	Instituto Federal de Educação, Ciência e Tecnologia Curso: Bacharelado em Ciência da Computação Disciplina: Arquitetura e Organização de Computadores Professor: Mário Luiz Rodrigues Oliveira Atividade: 1ª Trabalho Prático Formiga, MG, 24 de outubro de 2016

INSTRUÇÕES:

1. **Esta atividade deve, obrigatoriamente, ser resolvida individualmente.**
2. **Caso você ache que falta algum detalhe nas especificações, você deverá fazer as suposições que julgar necessárias e escrevê-las no seu relatório. Pode acontecer também que a descrição dessa atividade contenha dados e/ou especificações supérfluas para sua solução. Utilize sua capacidade de julgamento para separar o supérfluo do necessário.**
3. **Como produtos da atividade serão gerados três artefatos: códigos fontes de cada implementação, arquivo *MakeFile* e documentação da atividade.**
4. **Cada arquivo-fonte deve ter um cabeçalho constando as seguintes informações: nome autor(a), matrícula e data.**
5. **O arquivo contendo a documentação da atividade (relatório) deve ser devidamente identificado com o nome e matrícula do(a) autor(a) do trabalho. O arquivo contendo o relatório deve, obrigatoriamente, estar no formado PDF.**
6. **Devem ser entregues os arquivos contendo os códigos-fontes, arquivo *MakeFile* e o arquivo contendo a documentação da atividade (relatório). Compacte todos os artefatos gerados num único arquivo no formato RAR.**
7. **A atividade deve ser entregue, obrigatoriamente, via portal acadêmico acessado pela URL:https://meu.ifmg.edu.br/.**
8. **O prazo final para entrega desta atividade é até 23:59:00 do dia 05/12/2016.**
9. **O envio é de total responsabilidade do aluno. Não serão aceitos trabalhos enviados fora do prazo estabelecido.**
10. **Trabalhos plagiados serão desconsiderados, sendo atribuída nota 0 (zero) a todos os envolvidos.**
11. **O trabalho deve, obrigatoriamente, ser implementado na linguagem de programação C.**
12. **O valor desta atividade é 40 pontos. A implementação do montador é valorada em 20 pontos e a implementação do simulador é valorada em 20 pontos.**

	Instituto Federal de Educação, Ciência e Tecnologia Curso: Bacharelado em Ciência da Computação Disciplina: Arquitetura e Organização de Computadores Professor: Mário Luiz Rodrigues Oliveira Atividade: 1ª Trabalho Prático Formiga, MG, 24 de outubro de 2016

Introdução

Este trabalho visa a implementação de um montador e um simulador funcional para o processador RISC de 32 bits IFMG-RISC. Este processador possui 32 registradores de uso geral e 32 instruções (23 instruções projetadas pelo professor e 9 instruções projetadas pelo autor).

Cada instrução executada no IFMG-RISC demora quatro ciclos para completar. Esses ciclos de execução são denominados: IF, ID, EX/MEM e WB.

- IF (*instruction fetch*): a próxima instrução a ser executada é buscada na memória e armazenada no registrador de instruções (IR), neste ciclo incrementa-se o contador de programa (PC);
- ID (*instruction decode*): a instrução sendo executada é decodificada e os operandos são buscados no banco de registradores;
- EX/MEM (*execute and memory*): a instrução é executada e as condições dos resultados são "setados" (condition codes) para operações de ALU. Loads, stores, o cálculo do endereço efetivo e a resolução de branches são feitos também nesse ciclo;
- WB (*write back*): os resultados são escritos no banco de registradores.

1 Implementação do Montador para o IFMG-RISC

Um montador é um programa quer converte instruções simbólicas em instruções binárias. Existem basicamente duas maneiras de implementar um montador, a saber: montador de um passo e montador de dois passos. Nesse trabalho usaremos a segunda opção e portanto deve-se implementar um montador de 2 passos conforme sugerido na seção 7.3 referência TANENBAUM.

O montador deve ler um arquivo contendo instruções simbólicas do processador descrito nesse documento e produzir um arquivo de saída em formato compatível com o simulador projetado e implementado. Os nomes dos arquivos de entrada e de saída devem ser passados ao programa montador como parâmetro na linha de comando.

2 Implementação do Simulador para o IFMG-RISC

O processador IFMG-RISC deverá possuir as seguintes características:

- barramento de dados de 32 bits e barramento de endereço de 16 bits;
- 32 registradores de uso geral de 32 bits de largura;
- 32 instruções;
- instruções de 3 operandos;
- memória endereçada a nível de palavra, ou seja, cada endereço de memória deve se referir a quatro bytes. No total, o processador deverá possuir 64k(2^{16}) endereços. Então, a memória total do processador será de 256 *KBytes* (64k endereços x 4 bytes).

Um dos requisitos fundamentais para se entender como implementar o simulador funcional do processador IFMG-RISC é a compreensão do termo funcional. Uma descrição funcional divide o processador nos blocos que existirão em uma implementação real. Portanto, o processador deverá conter quatro rotinas principais, uma para cada ciclo a ser executado em cada instrução. Note que a única forma de comunicação entre essas quatro rotinas em um processador real é via registradores, que no caso do simulador serão implementados por estruturas de dados no programa do simulador.

Além dessas quatro rotinas, os elementos principais do processador real deverão estar encapsulados por módulos, tais como os elementos: banco de registradores, ALU e memória.

2.1 Conjunto de Instruções

O processador IFMG-RISC possui o seguinte conjunto de instruções:

- instruções aritméticas que envolvem a ALU;
- instruções que envolvem o uso de constantes e instruções que acessam a memória;
- instruções de transferência de controle.

2.1.1 ALU

O ULFA-RISC possui as seguintes instruções de ALU:

OPCODE	Operação
00000001	adição de inteiros
00000010	Subtração de inteiros
00000011	Zero
00000100	Xor
00000101	Or
00000110	Not
00000111	And
00001000	shift aritmético para a esquerda
00001001	shift aritmético para a direita
00001010	shift para lógico a esquerda
00001011	shift lógico para a direita
00001100	Copia

Todas as instruções de ALU alteram os valores dos *flags* do processador: *neg*, *zero, carry* e *overflow*. As instruções lógicas, como *and* e *or,* zeram os *flags overflow* e *carry*. Veja a descrição das instruções no Apêndice B para maiores detalhes.

2.1.2 Constantes e Acesso à Memória

O processador IFMG-RISC possui somente instruções para a carga de constantes de 16 bits, para carga de uma posição de memória específica e para armazenar um valor numa posição de

memória específica.

OPCODE	Operação
00001110	carrega constante de 16 bits nos 2 bytes mais significativos
00001111	carrega constante de 16 bits nos 2 bytes menos significativos
00010000	carrega o conteúdo de uma posição de memória num registrador
00010001	armazena o conteúdo de um registrador numa posição de memória

2.1.3 Transferência de controle

O processador IFMG-RISC possui cinco instruções para a transferência de controle, codificadas de três maneiras diferentes. A primeira codificação é utilizada para instruções de desvios condicionais, a segunda codificação é utilizada para a instrução de desvios incondicionais e a terceira codificação também é utilizada para instruções de desvios incondicionais, porém o endereço de desvio é especificado por um registrador. As cinco instruções para transferência de controle são: *jump and link, jump register, , jump se igual, jump se diferente e jump incondicional.*

Na instrução *jump and link*, o próximo PC (PC + 1) deve ser armazenado no registrador r31 e o endereço do desvio armazenado em PC. Na instrução *jump register*, a única operação a ser feita é armazenar o conteúdo do registrador r*c* no registrador que armazena o valor de PC.

Nas instruções de *Jump Condicional* (*jump se igual ou jump se diferente*) o endereço da próxima instrução a ser executada, caso o *Jump* não seja realizado, é o valor do PC incrementado de 1. Assim, no ciclo EX/MEM, esse endereço pode ser obtido diretamente do registrador que armazena o valor de PC, pois nesse ciclo o PC já foi incrementado de um. Se o *Jump* for tomado, o fluxo de instruções deve ser transferido para o endereço de memória especificado na instrução.

Na instrução *jump incondicional*, a única operação a ser feita é armazenar o endereço do

desvio no registrador que armazena o valor de PC.

Abaixo os opcodes das instruções:

OPCODE	Operação
00100000	jump and link
00100001	jump register
00100010	jump se igual
00100011	jump se diferente
00100100	jump incondicional

2.1.4 NOP

Uma instrução onde todos os 32 bits estão zerados será utilizada para indicar a instrução NOP.

2.1.5 Condição de término

Uma instrução onde todos os 32 bits são iguais a 1 será utilizada para indicar a instrução HALT (parada do sistema).

3 Documentação

A documentação será de grande importância na avaliação de todo o trabalho. Não é necessário que seja extensa, porém deve ser completa.

Deve conter basicamente:

- Um resumo da máquina simulada;
- Um resumo do montador implementado;

- Decisões de implementação;
- Instruções projetadas e implementadas pelo autor do trabalho. O autor deve justificar as novas instruções projetadas.
- Tutorial de como usar o montador e o simulador, com instruções completas e precisas de como executá-lo;
- Estruturas de dados utilizadas;
- Listagem do código fonte do montador e do simulador;
- Testes realizados para a validação do montador e do simulador.

3.1 Testes

Os testes pelo qual o processador passa ajuda no desenvolvimento da sua correção e robustez. Como o objetivo primeiro do projeto é prezar pela sua correção, é esperado que se faça uso de uma bateria de testes para auxílio na depuração.

Como sugestão de testes úteis, temos:

- Testes Massivos: procurando testar o maior número de combinações de instruções possíveis;
- Programas Reais: implementações de programas comuns para simular situações reais.

4 Referências

1. FERNANDES, Antônio Otávio. RISC-II. DCC-UFMG, Belo Horizonte, 1999
2. HENNESSY, John L.; PATTERSON, David A. Arquitetura de Computadores: Uma abordagem quantitativa. 5. ed. São Paulo: Campus, 2014
3. PATTERSON, David A.; HENNESSY, John L. Organização e Projeto de Computadores: A interface Hardware/Software, 4. ed. São Paulo: Campus, 2005.

4. TANENBAUM, A. S. Organização Estruturada de Computadores. 5ª edição. São Paulo: Pearson Prentice Hall, 2007.

5 Apêndice A: Formato de Entrada/Saida do Simulador

5.1 Introdução

Esta seção define como deve ser o formato do arquivo de instruções para o processador IFMG-RISC e a saída do simulador. Convém salientar que o formato, que será descrito a seguir, deve ser adotado obrigatoriamente por todos.

Como especificado, a memória do processador possui 16 bits de endereçamento, sendo endereçada por palavra. Ou seja, a memória pode armazenar até 65536 palavras de 32 bits cada.

5.2 Descrição do formato

O formato do arquivo de instruções para o processador IFMG-RISC é bem simples. Podemos defini-lo através de apenas três características:

1. O arquivo, em modo texto, deve conter apenas uma instrução por linha;
2. As instruções devem estar codificadas em binário;
3. A palavra-chave **address** pode ser usada para definir a partir de qual posição de memória as instruções devem ser armazenadas. A sintaxe é a seguinte;
   address end, onde **end** é um endereço de memória codificado em binário.

Observar que, se a diretiva *address* não for usada para definir a posição inicial de armazenamento das instruções, será considerado, por convenção, que as instruções serão colocadas a partir da posição *zero* de memória.

5.3 Exemplo de um arquivo de instruções

A seguir é apresentado um arquivo de instruções que obedece o formato descrito no item 2.

address 0

00011000100000000001100010000000

00011000100000110001100010000000

00011000100000100001100010000000

00011000100000110001100010000000

address 1000001

00011000100000000001100010000000

00011000100000110001100010000000

00011000100000100001100010000000

00011000100000110001100010000000

Veja o conteúdo de memória após o processador carregar o arquivo de instruções acima:

ENDEREÇO DE MEMÓRIA	CONTEÚDO	ENDEREÇO DE MEMÓRIA	CONTEÚDO
0 (Posição 0)	0001100010000000 0001100010000000	1000001 (Posição 65)	00011000100000000 001100010000000
10 (Posição 1)	0001100010000011 0001100010000000	1000010 (Posição 66)	00011000100000110 001100010000000
10 (Posição 2)	0001100010000010 0001100010000000	1000011 (Posição 67)	00011000100000100 001100010000000
11 (Posição 3)	0001100010000011 0001100010000000	1000100 (Posição 68)	00011000100000110 001100010000000

5.4 Formato da Saída

Para verificar a corretude do simulador, o mesmo deverá apresentar na saída as modificações ocorridas nos registradores e nas posições de memória a cada ciclo de clock.

Considere o exemplo abaixo onde é executada uma instrução de carga.

LOAD R1,A;

No primeiro ciclo (IF) carrega-se a instrução no registrador IR, logo após este ciclo IR deve ter como conteúdo a instrução LOAD R1,A e PC deve ter como conteúdo a posição de memória da próxima instrução a ser executada.

No segundo ciclo (ID) a instrução é decodificada e não há alterações nos conteúdos de registradores/memória.

No terceiro ciclo (EX/MEM) a instrução é executada.

No quarto ciclo (WB) a instrução LOAD escreve o resultado em R1, logo após este ciclo o registrador R1 deve ter o valor da posição A da memória.

Agora considere um exemplo onde é executada uma instrução ADD

ADD R3,R2,R1

No primeiro ciclo (IF) carrega-se a instrução no registrador IR, logo após este ciclo IR deve ter como conteúdo a instrução ADD R3,R2,R1 e PC deve ter como conteúdo a posição de memória da próxima instrução a ser executada.

No segundo ciclo (ID) a instrução é decodificada e os registradores R1 e R2 são lidos, logo neste ciclo não há alterações nos conteúdos de registradores/memória.

No terceiro ciclo (EX/MEM) a instrução é executada.

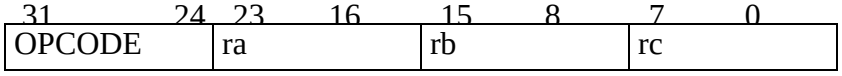
No quarto ciclo (WB) a instrução ADD escreve o resultado em R3, logo após este ciclo o registrador R3 deve ter como valor a soma dos conteúdos dos registradores R1 e R2.

6 Apêndice B: Descrição detalhada das instruções

Seguem abaixo as descrições das instruções implementadas pelo processador IFMG-RISC.

6.1 SOMA INTEIRA

add rc,ra,rb



Formato: add rc, ra, rb

Exemplo: add r3, r2, r1

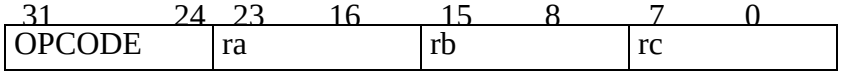
Operação: $rc = ra + rb$

Descrição: Soma (aritmética) o conteúdo de *ra* e de *rb* e coloca o resultado em *rc*.

Flags afetados: neg, zero, carry e overflow.

6.2 SUBTRAÇÃO INTEIRA

sub rc,ra,rb



Formato: sub rc, ra, rb

Exemplo: sub r3, r2, r1

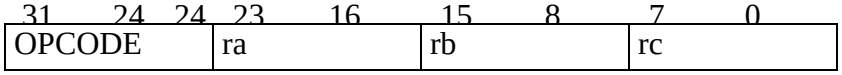
Operação: *rc = ra – rb*

Descrição: Subtrai o conteúdo de *rb* do conteúdo de *ra* e coloca o resultado em *rc*.

Flags afetados: neg, zero, carry e overflow.

6.3 ZERA

zeros rc



Formato: zeros rc

Exemplo: zeros r1

Operação: *rc* = 0

Descrição: Faz conteúdo de *rc* valer 0.

Flags afetados: neg = 0, zero = 1, carry = 0 e overflow = 0.

6.4 XOR LÓGICO

xor rc,ra,rb

OPCODE	ra	rb	rc

Formato: xor rc, ra, rb

Exemplo: xor r3, r2, r7

Operação: *rc = ra^ rb*

Descrição: Efetua *xor* lógico bit a bit de *ra* e *rb* e coloca resultado em *rc*.

Flags afetados: neg, zero, carry = 0 e overflow = 0.

6.5 OR LÓGICO **or rc,ra,rb**

31 24 24 23 16 15 8 7 0

OPCODE	ra	rb	rc

Formato: or rc, ra, rb

Exemplo: or r3, r2, r7

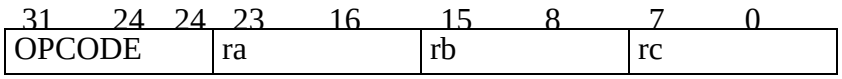
Operação: *rc = ra|rb*

Descrição: Efetua or lógico bit a bit de *ra* e *rb* e coloca resultado em *rc*.

Flags afetados: neg, zero, carry = 0 e overflow = 0.

6.6 NOT *ra*

not rc,ra



Formato: not rc, ra

Exemplo: not r4, r3

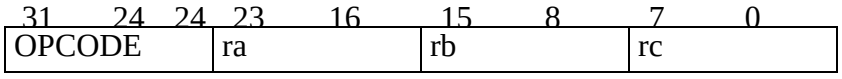
Operação: $rc = !ra$

Descrição: Faz conteúdo de *rc* valer o complemento do conteúdo de *ra*.

Flags afetados: neg, zero, carry = 0 e overflow = 0.

6.7 AND LÓGICO

and rc,ra,rb



Formato: and rc, ra, rb

Exemplo: and r3, r2, r7

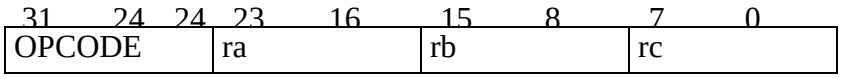
Operação: $rc = ra \& rb$

Descrição: Efetua and lógico bit a bit de *ra* e *rb* e coloca resultado em *rc*.

Flags afetados: neg, zero, carry = 0 e overflow = 0.

asl rc,ra,rb

6.8 SHIFT ARITMÉTICO PARA A ESQUERDA



Formato: asl rc, ra, rb

Exemplo: asl r3, r2,r5

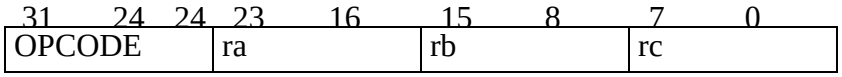
Operação: rc=ra<<rb

Descrição: Coloca cada bit ra_i em rc_{i+1} e preenche com 0 a posição rc_i. O registrador rb indica quantos bits serão deslocados

Flags afetados: neg, zero, carry = 0 e overflow = 0.

6.9 SHIFT ARITMÉTICO PARA A DIREITA

asr rc,ra,rb



Formato: asr rc, ra, rb

Exemplo: asr r3, r2,r5

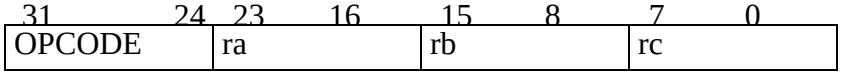
Operação: rc=ra>>rb

Descrição: Coloca cada bit ra_i em rc_{i-1} e preenche com o valor do bit ra_{31} as posições deslocadas. O registrador rb indica quantos bits devem ser deslocados.

Flags afetados: neg, zero, carry = 0 e overflow = 0.

6.10 SHIFT LÓGICO PARA A ESQUERDA

lsl rc,ra,rb



Formato: lsl rc, ra,rb

Exemplo: lsl r3, r2,r1

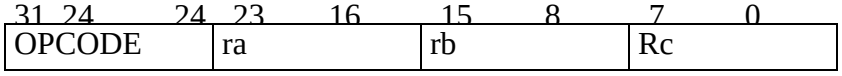
Operação: rc= ra << rb

Descrição: Coloca cada bit ra_i em rc_{i+1} e preenche com 0 as posições deslocadas. O registrador rb indica quantos bits devem ser deslocados.

Flags afetados: neg, zero, carry =0 overflow = 0.

6.11 SHIFT LÓGICO PARA A DIREITA

Lsr rc,ra,rb



Formato: lsr rc, ra,rb

Exemplo: lsr r3, r2,r1

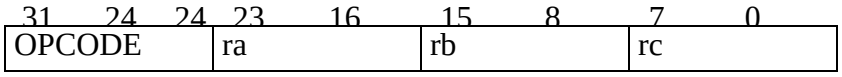
Operação: rc = ra >> rb

Descrição: Coloca cada bit ra_i em rc_{i-1} e preenche com 0 as posições deslocadas. O registrador rb indica quantos bits devem ser deslocados.

Flags afetados: neg, zero. carry = 0. overflow = 0.

6.12 COPIA *ra*

passa rc,ra



Formato: passa rc, ra

Exemplo: passa r4, r3

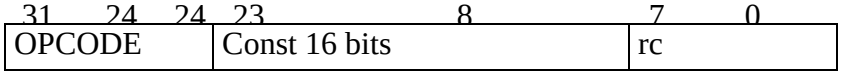
Operação: *rc = ra*

Descrição: Faz conteúdo de *rc* igual ao conteúdo de *ra*.

Flags afetados: neg, zero, carry = 0 e overflow = 0.

6.13 CARREGA CONSTANTE NOS 2 BYTES MAIS SIGNIFICATIVOS

lch c, Const16



Formato: lcl rc, Const16

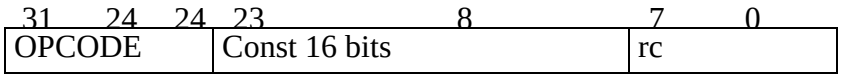
Exemplo: lcl r3, Const 16

Operação: rc =(Const16 << 16) | (rc & 0x0000ffff)

Descrição: Carrega nos 2 bytes mais significativo de *rc* o conteúdo de Const16.

Flags afetados: Nenhum.

6.14 CARREGA CONSTANTE NOS 2 BYTES MENOS SIGNIFICATIVOS — lcl c, Const16



Formato: lcl rc, Const16

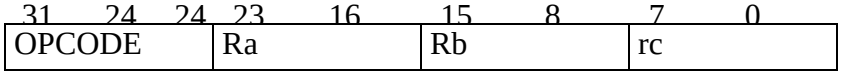
Exemplo: lcl r3, Const16

Operação: rc=Const16 | (rc & 0xffff0000)

Descrição: Carrega nos 2 bytes menos significativos de *rc* o conteúdo de Const16.

Flags afetados: Nenhum.

6.15 LOAD WORD — load rc, ra



Formato: load rc, ra

Exemplo: load r3, r6

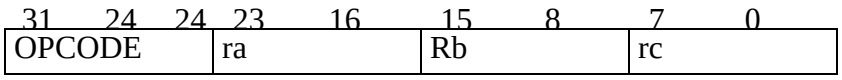
Operação: *rc* = memória [*ra*]

Descrição: Carrega no registrador *c* o conteúdo da memória endereçada pelo registrador *a*.

Flags afetados: nenhum.

6.16 STORE WORD

store rc, ra



Formato: store rc, ra

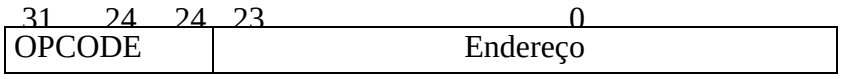
Exemplo: store r3, r6

Operação: memória [*rc*] = ra

Descrição: carrega na posição da memória endereçada pelo registrador rc o conteúdo do registrador ra.

Flags afetados: nenhum.

6.17 JUMP AND LINK **jal end**



Formato: jal end

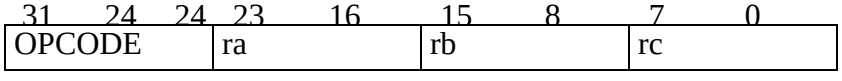
Exemplo: jal 100

Operação: faça r31 = PC e PC =end

Descrição: Realiza chamadas a procedimentos guardando o endereço do PC atual no registrador
R31 (para o retorno após o procedimento) e colocando em PC o valor do endereço do desvio (primeira instrução do procedimento).

Flags afetados: Nenhum.

6.18 JUMP REGISTER **jr rc**



Formato: jr rc

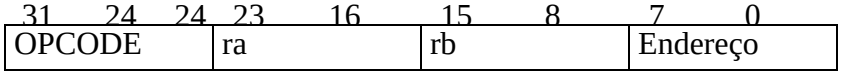
Exemplo: jr r31

Operação: faça PC = r31.

Descrição: Armazena o conteúdo do registrador rc em PC.

Flags afetados: Nenhum.

6.19 JUMP SE IGUAL **beq ra,rb, end**



Formato: beq ra,rb,end

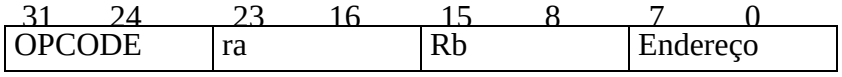
Exemplo: beq r7,r8,34

Operação: faça PC=end, se o jump for tomado

Descrição: Realiza desvio de fluxo condicional. Compara o conteúdo dos registradores ra e rb e se forem iguais o registrador PC recebe o endereço de memória para o qual o fluxo de execução deve ser transferido.

Flags afetados: Nenhum.

6.20 JUMP SE DIFERENTE **bne ra,rb, end**



Formato: beq ra,rb,end

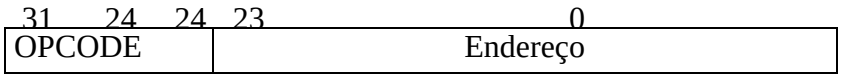
Exemplo: beq r7,r8,34

Operação: faça PC=end, se o jump for tomado

Descrição: Realiza desvio de fluxo condicional. Compara o conteúdo dos registradores ra e rb e se forem diferentes o registrador PC recebe o endereço de memória para o qual o fluxo de execução deve ser transferido.

Flags afetados: Nenhum.

6.21 JUMP INCONDICIONAL **j Destino**



Formato: j Destino

Exemplo: j loop

Operação: faça PC = Endereço

Descrição: Realiza desvio de fluxo incondicional. O endereço da próxima instrução a ser executada será o endereço de memória presente na instrução.

Flags afetados: Nenhum.