DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

UNIVERSIDADE FEDERAL DE LAVRAS

ARQUITETURA DE COMPUTADORES
PROFESSOR: Luiz Henrique Andrade Correia

μRISC-IV

DATA DE ENTREGA DO SIMULADOR FUNCIONAL: 06/12/2019

Sumário

1 Introdução

Este trabalho visa a implementação de um simulador funcional para o processador RISC de 16 bits μRISC. Este processador possui 16 registradores de uso geral e 42 instruções.

Cada instrução executada no μRISC demora quatro ciclos para completar, embora 4 instruções possam estar sendo executadas paralelamente. 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);
- 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.

2 Implementação do Simulador para o μRISC

O processador μRISC deverá possuir as seguintes características:

- 16 bits;
- 8 registradores de uso geral de 16 bits de largura;
- 42 instruções;
- instruções de 3 operandos;
- *big endian*;
- memória endereçada a nível de palavra, ou seja, cada endereço de memória deve se referir a dois bytes. No total, o processador deverá possuir 64k (2^{16}) endereços. Então, a memória total do processador será de 128KB (64k endereços x 2 bytes).

Um dos requisitos fundamentais para se entender como implementar o simulador funcional do processador μRISC 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, ALUs, incrementadores e memória.

A implementação deve ser feita em C ou C++.

2.1 Conjunto de Instruções

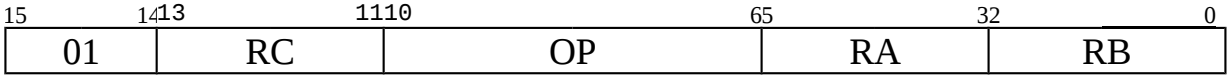
O processador μRISC possui o seguinte conjunto de instruções:

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

2.1.1 ALU

As instruções de ALU seguem o seguinte formato:

Formato I:



OP_{bin}	Operação	Mnemônico
00000	C = 0	zeros c
00010	C = A & B	and c,a,b
01010	C = !A & B	andnota c,a,b
01001	C = A	passa c,a
00110	C = A ^ B	xor c,a,b
00100	C = A | B	or c,a,b
00101	C = !A & !B	nor c,a,b
00111	C = !A ^ B	xnor c,a,b
01000	C = !A	passnota c,a
01011	C = A | !B	ornotb c,a,b
00011	C = !A | !B	nand c,a,b
00001	C = 1	ones c
11000	C = A + B	add c,a,b
11010	C = A + B + 1	addinc c,a,b
11100	C = A + 1	inca c,a
11011	C = A - B - 1	subdec c,a,b
11001	C = A - B	sub c,a,b
11101	C = A - 1	deca c,a
10000	C = shift lógico para a esquerda	lsl c,a
10010	C = shift lógico para a direita	lsr c,a
10011	C = shift aritmético para a dir.	asr c,a
10001	C = shift aritmético para a esq.	asl c,a

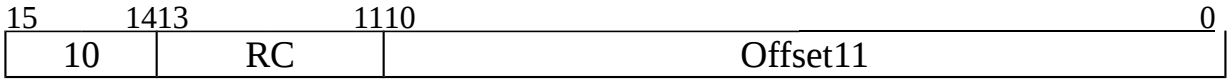
Todas as instruções de ALU alteram os valores dos *flags* do processador: *neg*, *zero*, *carry* e *overflow*. As instruções lógicas, como *and* e *or,* zeram os *flags overflow* e *carry*. Veja a descrição das instruções no Apêndice D para maiores detalhes.

2.1.2 Constantes

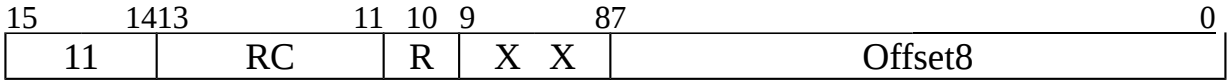
O processador μRISC possui somente instruções para a carga de constantes com sinal, representadas pelas instruções indicadas abaixo:

Formato	Operação	Mnemônico
II	C = Constante	loadlit c, Const11
III	C = Const8 | (C & 0xff00)	lcl c, Const8
III	C = (Const8 << 8) | (C & 0x00ff)	lch c, Const8

Formato II:



Formato III:



R	Operação
0	lcl
1	lch

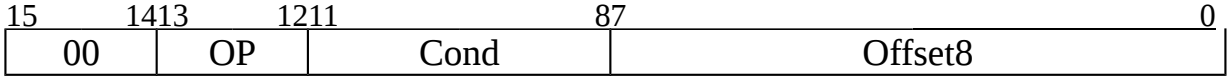
Na instrução *loadlit*, antes de ser carregada em um registrador, a constante precisa ter primeiro seu sinal estendido.

Para estender uma constante com sinal de *k* bits(de *0* a *k-1*) para *n*(*n* > *k*) bits, copia-se o bit *k-1* da constante (ou seja, o bit de sinal) para os bits *k* a *n-1* da nova constante. Exemplo: seja a constante de 8 bits com sinal 11110101.
Em 16 bits, esta constante é representada como 1111111111110101.
Já a constante 01110101 sem sinal é representada como 0000000001110101.

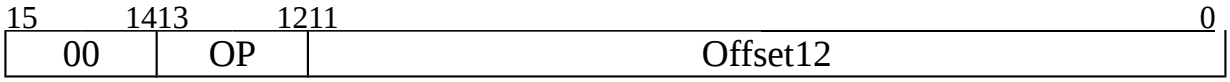
2.1.3 Transferência de controle

O processador μRISC possui cinco instruções para a transferência de controle, codificadas de três maneiras diferentes, que são indicadas abaixo. A primeira codificação é utilizada para instruções de desvios condicionais (onde o desvio é relativo ao PC e de comprimento de 8 bits) e a segunda codificação é utilizada para a instrução de desvios incondicionais (onde o desvio é também relativo ao PC e de comprimento de 12 bits).

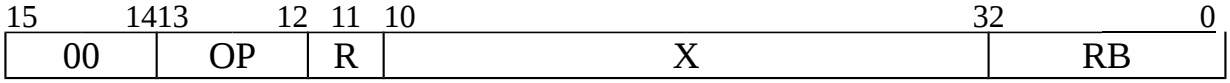
Formato IV:



Formato V:



Formato VI:



Formato	OP_{bin}	Operação	Mnemônico
IV	00	Jump False	jf.cond Destino
IV	01	Jump True	jt.cond Destino
V	10	Jump Incondicional	j Destino
VI	11	Jump and Link	jal b
VI	11	Jump Register	jr b

No caso da instrução de *jump and link*, o próximo PC(PC + 1) deve ser armazenado no registrador r7 e o conteúdo do registrador *b* armazenado em PC. Já no caso da instrução *jump register*, a única operação a ser feita é armazenar o conteúdo do registrador *b* no registrador PC.

R	Operação
0	Jump and Link
1	Jump Register

$COND_{bin}$	Condição	Mnemônico
0100	Resultado ALU negativo	.neg
0101	Resultado ALU zero	.zero
0110	Carry da ALU	.carry
0111	Resultado ALU negativo ou zero	.negzero
0000	TRUE	.true
0011	Resultado ALU *overflow*	.overflow

Nas instruções de *Jump True*, *Jump False* e *Jump* o **desvio é relativo ao PC.** O endereço da próxima instrução a ser executada, caso o *Jump* seja realizado, é o valor do PC incrementado de 1 mais a constante estendida de sinal. Assim, no ciclo EX/MEM, esse endereço pode ser obtido somente somando o PC com a constante, pois nesse ciclo o PC já foi incrementado de um.

2.1.4 Memória

Formato VII:

OP_{bin}	Operação	Mnemônico
10100	C = Mem[A]	load c, a ld c, a
10110	Mem[A] = B	store a, b st a, b

Observação:

1. Na instrução de **load**, o registrador **RB** não tem efeito.
2. Para a instrução de **store**, o registrador **RC** não tem efeito.

2.1.5 Condição de término

A instrução HALT (parada do sistema) deve ser implementada como **L: j L**, isto é, desvio incondicional para o mesmo endereço do desvio.
O código da instrução de HALT é ***0xffff, última posição da memória.***

2.2 Sintaxe de chamada

O programa do simulador deve aceitar um mínimo de parâmetros e opções:

```
uRISC <arq_entrada> [-d <pos_inicial>  <numero_palavras>] [-s] [-p],
```

onde :

`<arq_entrada>`: é o nome do arquivo em binário com o código a ser executado pela máquina;

`-d` : opção para pedido de dump memória a partir de `<pos_inicial>`(em hexadecimal) que deve imprimir então (na saída padrão) `<numero_palavras>` de palavras (em hexadecimal) sequencialmente. Essa operação deve ser executada no final da execução do arquivo de instruções;

`-s` : *screen*, opção para pedido de impressão na saída padrão de um resumo do estado do processador após a execução de cada instrução;

`-p` : opção que faz pausar a impressão cada vez que a tela 'enche' esperando que alguma tecla seja acionada.

Exemplos:

```
# execução simples #
lhacorrreia:~-> uRISC teste1

# execução com pedido de dump de 1 palavra a partir de 0x0100 #
lhacorreia:~-> uRISC teste1 -d 03e8 1

# execução com pedido de dump e de screen #
lhacorreia:~-> uRISC teste1 -d 03e8 1 -s

# execução anterior com pedido de pausa #
lhacorreia:~-> uRISC teste1 -d 03e8 1 -s -p

# execução com pedido de screen com pausa #
lhacorreia:~-> uRISC teste1 -s -p
```

Veja um exemplo(é apenas uma sugestão!) para a opção *screen*:

```
--------------------------------------------------------------------------
                        Status do Processador

        REGISTRADORES                          Flags

        Reg00 = 0000                    Neg             0
        Reg01 = 00aa                    Zero            1
        Reg02 = 0190                    Carry           0
        Reg03 = 7fff                    NegZero         0
        Reg04 = 23e0                    Overflow        1
        Reg05 = 7caf
        Reg06 = 0010
        Reg07 = 0015
```

```
                    PC    = 00a1
                    IR    = 8807

                Instrução executada = 00a0
-----------------------------------------------------------------
```

NOTA: Deve-se ter uma atenção toda especial para a saída do dump, uma vez que esse será utilizado para conferir o resultado do processamento.

A opção *screen* será de grande valia quando da depuração do programa e acompanhamento da execução.

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;
- Decisões de implementação;
- Tutorial de como usar o simulador, com instruções completas e precisas de como executá-lo;
- Desenho esquemático da implementação(***datapath***). Veja [1] para exemplos de *datapaths*;
- Estruturas utilizadas para representação do hardware;
- Interpretação do esquema de **monociclo**;
- Descrição dos módulos utilizados;
- Descrição dos testes utilizados na verificação do simulador e a explicação de por que esses testes são completos e garantem o funcionamento correto do simulador;
- Listagem do código fonte do simulador e dos testes realizados. Essa listagem deve ser enviada para o AVA.

Esses são tópicos básicos que são considerados importante na documentação, porém não são suficientes para constituí-la por completo.

Outras informações relevantes do projeto também devem constar da documentação.

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.

Assim, tendo em vista toda a importância dos testes no projeto, eles devem ser apresentados juntos com a documentação. Deve-se, também, explicar os motivos pelos quais aqueles testes foram usados e sua utilidade. 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.

3.2 O que deve ser entregue

No dia da entrega do trabalho devem ser apresentados:

- a documentação do simulador implementado, juntamente com o código fonte do simulador e dos testes realizados. É importante ressaltar que o código e os testes devem ser obrigatoriamente entregues. Trabalhos **iguais** terão a **nota dividida** pelo número de trabalhos iguais;
- O trabalho será apresentado ao professor e a nota de avaliação será individual mediante arguição.

4 Referências

[1] Patterson, David A. and Hennessy, John L.. Computer Organization & Design – The Hardware/Software Interface. 3Th Edition Morgan Kaufmann Publishers, Inc., 2005.

[2] Philip M. Sailer and David R. Kaeli. *The DLX Instruction Set Architecture Handbook*. Morgan Kaufmann Publishers, Inc., 1996.

5 Apêndice A: Formato do arquivo de entrada do montador

5.1 Introdução

Esta seção visa fornecer as informações necessárias para escrever programas em linguagem de montagem para o processador μRISC. Os arquivos executáveis do montador e do linker serão fornecidos para converter o programa em linguagem de montagem para linguagem de máquina. A seguir serão apresentadas as convenções utilizadas pelo montador.

5.2 Formato do arquivo de entrada

O formato do arquivo será explicado através de um exemplo. Suponha o seguinte programa em alto nível:

```
int soma(int reg1, int reg2)
 {
   return reg1 + reg2;
 }
void main()
 {
   static int resultado = 8;
   static int teste;
   int a = 500;
   int b = 10000;
   resultado = soma(a, b);
 }
```

Uma das várias formas de se escrever este mesmo programa em linguagem de montagem é apresentada a seguir:

; Nome deste programa: TESTE1.ASM

```
.module m0
.pseg
.global _soma;  exporting
        ; ---- PROLOGUE for _soma ----
        ; framesize=5, argbuildsize=0
        ; aqui devem ser colocadas instruções para inicialização
        loadlit r4,5    ;create new stack frame
        sub sp,sp,r4
        loadlit r4,0    ;save the old frame pointer
        add r4,r4,sp
        store r4,fp
        or fp,sp,sp     ; and create new one
        loadlit r4,0
        add fp,fp,r4    ; with proper offset.
        loadlit r4,1
        add r4,r4,fp
        store r4,ra     ; save the return address
        loadlit r4,2
        add r4,r4,fp
        store r4,r1     ; save regs
        ; ---- END PROLOGUE ----
        loadlit r4,5
        add r4,fp,r4
        load r0,r4      ; INDIR (reg1) via r4
        loadlit r4,7
        add r4,fp,r4
        load r1,r4      ; INDIR (reg2) via r4
        add r0,r0,r1
L1:
        ; ---- EPILOGUE ----
        loadlit r4,2
```

```
        add r4,r4,fp
        load r1,r4      ; restore regs
        inca r4,fp
        load ra,r4      ; restore return addr
        load fp,fp      ; restore old fp
        loadlit r4,5    ; discard old stack frame
        add sp,sp,r4
        jr ra   ; and return
        ; ---- END EPILOGUE ----
.dseg 1
L3:             ; resultado
.const  8
.dseg 3
L4:             ; teste
.blk    1       ; reserve an 1-word block
.pseg
.global _main;  exporting
        ; ---- PROLOGUE for _main ----
        ; framesize=13, argbuildsize=4
        loadlit r4,13   ;create new stack frame
        sub sp,sp,r4
        loadlit r4,8    ;save the old frame pointer
        add r4,r4,sp
        store r4,fp
        or fp,sp,sp     ; and create new one
        loadlit r4,8
        add fp,fp,r4    ; with proper offset.
        loadlit r4,1
        add r4,r4,fp
        store r4,ra     ; save the return address
        loadlit r4,2
        add r4,r4,fp
        store r4,r1     ; save regs
        ; ---- END PROLOGUE ----
        loadlit r1,500
        loadlit r4,-2
        add r4,r4,fp
        store r4,r1     ;  ASGNP (a) via r4
        lcl r1, LOWBYTE 10000
        lch r1, HIGHBYTE 10000
        loadlit r4,-4
        add r4,r4,fp
        store r4,r1     ;  ASGNP (b) via r4
        loadlit r4,-2
        add r4,fp,r4
        load r0,r4      ; INDIR (a) via r4
        loadlit r4, 0
        add r4,r4,sp
        store r4,r0
        loadlit r4,-4
        add r4,fp,r4
        load r0,r4      ; INDIR (b) via r4
        loadlit r4, 2
        add r4,r4,sp
        store r4,r0
        lcl r0,LOWBYTE _soma
        lch r0,HIGHBYTE _soma
        jal r0  ; _soma
        lcl r1,LOWBYTE L3
        lch r1,HIGHBYTE L3
        store r1,r0
L2:
        ; ---- EPILOGUE ----
        loadlit r4,2
        add r4,r4,fp
        load r1,r4      ; restore regs
        inca r4,fp
        load ra,r4      ; restore return addr
        load fp,fp      ; restore old fp
        loadlit r4,13   ; discard old stack frame
        add sp,sp,r4
HALT:
        j HALT
        ; ---- END EPILOGUE ----
.end
```

Neste programa, podemos observar as seguintes convenções:

- Início do arquivo: indicado pela linha **.module m0**

- Fim do arquivo: indicado pela linha **.end**

- Comentários: cada linha de comentário começa pelo símbolo **;**
  Um exemplo de linha de comentário:
  store r4,ra ; save the return address

- Cada registrador é referenciado através da letra minúscula **r** seguida pelo número do registrador.

  Exemplo: na instrução **add r0, r0, r1** o conteúdo do registrador n.º 1 está sendo somado com o conteúdo do registrador n.º 0 e o resultado armazenado no registrador n.º 0.

  Existem também alguns alias para números de registradores:
  - **ra(return address)** - registrador n.º 7, contém o endereço de retorno quando executada uma instrução de jal(jump-and-link)
  - **sp(stack pointer)** - registrador n.º 6, apontador para o topo da pilha
  - **fp(frame pointer)** - registrador n.º 5

- As instruções devem estar codificadas em mnemônicos, sendo que toda instrução deve estar contida em uma única linha.
  Exemplo de instruções: loadlit r4, 5
  sub sp, sp, 4
  store r4, r2
  Observar que os espaços em branco são ignorados. Ou seja, as instruções
  loadlit r4, 9
  loadlit r4, 9
  são idênticas.

- Qualquer segmento de programa deve ser iniciado pela diretiva **.pseg**, enquanto todo segmento de dados é iniciado através da diretiva **.dseg**.

- Podemos definir rótulos no programa. Um rótulo é um alias para a posição de memória da próxima instrução ou dado do programa seguinte ao rótulo. Exemplos no programa TESTE1.ASM: **L2** , **L3** e **L4** . Algumas características de rótulos:
  - Podemos definir um rótulo em qualquer posição do arquivo.
  - Todo rótulo deve ser iniciado com uma letra. Os caracteres intermediários podem ser alfanuméricos ou o underscore. Para indicar o fim do rótulo usamos dois pontos(**:**). Observar que só usamos os dois pontos quando definimos o rótulo.
  - Para cada rótulo declarado no arquivo existirá um endereço de memória associado, que é o endereço da instrução ou constante seguinte ao rótulo.
  - Toda vez que houver uma referência a um rótulo, ele será substituído pelo seu valor.

  No trecho abaixo do programa, o rótulo **L1** faz referência ao endereço de memória onde está armazenada a instrução **loadlit r4, 2**.

```
L1:   ; Estamos definindo o rótulo
      ; ---- EPILOGUE ----
      loadlit r4,2
      add r4,r4,fp
```

- A diretiva **.const x** pode ser usada para indicar ao montador que na posição atual de memória(indicada pelo contador de posição) deve ser armazenada uma constante de valor **x.** Observar que este valor deve estar entre -32768 e 32767. Na verdade, o montador só considera os 16 bits menos significativos da constante,

Exemplo:

```
.dseg
L3:         ; resultado
.const  8
```

No trecho acima, se o rótulo **L3** se referir ao endereço de memória f14e(hex), a constante 8(0008 em hexadecimal) será armazenada na posição de memória f14e(hex).

- A diretiva LOWBYTE **x** indica para o montador o byte menos significativo da constante ou rótulo **x**(lembre-se que todo rótulo está associado a um valor). HIGHBYTE **x**, de maneira similar, indica o byte mais significativo da constante ou rótulo **x**.

Exemplo: LOWBYTE 10000 se refere ao byte 10(hex) pois ele é o byte menos significativo da constante 10000(2710 em hexadecimal). Já HIGHBYTE 10000 se refere ao byte 27(hex).

Usa-se geralmente estas duas palavras(LOWBYTE e HIGHBYTE) para se carregar uma constante de 16 bits em um registrador através das instruções **lcl** e **lch**. Os dois trechos de código abaixo são um exemplo:

```
lcl r1, LOWBYTE 10000
lch r1, HIGHBYTE 10000
; ....
lcl r1, LOWBYTE L3
lch r1, HIGHBYTE L3
```

Nas duas primeiras instruções, a constante 10000 é carregada no registrador r1, enquanto nas duas últimas instruções o valor de L3(ou seja, a posição de memória que este rótulo referencia) é carregado no registrador r1.

- A diretiva **.blk x** é usada para reservar um bloco de **x** palavras. É usada, por exemplo, quando criamos e não inicializamos uma variável estática.

Veja um exemplo:

```
L4:         ; teste
.blk   1    ; reserve an 1-word block
```

Como a variável **teste** não é inicializada, é necessário reservar um bloco de uma

palavra na memória para a variável.

Outro exemplo:

```
L2:          ; variável1          L2:          ; variável1
.const -1                         .blk 1
```

Os dois trechos de programa acima mostram as duas formas de se alocar um espaço de memória para uma variável estática(de 16 bits). No trecho à esquerda, usamos **.const -1** para inicializar a variável com o valor -1. No trecho à direita, não é feita a inicialização da variável, apenas é reservado o espaço de memória através de **.blk 1**(é reservado um bloco de 1 palavra ou dois bytes para a variável).

- Definimos uma função através da diretiva **.global _nome_da_função**.

Exemplo:
```
.global _soma
; Instruções
```

Por convenção o símbolo **_** deve ser sempre utilizado antes do nome da função. Ao usarmos **.global _nome_da_função** estamos, na verdade, declarando um rótulo **_nome_da_função** para a posição de memória onde se encontra a primeira instrução da função. Assim para chamarmos a função soma declarada no programa TESTE1.ASM, devemos escrever o seguinte código:

```
lcl r0, LOWBYTE _soma      ; carrega o endereço da função soma
lch r0, HIGHBYTE _soma     ; no registrador r0
jal r0   ; chama a função soma e salva o endereço de retorno em ra(ou r7)
```

Convém observar que toda função deve salvar o endereço de retorno e os registradores que serão usados antes de executar qualquer outra operação. Além disso, lembrar que estes valores devem ser restaurados no final da execução da função. Veja exemplos no código do programa TESTE1.ASM.

5.3 Usando o montador e o *linker*

O montador e o *linker* devem ser invocados utilizando os seguintes comandos em UNIX(na lhacorreia no diretório /pkg/cursos/sis/bin):

```
urasm input_file1.asm
urlink input_file1.obj  input_file2.obj ... input_file.obj <-onome_do_arquivo_executável>
```

O primeiro comando executa o montador usando como arquivo de entrada **input_file1.asm** e obtendo como saída um arquivo de nome **input_file1.obj.** No segundo comando, **input_file1.obj** é usado como arquivo de entrada para o linker, que completa a conversão do arquivo para linguagem de máquina, que deverá ser lido pelo simulador. O formato deste arquivo será descrito nas duas próximas páginas.

Obs.: O arquivo de entrada para o montador deve ter uma extensão **.asm.**

Exemplo: suponha que temos um arquivo em linguagem de montagem cujo nome é

teste1.asm. Para convertê-lo para linguagem de máquina utilizaríamos os comandos:

```
urasm teste1.asm
urlink teste1.obj -oteste1
```

Neste caso o nome do arquivo executável seria **teste1**. Caso o nome do arquivo executável não fosse definido, o nome seria **a.out** por convenção.

Um dos arquivos(**.obj**) que serão usados para se construir o arquivo de instruções(utilizando o *linker*) deve ter uma função **main**(declarada através de **.global _main**), que indica onde o programa deve iniciar a execução.

5.4 Bugs

Não podemos usar constantes com instruções de desvio. Por exemplo, não podemos usar a instrução **j -1** . Só podemos usar instruções de desvios com rótulos como **j L**.

6 Apêndice B: Formato do arquivo de instruções

6.1 Introdução

Esta seção define como deve ser o formato do arquivo de instruções para o processador μRISC. Convém salientar que o formato, que será descrito a seguir, deve ser adotado obrigatoriamente por todos os grupos.

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 16 bits cada(os endereços vão de 0 a 65535 em decimal ou de 0x0000 a 0xffff em hexadecimal).

6.2 Descrição do formato

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

- O arquivo, em modo texto, deve conter apenas uma instrução por linha.

- As instruções devem estar codificadas em hexadecimal.

- 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 hexadecimal. Um pequeno exemplo ilustra a utilidade desta palavra-chave. Seja o seguinte arquivo de instruções hipotético:

```
address 00ff
b7ff
c041
address 1000
c868
2fff
```

Neste caso as instruções *0xb7ff* e *0xc041* serão armazenadas nas posições de memória *0x00ff* e *0x0100* respectivamente e as instruções *0xc868* e *0x2fff* nas posições de memória *0x1000* e *0x1001* respectivamente.

Observar que, se a diretiva *address* não for usada para definir onde as instruções devem ser armazenadas, será considerado, por convenção, que as instruções serão colocadas a partir da posição *0x0000* de memória.

6.3 Exemplo de um arquivo de instruções

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

address 0000
b7ff
c041
c400
3000
2fff
address 0041
a01d
980f
501c
3807

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

ENDEREÇO DE MEMÓRIA	CONTEÚDO	ENDEREÇO DE MEMÓRIA	CONTEÚDO
0x0000 (Posição 0)	b7ff	0x0041 (Posição 65)	a01d
0x0001 (Posição 1)	c041	0x0042 (Posição 66)	980f
0x0002 (Posição 2)	c400	0x0043 (Posição 67)	501c
0x0003 (Posição 3)	3000	0x0044 (Posição 68)	3807
0x0004 (Posição 4)	2fff		

7 Apêndice C: Exemplo de teste para o μRISC

7.1 Instruções para a execução do teste

Nesta seção é apresentado um teste para o μRISC. O teste é o arquivo **teste1.asm**(arquivo no formato de entrada do montador) que é listado a seguir.

O arquivo de entrada para o montador(*urasm*) é o **teste1.asm**. Para obter o arquivo no formato de entrada do simulador execute os seguintes comandos na máquina **lhacorreia**:

```
 /pkg/cursos/sis/bin/urasm teste1.asm
/pkg/cursos/sis/bin/urlink teste1.obj -oteste1
```

O arquivo no formato de entrada do simulador é o arquivo **teste1**. O arquivo **teste1.lis** gerado conterá a listagem da montagem do código, sendo grande utilidade na verificação do teste.

Se seu simulador do μRISC funcionar corretamente, no final da execução do **teste1**, a posição de memória 1000(0x3e8 em hexadecimal) conterá o valor 10. Se isso ocorrer, ótimo! Seu simulador **PARECE** funcionar corretamente para uma parcela das instruções. Se seu simulador passar nesse teste, você **não demonstrou** que ele está totalmente correto. Faça mais testes até poder garantir isso.

Se seu simulador entrar em loop infinito, isso não é um bom sinal. Se no final da execução do **teste1**, o seu simulador conter um valor menor do que 10 na posição 1000 de memória, algumas instruções não estão funcionando corretamente. Siga a execução do teste(veja o arquivo **teste1.lis**) para descobrir o que está dando errado.

Vale lembrar novamente que os testes para a avaliação do simulador do μRISC serão exaustivos e abrangentes, e possivelmente detectarão qualquer erro no seu simulador. Garanta que ele funcione em qualquer caso.

7.2 Listagem do arquivo teste1

```
.module m0
.pseg
.global _main
; TESTE uRISC - Flags, Jumps condicionais e
;               incondicionais, instrucoes de
;               carregamento de constantes, operacoes
;               aritmeticas
; RESULTADO DO TESTE: nota de 0 a 10, que no final
;                     da execucao do programa, estara
;                     na posicao 1000 de memoria(3e8 em hexa)

; Instrucoes que saltam para as rotinas de tratamento de
; de interrupcoes
j RESET
j INT1
j INT2
j INT3

; Inicio do programa
RESET:
; Carrega registradores
loadlit r4, 0   ; r4 contera o score
lcl r0, LOWBYTE 32768  ; carrega 8000 em r0
lch r0, HIGHBYTE 32768 ;
```

```
lcl r1, LOWBYTE 56750  ; carrega ddae em r1
lch r1, HIGHBYTE 56750 ;
lcl r2, LOWBYTE 32746  ; carrega 7fea em r2
lch r2, HIGHBYTE 32746 ;
lcl r3, LOWBYTE 11207  ; carrega 2bc7 em r3
lch r3, HIGHBYTE 11207 ;

; ---------------------------------------
; Primeira bateria de teste:
; testa jump condicional quando a condicao
; e falsa

; Teste 1: ocorre overflow
add r6, r0, r0
jf.overflow T2    ; erro se saltar
inca r4, r4      ; OK ! Mais um ponto

T2:
; Teste 2: nao ocorre overflow
add r6, r3, r3
jt.overflow T3   ; erro se saltar
inca r4, r4      ; OK ! Mais um ponto

T3:
; Teste 3: resultado e negativo
sub r6, r3, r2
jf.neg T4         ; erro se saltar
inca r4, r4      ; OK ! Mais um ponto

T4:
; Teste 4: resultado nao e negativo
sub r6, r2, r3
jt.neg T5         ; erro se saltar
inca r4, r4      ; OK ! Mais um ponto

T5:
; Teste 5: resultado e zero
sub r6, r2, r2
jf.zero T6       ; erro se saltar
inca r4, r4      ; OK ! Mais um ponto

T6:
; Teste 6: resultado nao e zero
sub r6, r3, r2
jt.zero T7       ; erro se saltar
inca r4, r4      ; OK ! Mais um ponto

T7:
; Teste 7: resultado e negativo ou zero
sub r6, r3, r3
jf.negzero T8    ; erro se saltar
inca r4, r4      ; OK ! Mais um ponto

T8:
; Teste 8: resultado nao e negativo e nem igual a zero
sub r6, r2, r3
jt.negzero T9    ; erro se saltar
inca r4, r4      ; OK ! Mais um ponto

T9:
; Teste 9: ocorre carry
add r6, r0, r0
jf.carry T10     ; erro se saltar
inca r4, r4      ; OK ! Mais um ponto

T10:
; Teste 10: nao ocorre carry
add r6, r2, r3
jt.carry T11     ; erro se saltar
inca r4, r4      ; OK ! Mais um ponto

; ---------------------------------------
; Segunda bateria de teste:
; testa jump condicional quando a condicao
; e verdadeira

T11:
```

```
; Teste 1: ocorre overflow
sub r6, r0, r2
jt.overflow T11a ; OK se saltar
j T12
T11a:
inca r4, r4      ; OK ! Mais um ponto

T12:
; Teste 2: nao ocorre overflow
add r6, r3, r3
jf.overflow T12a ; OK se saltar
j T13
T12a:
inca r4, r4      ; OK ! Mais um ponto

T13:
; Teste 3: resultado e negativo
sub r6, r3, r2
jt.neg T13a      ; OK se saltar
j T14
T13a:
inca r4, r4      ; OK ! Mais um ponto

T14:
; Teste 4: resultado nao e negativo
sub r6, r2, r3
jf.neg T14a      ; OK se saltar
j T15
T14a:
inca r4, r4      ; OK ! Mais um ponto

T15:
; Teste 5: resultado e zero
sub r6, r6, r6
jt.zero T15a     ; OK se saltar
j T16
T15a:
inca r4, r4      ; OK ! Mais um ponto

T16:
; Teste 6: resultado nao e zero
subdec r6, r3, r2
jf.zero T16a     ; OK se saltar
j T17
T16a:
inca r4, r4      ; OK ! Mais um ponto

T17:
; Teste 7: resultado e negativo ou zero
sub r6, r1, r1
jt.negzero T17a  ; OK se saltar
j T18
T17a:
inca r4, r4      ; OK ! Mais um ponto

T18:
; Teste 8: resultado nao e negativo e nem igual a zero
sub r6, r2, r3
jf.negzero T18a  ; OK se saltar
j T19
T18a:
inca r4, r4      ; OK ! Mais um ponto

T19:
; Teste 9: ocorre carry
add r6, r0, r1
jt.carry T19a    ; OK se saltar
j T20
T19a:
inca r4, r4      ; OK ! Mais um ponto

T20:
; Teste 10: nao ocorre carry
add r6, r2, r3
jf.carry T20a    ; OK se saltar
j FIM
T20a:
```

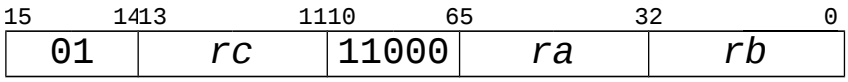
```
inca r4, r4      ; OK ! Mais um ponto

FIM:
asr r4, r4
loadlit r5, 1000 ; resultado do teste na posicao 0x3e8 de memoria
store r5, r4
L: j L
INT1: jr r7
INT2: jr r7
INT3: jr r7
.end
```

8 Apêndice D: Descrição detalhada das instruções

Seguem abaixo as descrições das instruções implementadas pelo processador μRISC.

8.1 SOMA INTEIRA **add rc,ra,rb**



Tipo: I

Formato: add rc, ra, rb

Exemplo: add r3, r2, r1

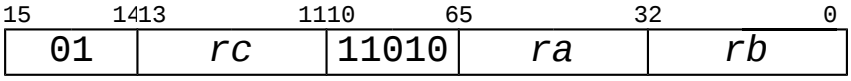
Operação: $rc = ra + rb$

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

Flags afetados: neg, zero, carry e overflow.

8.2 SOMA INTEIRA COM INCREMENTO

addinc rc,ra,rb



Tipo: I

Formato: addinc rc, ra, rb

Exemplo: addinc r3, r2, r1

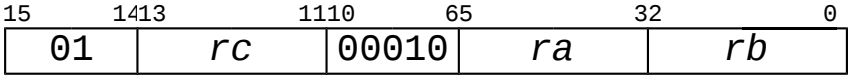
Operação: $rc = ra + rb + 1$

Descrição: Soma (aritmética) o conteúdo de *ra* e *rb*, adiciona 1 e coloca o resultado em *rc*.

Flags afetados: neg, zero, carry, overflow.

8.3 AND LÓGICO

and rc,ra,rb



Tipo: I

Formato: and rc, ra, rb

Exemplo: and r3, r2, r7

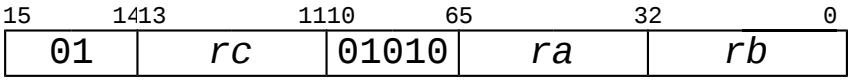
Operação: $rc = ra \& rb$

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

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

8.4 NOT *ra* e AND LÓGICO

andnota rc,ra,rb



Tipo: I

Formato: andnota rc, ra, rb

Exemplo: andnota r3, r2, r7

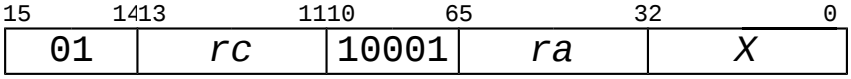
Operação: $rc = !ra \& rb$

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

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

8.5 SHIFT ARITMÉTICO PARA A ESQUERDA

asl rc,ra



Tipo: I

Formato: asl rc, ra

Exemplo: asl r3, r2

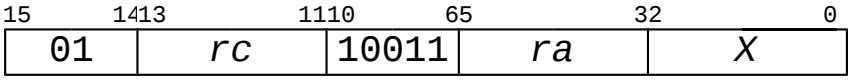
Operação: $rc = ra << 1$

Descrição: Coloca cada bit ra_i em rc_{i+1} e preenche com 0 a posição rc_0.

Flags afetados: neg, zero. carry = ra_{15} e overflow = ra_{15} ^ ra_{14}.

8.6 SHIFT ARITMÉTICO PARA A DIREITA

asr rc,ra



Tipo: I

Formato: asr rc, ra

Exemplo: asr r3, r2

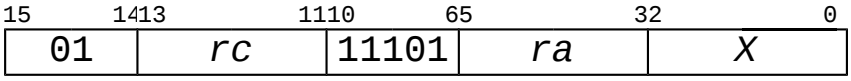
Operação: $rc = ra >> 1$

Descrição: Coloca cada bit ra_i em rc_{i-1} (exceto ra_0) e preenche com ra_{15} a posição rc_{15}.

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

8.7 DECREMENTO

deca rc,ra



Tipo: I

Formato: deca rc, ra

Exemplo: deca r3, r1

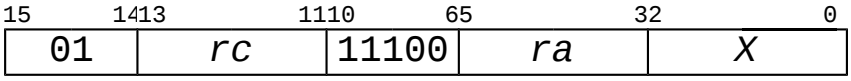
Operação: $rc = ra - 1$

Descrição: Subtrai 1 do conteúdo de *ra* e coloca o resultado em *rc*.

Flags afetados: neg, zero, carry, overflow.

8.8 INCREMENTO

inca rc,ra



Tipo: I

Formato: inca rc, ra

Exemplo: inca r3, r1

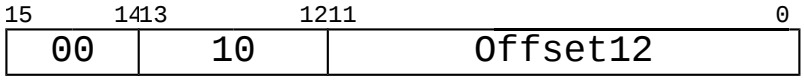
Operação: $rc = ra + 1$

Descrição: Adiciona 1 ao conteúdo de *ra* e coloca o resultado em *rc*.

Flags afetados: neg, zero, carry, overflow.

8.9 JUMP INCONDICIONAL

j Destino



Tipo: V

Formato: j Destino

Exemplo: j loop

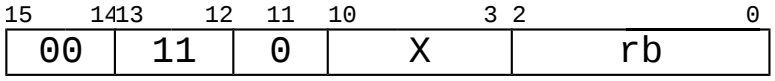
Operação: faça PC = (PC + 1) + Offset12EstendidoPara16Bits

Descrição: Realiza desvio de fluxo incondicional relativo ao PC. O endereço da próxima instrução a ser executada será o novo PC(PC + 1) mais a constante de 12 bits(presente nos bits de 0 a 11 da instrução) estendida de sinal.

Flags afetados: Nenhum.

8.10 JUMP AND LINK

jal rb



Tipo: VI

Formato: jal rb

Exemplo: jal r4

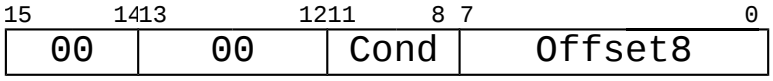
Operação: faça r7 = PC e PC = rb

Descrição: Realiza chamadas a procedimentos guardando o endereço do PC atual no registrador *r7* (para o retorno após o procedimento) e colocando em PC o valor de *rb* (primeira instrução do procedimento).

Flags afetados: Nenhum.

8.11 JUMP FALSE

jf.cond Destino



Tipo: IV

Formato: jf.cond offset8

Exemplo: jf.zero 10

Operação: se condição é falsa, faça PC = (PC + 1) + Offset8EstendidoPara16Bits

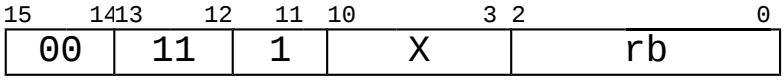
Descrição: Realiza desvio condicional relativo ao PC se a condição é falsa. O endereço da próxima instrução a ser executada será o novo PC(PC + 1) mais a constante de 8 bits(presente nos bits de 0 a 7 da instrução) estendida de sinal.

Flags afetados: Nenhum.

$COND_{bin}$	**Condição**	**Mnemônico**
0100	Resultado ALU negativo	.neg
0101	Resultado ALU zero	.zero
0110	Carry da ALU	.carry
0111	Resultado ALU negativo ou zero	.negzero
0000	TRUE	.true
0011	Resultado ALU *overflow*	.overflow

8.12 JUMP REGISTER

jr rb



Tipo: VI

Formato: jr rb

Exemplo: jr r7

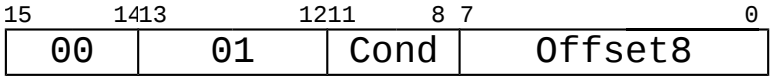
Operação: faça PC = r7.

Descrição: Armazena o conteúdo do registrador *b* em PC.

Flags afetados: Nenhum.

8.13 JUMP TRUE

jt.cond Destino



Tipo: IV

Formato: jt.cond Destino

Exemplo: jt.true loop

Operação: se condição é verdadeira, faça PC = (PC + 1) + Offset8EstendidoPara16Bits

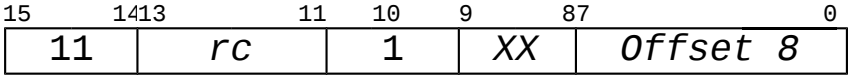
Descrição: Realiza desvio condicional relativo ao PC se a condição é verdadeira. O endereço da próxima instrução a ser executada será o novo PC(PC + 1) mais a constante de 8 bits(presente nos bits de 0 a 7 da instrução) estendida de sinal.

Flags afetados: Nenhum.

$COND_{bin}$	**Condição**	**Mnemônico**
0100	Resultado ALU negativo	.neg
0101	Resultado ALU zero	.zero
0110	Carry da ALU	.carry
0111	Resultado ALU negativo ou zero	.negzero
0000	TRUE	.true
0011	Resultado ALU *overflow*	.overflow

8.14 CARREGA CONSTANTE NO BYTE MAIS SIGNIFICATIVO

lch c, Const8



Tipo: III

Formato: lcl rc, Const 8

Exemplo: lcl r3, Const 8

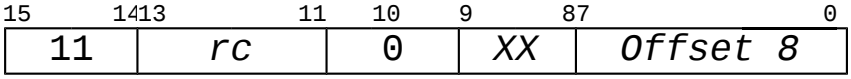
Operação: $rc = (offset8 << 8) \mid (rc \,\&\, 0x00ff)$

Descrição: Carrega no byte mais significativo de *rc* o conteúdo de *Offset8*.

Flags afetados: Nenhum.

8.15 CARREGA CONSTANTE NO BYTE MENOS SIGNIFICATIVO

lcl c, Const8



Tipo: **III**

Formato: lcl rc, Const 8

Exemplo: lcl r3, Const 8

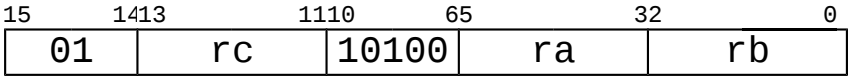
Operação: rc = Offset8 | (rc & 0xFF00)

Descrição: Carrega no byte menos significativo de *rc* o conteúdo de Offset 8.

Flags afetados: Nenhum.

8.16 LOAD WORD

load rc, ra



Tipo: VII

Formato: load rc, ra

Exemplo: load r3, r6

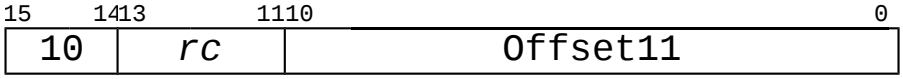
Operação: *rc* = memória [*ra*]

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

Flags afetados: nenhum.

Observação: Na instrução de **load**, o registrador **RB** não tem efeito.

8.17 CARREGA CONSTANTE DE 11 BITS COM SINAL — loadlit c, Const



Tipo: II

Formato: loadlit rc, Const

Exemplo: loadlit r3, 1000

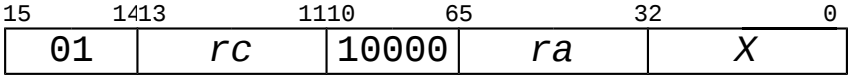
Operação: rc = Offset11EstendidoPara16Bits

Descrição: Carrega em *rc* o conteúdo de *Offset 11* estendido para 16 bits.

Flags afetados: Nenhum.

8.18 SHIFT LÓGICO PARA A ESQUERDA

lsl rc,ra



Tipo: I

Formato: lsl rc, ra

Exemplo: lsl r3, r2

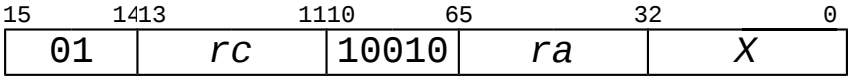
Operação: $rc = ra << 1$

Descrição: Coloca cada bit ra_i em rc_{i+1} (exceto ra_{15}) e preenche com 0 a posição rc_0.

Flags afetados: neg, zero. carry = ra_{15}. overflow = 0.

8.19 SHIFT LÓGICO PARA A DIREITA

lsr rc,ra



Tipo: I

Formato: lsr rc, ra

Exemplo: lsr r3, r2

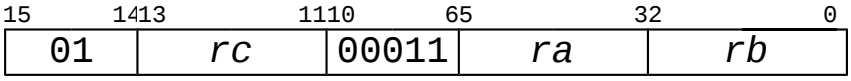
Operação: $rc = ra >> 1$

Descrição: Coloca cada bit ra_i em rc_{i-1} (exceto ra_0) e preenche com 0 a posição rc_{15}.

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

8.20 NAND LÓGICO

nand rc,ra,rb



Tipo: I

Formato: nand rc, ra, rb

Exemplo: nand r3, r2, r7

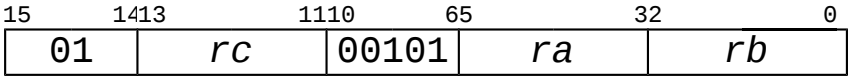
Operação: *rc = !ra | !rb*

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

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

8.21 NOR LÓGICO

nor rc,ra,rb



Tipo: I

Formato: nor rc, ra, rb

Exemplo: nor r3, r2, r7

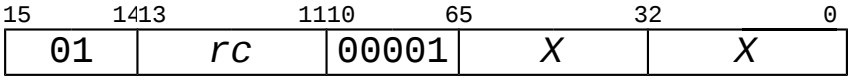
Operação: *rc = !ra & !rb*

Descrição: Efetua *nor* lógico bit a bit de *ra* e *rb* e coloca resultado em *rc*.

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

8.22 ONES

ones rc



Tipo: I

Formato: ones rc

Exemplo: ones r1

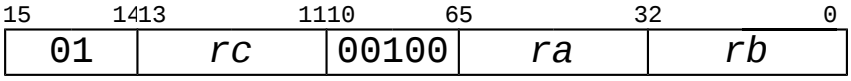
Operação: $rc = 1$

Descrição: Faz conteúdo de *rc* valer 1.

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

8.23 OR LÓGICO

or rc,ra,rb



Tipo: I

Formato: or rc, ra, rb

Exemplo: or r3, r2, r7

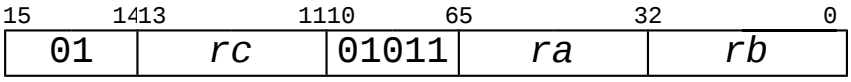
Operação: $rc = ra | rb$

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

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

8.24 NOT rb e OR LÓGICO

ornotb rc,rb,rb



Tipo: I

Formato: ornotb rc,ra,rb

Exemplo: ornotb r3,r2,r7

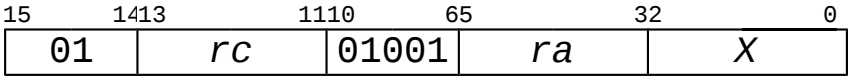
Operação: $rc = ra \,|\, !rb$

Descrição: Efetua *or* lógico bit a bit de *ra* e *not rb* e coloca o resultado em *rc*.

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

8.25 COPIA *ra*

passa rc,ra



Tipo: I

Formato: passa rc, ra

Exemplo: passa r4, r3

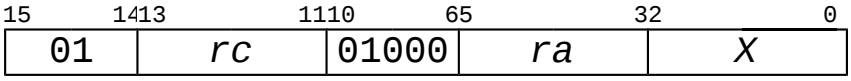
Operação: *rc = ra*

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

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

8.26 NOT *ra*

passnota rc,ra



Tipo: I

Formato: passnota rc, ra

Exemplo: passnota r4, r3

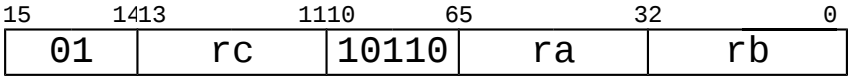
Operação: $rc = !ra$

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

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

8.27 STORE WORD

store ra, rb



Tipo: VII

Formato: store ra, rb

Exemplo: store r3, r6

Operação: memória [*ra*] = rb

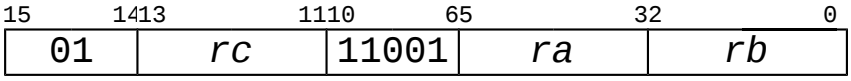
Descrição: carrega na posição da memória endereçada pelo registrador *a* o conteúdo do registrador *b*.

Flags afetados: nenhum.

Observação: Para a instrução de **store**, o registrador **RC** não tem efeito.

8.28 SUBTRAÇÃO INTEIRA

sub rc,ra,rb



Tipo: I

Formato: sub rc, ra, rb

Exemplo: sub r3, r2, r1

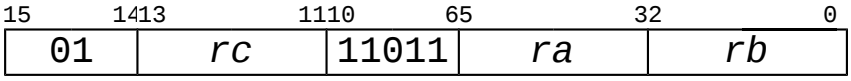
Operação: $rc = ra - rb$

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

Flags afetados: neg, zero, carry, overflow.

8.29 SUBTRAÇÃO INTEIRA COM DECREMENTO

subdec rc,ra,rb



Tipo: I

Formato: subdec rc, ra, rb

Exemplo: subdec r3, r2, r1

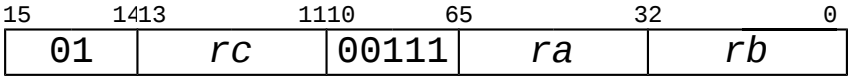
Operação: $rc = ra - rb - 1$

Descrição: Subtrai o conteúdo de *rb* do conteúdo de *ra*, subtrai 1 da diferença e coloca o resultado em *rc*.

Flags afetados: neg, zero, carry e overflow.

8.30 XNOR LÓGICO

xnor rc,ra,rb



Tipo: I

Formato: xnor rc, ra, rb

Exemplo: xnor r3, r2, r7

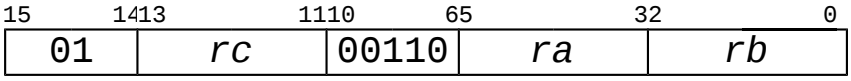
Operação: $rc = !(ra \wedge rb)$

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

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

8.31 XOR LÓGICO

xor rc,ra,rb



Tipo: I

Formato: xor rc, ra, rb

Exemplo: xor r3, r2, r7

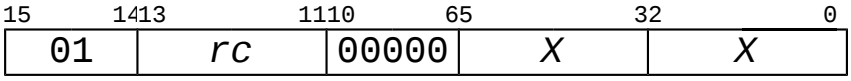
Operação: $rc = ra \wedge rb$

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

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

8.32 ZERA

zeros rc



Tipo: I

Formato: zeros rc

Exemplo: zeros r1

Operação: $rc = 0$

Descrição: Faz conteúdo de *rc* valer 0.

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