6, $t5, 20
sw $t4, 0($t0)
sw $t5, 4($t0)
sw $t6, 8($t0)

.data
a: .word 1, 2, 3
b: .word 0xF, 1, 7
c: .word 1, 2, 3
```

Tipo R	Tempo de exeução
	0.011683 ms
	0.013351 ms
	0.011444 ms

Table 7. Tabela das médias de tempo de execução de instruções do tipo R.

Tipo I	Tempo de exeução
	0.014544 ms
	0.013113 ms
	0.012159 ms

Table 8. Tabela das médias de tempo de execução de instruções do tipo I.

Neste, algumas exceções foram levadas em conta, como a inversão da ordem dos campos de texto e dado; a diferença do código objeto para a função **beq** quando, entre ela e sua respectiva *label*, houvesse a presença da instrução **li**, cujo número de instruções que a compõe muda de acordo com o imediato recebido (podendo ser composta por 1 ou 2 instruções diferentes) e afeta diretamente a distância relativa do branch à label; diversas declarações de dados em linhas separadas, incluindo sua representação em hexadecimal.

Diversos arquivos foram testados para menores correções e o programa respondeu positivamente para todos.

4.3. Análise de desempenho

Podemos considerar a aplicação como sendo um laço principal de leitura do arquivo que, ao encontrar uma instrução, realiza sua identificação e tratamento. Assim, temos que a maior parcela de trabalho na execução do código está justamente nesse processamento, que transforma uma linha de caracteres de instrução em seu código objeto.

No programa, temos 3 tipos de tratamentos, como explicados no item referente à metodologia: *transforma_tipoR*, *transforma_tipoI* e *transforma_jump_branch*. Utilizando-se o mesmo arquivo de entrade, e com o auxílio da biblioteca "time", importada no começo do código, o tempo de execução entre uma linha antes da chamada dessas funções e uma linha após a atribuição de seu retorno foi medido 15 vezes, gerando uma média a cada 5 execuções, apresentadas nas Tabelas 7, 8 e 9.

Para entender melhor esse resultado, foi analisado novamente o desempenho, agora com um código exclusivamente composto de instruções de jumps e branchs:

Branch/Jump	Tempo de exeução
	0.410557 ms
	0.369549 ms
	0.372648 ms

Table 9. Tabela das médias de tempo de execução de instruções de branch/jump.

Branch/Jump	Tempo de exeução
bne	0.344992 ms
beq	0.364542 ms
bgez	0.377417 ms
bgezal	0.432491 ms
j	0.480413 ms
jal	0.356674 ms

Table 10. Tabela das médias de tempo de execução de cada instrução de branch/jump.

```
.text
LABEL:
bne $t1, $zero, LABEL1
LABEL1:
beq $t1, $zero, LABEL2
LABEL2:
bgez $t1, LABEL3
LABEL3:
bgezal $t1, LABEL4
LABEL4:
j LABEL5
LABEL5:
jal LABEL6
LABEL6:
add $t0, $t1, $t1
```

A partir dessa execução, foram obtidos os resultados presentes na Tabela 10.

5. Discussão e Conclusões

Comparando-se os resultados, foi percebido que o tempo de execução entre as instruções do tipo R e do tipo I não foi muito diferente, sendo a segunda um pouco maior. Com relação às instruções de branch e jump, foi notada uma grande diferença no tempo de execução, o que segue o esperado, visto que, dentro desta, independente da instrução que estivermos lidando, devemos fazer uma nova varredura no arquivo para a localização de posições (relativa ou absoluta).

Com a análise exclusiva das instruções de branchs e jumps, foi percebido que a média delas é semelhante, o que segue a lógica de terem sido separadas em uma mesma função por possuírem cálculos parecidos.

Algumas propostas podem se adequar para o aprimoramento do código, como a presença de variáveis globais que possam ser utilizadas para marcar ocorrências de jumps e branchs, bem como de labels, evitando a necessidade de uma nova leitura do arquivo para coletá-las. Outra possibilidade seria a atribuição do arquivo em uma lista, na qual, em casos de busca por labels, poderia ser

realizado um algoritmo de busca binária, cuja complexidade, levando em consideração cada item da lista como sendo uma linha do arquivo de entrada, é de O(log_n) em seu pior caso, enquanto a busca realizada atualmente é linear, com complexidade O(n).

References

[1] Patterson, D.A., Hennessy, J.L., Computer Organization and Design – The Hardware/Software Interface, Fourth Edition, Mourgan Kaufmann, 2009;

[2] Memory Initialization File (.mif), Intel, `https://www.intel.com/content/www/us/en/programmable/quartushelp/13.0/mergedProjects/reference/glossary/def_mif.htm`, Acesso em: 21 set 2020, 09:43.

[3] MIPS® Architecture for Programmers Volume II-A: The MIPS32® Instruction Set Manual, Revision 6.06, MIPS, 2016.

[4] MIPS32® Instruction Set Quick Reference. Wave Computing, Inc. Revision 01.01.

Observações - As funções utilizadas na linguagem *Python* foram retiradas diretamente de seu manual, que pode ser visualizado em `https://doc.bccnsoft.com/docs/python-3.5.2-docs-html/`.