

# **Viking CPU - Manual de referência v0.3**

Sérgio Johann Filho

24 de janeiro de 2018

# Sumário

<b>1</b>	<b>A arquitetura Viking . . . . .</b>	<b>2</b>
1.1	Registradores . . . . .	3
1.2	Formatos de instrução . . . . .	3
1.2.1	Instruções tipo R . . . . .	3
1.2.2	Instruções tipo I . . . . .	4
1.3	Modos de endereçamento . . . . .	5
1.4	Conjunto de instruções . . . . .	5
1.4.1	Computação . . . . .	5
1.4.2	Deslocamento . . . . .	9
1.4.3	Carga e armazenamento . . . . .	9
1.4.4	Desvios condicionais . . . . .	11
1.5	Características únicas . . . . .	12
1.5.1	Carga de constantes . . . . .	12
1.5.2	Extensão de sinal . . . . .	13
1.5.3	Desvios condicionais . . . . .	13
1.5.4	Outras operações . . . . .	13
1.6	Tipos de dados . . . . .	14
1.7	Convenções de chamada de função . . . . .	14
1.7.1	Pilha . . . . .	14
1.7.2	Registradores . . . . .	15
1.7.3	Chamada e retorno de funções . . . . .	15
<b>2</b>	<b>Síntese de pseudo instruções . . . . .</b>	<b>17</b>
2.1	Pseudo operações básicas . . . . .	17
2.2	Operações de deslocamento . . . . .	18
2.3	Pseudo operações não suportadas pelo montador . . . . .	19
2.3.1	Testes, seleção e desvios (condicionais) . . . . .	19
2.3.2	Operações condicionais equivalentes . . . . .	19
2.3.3	Desvios incondicionais . . . . .	20
2.3.4	Operações aritméticas adicionais . . . . .	20
<b>3</b>	<b>Montagem de código e simulação . . . . .</b>	<b>21</b>
3.1	Montador . . . . .	21
3.1.1	Formato da linguagem de montagem . . . . .	21
3.1.2	Sintaxe de linha de comando . . . . .	22
3.2	Simulador . . . . .	23
3.2.1	Mapa de memória . . . . .	24
3.2.2	Sintaxe de linha de comando . . . . .	24
<b>A</b>	<b>Exemplos . . . . .</b>	<b>25</b>
<b>B</b>	<b>Rotinas <i>mulsi3</i>, <i>divsi3</i>, <i>modsi3</i> e <i>udivmodsi4</i> . . . . .</b>	<b>28</b>

# Capítulo 1

## A arquitetura Viking

Viking é uma arquitetura simples, construída de acordo com a filosofia RISC. Essa arquitetura foi planejada com o objetivo de servir como ponto de partida para um conjunto de instruções básico extensível, em que uma quantidade reduzida de *hardware* é necessário para implementar seu conjunto de operações<sup>1</sup>, e ainda possuir funcionalidade suficiente para a execução de *software* de alto nível. Por exemplo, o banco de registradores possui poucas entradas, poucos multiplexadores são necessários, não existem qualificadores de estado de operações, unidades multiplicação e divisão não foram definidas, tampouco uma unidade de deslocamento (*barrel shifter*). Esse processador pode ser implementado em variantes de 16 e 32 bits, sendo que a diferença entre as duas se dá apenas com relação ao tamanho dos registradores do seu banco, o que não altera o conjunto de instruções básico.

Um pequeno número de operações é definido no conjunto de instruções da arquitetura Viking (17 operações básicas). Apesar do pequeno número de instruções, estas são poderosas o suficiente para realizarem todas as operações de máquinas com um maior número de instruções. Para que isso seja possível, muitas vezes uma instrução é utilizada de modo pouco ortodoxo ou uma combinação de instruções implementam um único comportamento. As operações são separadas em quatro classes distintas:

1. *Computação* (AND, OR, XOR, SLT, SLTU, ADD, SUB, LDR, LDC)
2. *Deslocamento* (LSR, ASR)
3. *Carga e armazenamento* (LDB, STB, LDW, STW)
4. *Desvios condicionais* (BEZ, BNZ)

Seguindo a filosofia RISC, as instruções são definidas em uma codificação que utiliza apenas dois formatos de instrução, com um tamanho fixo de 16 bits por instrução. Assim, a lógica necessária para a decodificação de instruções é reduzida significativamente, comparado ao que seria necessário para decodificar instruções com um tamanho variável. Além disso, um tamanho de 16 bits permite uma boa densidade de código, quando comparado a outros ISAs que possuem instruções de tamanho fixo porém com 32 bits.

---

<sup>1</sup>O conjunto de instruções foi definido com o intuito de minimizar a complexidade da arquitetura e de forma que possa facilmente sintetizar operações mais complexas através de poucas operações básicas.

## 1.1 Registradores

Assim como outros processadores RISC, o processador Viking é definido como uma arquitetura baseada em operações de carga e armazenamento (*load/store*) para acesso à memória de dados. Para que operações lógicas e aritméticas possam ser executadas, é necessário que os operandos sejam trazidos da memória ou carregados como constantes em um ou mais registradores de propósito geral (GPRs).

São definidos 8 registradores (*r0* - *r7*) e estes podem ser utilizados para qualquer finalidade, sendo apenas recomendado seu uso em função das convenções apresentadas na Seção 1.7. Além dos 8 registradores de propósito geral (GPRs), é definido um registrador chamado contador de programa (PC). Esse registrador aponta para a instrução corrente do programa, e não pode ser modificado diretamente pelo programador. A cada instrução que é decodificada, o PC avança para a próxima posição. Desvios condicionais podem fazer com que o PC seja atualizado com o destino do desvio, caso tomado. As instruções possuem um tamanho de 16 bits, portanto o contador de programa é incrementado com esse tamanho.

## 1.2 Formatos de instrução

Existem apenas dois formatos de instrução definidos na arquitetura Viking (tipos R e I). Em instruções do tipo R, um registrador é definido como destino (*Rst*) e dois registradores são definidos como fontes (*RsA* e *RsB*). Em instruções do tipo I, um registrador é definido como fonte e destino da operação (*Rst*), e o segundo valor usado como fonte é obtido a partir do campo *Immediate* codificado diretamente na instrução. Os índices utilizados para indexar o banco de registradores são codificados na instrução em 3 bits cada, o suficiente para referenciar 8 registradores por operando ou destino para escrita do resultado.

### 1.2.1 Instruções tipo R

Em instruções do tipo R os campos *Opcode* (4 bits) e *Op2* (2 bits) definem a operação específica. Nesse tipo de instrução três registradores são referenciados, e o papel desses registradores depende da classe à qual a instrução está associada. As instruções do tipo R possuem o campo *Imm* com o valor fixo em 0.

$I_{<15:12>}$	$I_{<11>}$	$I_{<10:8>}$	$I_{<7:5>}$	$I_{<4:2>}$	$I_{<1:0>}$
<i>Opcode</i>	<i>Imm</i>	<i>Rst</i>	<i>RsA</i>	<i>RsB</i>	<i>Op2</i>
x x x x	0	r r r	r r r	r r r	x x

A função dos campos adicionais em instruções do tipo R é definida como:

- *Rst* - registrador destino (alvo) da operação;
- *RsA* - registrador Fonte 1 (Operando A);
- *RsB* - registrador Fonte 2 (Operando B ou base);
  - Operando B em operações da classe computação
  - Endereço base para instruções de carga e armazenamento e desvios;

Para instruções de deslocamento, o registrador Fonte 2 deve ser sempre  $r0$ . O motivo para isso é que não é necessário codificar a quantidade a ser deslocada, uma vez que a arquitetura pode deslocar apenas 1 bit por instrução. Em instruções de carga, o registrador Fonte 1 deve ser sempre  $r0$  e em instruções de armazenamento e desvios condicionais, o registrador alvo é sempre  $r0$ . Abaixo são apresentados alguns exemplos de instruções do tipo R, utilizando a sintaxe da linguagem de montagem apresentada no Capítulo 3. Importante observar que em instruções de armazenamento e desvios condicionais  $Rst$  deve ser  $r0$ , em instruções de carga  $RsA$  deve ser  $r0$  e em deslocamentos  $RsB$  deve ser  $r0$ <sup>2</sup>.

Operação	Significado
<code>add r3,r1,r2</code>	$r3 = r1 + r2$
<code>ldb r3,r0,r2</code>	$r3 = \text{MEM}[r2]$
<code>stw r0,r1,r2</code>	$\text{MEM}[r2] = r1$
<code>and r2,r3,r4</code>	$r2 = r3 \text{ and } r4$
<code>bez r0,r2,r3</code>	if ( $r2 == \text{zero}$ ) $\text{PC} = r3$
<code>slt r3,r1,r2</code>	if ( $r1 < r2$ ) $r3 = 1$ , else $r3 = 0$
<code>lsr r5,r3,r0</code>	$r5 = r3 >> 1$

### 1.2.2 Instruções tipo I

Em instruções do tipo I o campo *Opcode* (4 bits) define a operação específica. Nesse tipo de instrução um registrador é referenciado, e o papel desse registrador depende da classe à qual a instrução está associada. As instruções do tipo I possuem o campo *Imm* com o valor fixo em 1.

$I_{<15:12>}$	$I_{<11>}$	$I_{<10:8>}$	$I_{<7:0>}$
<i>Opcode</i>	<i>Imm</i>	<i>Rst</i>	<i>Immediate</i>
x x x x	1	r r r	i i i i i i i i

A função dos campos adicionais em instruções do tipo I é definida como:

- *Rst* - registrador Fonte 1 e destino;
- *Immediate* - campo com valor imediato;
  - Fonte 2 em instruções da classe computação;
  - Endereço relativo ao contador de programa em desvios;

Para desvios condicionais, endereço efetivo é calculado somando-se o valor atual do contador de programa (PC) ao campo *Immediate* (extendido em sinal<sup>3</sup> e representado em complemento de 2). Dessa forma, é possível realizar desvios relativos ao PC de  $\pm 128$  bytes<sup>4</sup>, o suficiente para lidar com a maior parte dos casos que envolvem saltos de tamanho reduzido, como em comandos de seleção e laços curtos. Abaixo são apresentados alguns exemplos de instruções do tipo I, utilizando a sintaxe da linguagem de montagem.

<sup>2</sup>O motivo para tais convenções é fixar no formato de instruções o papel dos registradores *Rst*, *RsA* e *RsB*, evitando a utilização de multiplexadores adicionais. No tipo R, o primeiro registrador sempre é escrito, e os dois últimos sempre lidos. No tipo I, o primeiro é sempre lido e escrito.

<sup>3</sup>A implementação de extensão de sinal é apresentada na Seção 1.5.2.

<sup>4</sup>No futuro o campo *Immediate* poderá codificar apenas a magnitude alinhada, o que aumenta o alcance dos desvios relativos para  $\pm 256$  bytes.

Operação	Significado
add r5,10	$r5 = r5 + 10$
or r2,1	$r2 = r2 \text{ or } 1$
xor r5,-1	$r5 = r5 \text{ xor } -1 = \text{not } r5$
ldr r3,5	$r3 = 5$
ldc r3,10	$r3 = (r3 \ll 8) \text{ or } 10$
slt r4,10	if ( $r4 < 10$ ) $r4 = 1$ , else $r4 = 0$
bez r4,28	if ( $r4 == \text{zero}$ ) $PC = PC + 28$

## 1.3 Modos de endereçamento

Apenas três modos de endereçamento são utilizados na arquitetura, sendo esses:

1. *Registrador*
2. *Imediato*
3. *Relativo ao PC*

O primeiro modo (registrador) é utilizado por instruções do tipo R apenas. Instruções que fazem uso desse modo pertencem às classes computação, deslocamento, carga e armazenamento e desvios condicionais. O segundo modo (imediato) é utilizado por instruções do tipo I apenas, classe computação. O último modo (relativo ao PC) é utilizado por instruções do tipo I, classe desvios condicionais.

Dois modos de endereçamento bastante comuns são os modos *direto* e *indireto*. A arquitetura Viking não define esses modos de endereçamento, uma vez que a memória de dados é acessada exclusivamente por operações de carga e armazenamento. No entanto, tais modos podem ser emulados<sup>5</sup> com o uso de múltiplas instruções de carga, permitindo acesso à memória pelo número indireções desejado. Outros modos de endereçamento como *base + deslocamento*, *base + índice*, *indireto à registrador*, *indireto à memória* e *auto incremento*, entre outros, não foram definidos com o objetivo de simplificar a arquitetura.

## 1.4 Conjunto de instruções

O conjunto de instruções básico definido na arquitetura é apresentado a seguir. Diversos códigos de operação são reservados para extensões futuras, como operações aritméticas, carga e armazenamento e desvios, além de instruções mais poderosas com tamanho de 32 bits.

As operações definidas no conjunto de instruções básico permitem que operações não elementares possam ser geradas a partir de sequências curtas. Como as instruções possuem tamanho de 16 bits, a densidade do código é boa.

### 1.4.1 Computação

#### AND - bitwise logical product

Realiza o produto lógico de dois valores e armazena o resultado em um registrador.

- AND Rst, RsA, RsB

$$\text{GPR}[\text{Rst}] \leftarrow \text{GPR}[\text{RsA}] \text{ and } \text{GPR}[\text{RsB}]$$

<sup>5</sup>No Capítulo 2 são apresentadas pseudo operações que emulam o modo de endereçamento direto.

$I_{\langle 15:12 \rangle}$	$I_{\langle 11 \rangle}$	$I_{\langle 10:8 \rangle}$	$I_{\langle 7:5 \rangle}$	$I_{\langle 4:2 \rangle}$	$I_{\langle 1:0 \rangle}$
<i>Opcode</i>	<i>Imm</i>	<i>Rst</i>	<i>RsA</i>	<i>RsB</i>	<i>Op2</i>
0 0 0 0	0	r r r	r r r	r r r	0 0

- AND Rst, Immediate

$\text{GPR}[\text{Rst}] \leftarrow \text{GPR}[\text{Rst}] \text{ and } \text{ZEXT}(\text{Immediate})$

$I_{\langle 15:12 \rangle}$	$I_{\langle 11 \rangle}$	$I_{\langle 10:8 \rangle}$	$I_{\langle 7:0 \rangle}$
<i>Opcode</i>	<i>Imm</i>	<i>Rst</i>	<i>Immediate</i>
0 0 0 0	1	r r r	i i i i i i i i

## OR - bitwise logical sum

Realiza a soma lógica de dois valores e armazena o resultado em um registrador.

- OR Rst, RsA, RsB

$\text{GPR}[\text{Rst}] \leftarrow \text{GPR}[\text{RsA}] \text{ or } \text{GPR}[\text{RsB}]$

$I_{\langle 15:12 \rangle}$	$I_{\langle 11 \rangle}$	$I_{\langle 10:8 \rangle}$	$I_{\langle 7:5 \rangle}$	$I_{\langle 4:2 \rangle}$	$I_{\langle 1:0 \rangle}$
<i>Opcode</i>	<i>Imm</i>	<i>Rst</i>	<i>RsA</i>	<i>RsB</i>	<i>Op2</i>
0 0 0 1	0	r r r	r r r	r r r	0 0

- OR Rst, Immediate

$\text{GPR}[\text{Rst}] \leftarrow \text{GPR}[\text{Rst}] \text{ or } \text{ZEXT}(\text{Immediate})$

$I_{\langle 15:12 \rangle}$	$I_{\langle 11 \rangle}$	$I_{\langle 10:8 \rangle}$	$I_{\langle 7:0 \rangle}$
<i>Opcode</i>	<i>Imm</i>	<i>Rst</i>	<i>Immediate</i>
0 0 0 1	1	r r r	i i i i i i i i

## XOR - bitwise logical difference

Realiza a diferença lógica de dois valores e armazena o resultado em um registrador. No tipo I, o segundo valor possui extensão de sinal.

- XOR Rst, RsA, RsB

$\text{GPR}[\text{Rst}] \leftarrow \text{GPR}[\text{RsA}] \text{ xor } \text{GPR}[\text{RsB}]$

$I_{\langle 15:12 \rangle}$	$I_{\langle 11 \rangle}$	$I_{\langle 10:8 \rangle}$	$I_{\langle 7:5 \rangle}$	$I_{\langle 4:2 \rangle}$	$I_{\langle 1:0 \rangle}$
<i>Opcode</i>	<i>Imm</i>	<i>Rst</i>	<i>RsA</i>	<i>RsB</i>	<i>Op2</i>
0 0 1 0	0	r r r	r r r	r r r	0 0

- XOR Rst, Immediate

$\text{GPR}[\text{Rst}] \leftarrow \text{GPR}[\text{Rst}] \text{ xor } \text{SEXT}(\text{Immediate})$

$I_{\langle 15:12 \rangle}$	$I_{\langle 11 \rangle}$	$I_{\langle 10:8 \rangle}$	$I_{\langle 7:0 \rangle}$
<i>Opcode</i>	<i>Imm</i>	<i>Rst</i>	<i>Immediate</i>
0 0 1 0	1	r r r	i i i i i i i i

### SLT - set if less than

Compara dois valores (com sinal, em complemento de 2). Se o primeiro for menor que o segundo, armazena 1 (verdadeiro) em um registrador. Senão, armazena 0 (falso). No tipo I, o segundo valor possui extensão de sinal. O cálculo do valor dessa instrução é definido por  $SLT = N \text{ xor } V$ , resultante de uma subtração realizada internamente e avaliação da diferença lógica dos qualificadores *negative* e *overflow*, também internos a ULA. O valor da condição SLT é armazenado no bit menos significativo do registrador destino, sendo os outros zerados.

- SLT Rst, RsA, RsB

```
if (GPR[RsA] < GPR[RsB]) GPR[Rst] ← 1
else GPR[Rst] ← 0
```

$I_{<15:12>}$	$I_{<11>}$	$I_{<10:8>}$	$I_{<7:5>}$	$I_{<4:2>}$	$I_{<1:0>}$
<i>Opcode</i>	<i>Imm</i>	<i>Rst</i>	<i>RsA</i>	<i>RsB</i>	<i>Op2</i>
0 0 1 1	0	r r r	r r r	r r r	0 0

- SLT Rst, Immediate

```
if (GPR[RsA] < SEXT(Immediate)) GPR[Rst] ← 1
else GPR[Rst] ← 0
```

$I_{<15:12>}$	$I_{<11>}$	$I_{<10:8>}$	$I_{<7:0>}$
<i>Opcode</i>	<i>Imm</i>	<i>Rst</i>	<i>Immediate</i>
0 0 1 1	1	r r r	i i i i i i i i

### SLTU - set if less than (unsigned)

Compara dois valores (sem sinal). Se o primeiro for menor que o segundo, armazena 1 (verdadeiro) em um registrador. Senão, armazena 0 (falso). No tipo I, o segundo valor possui extensão de sinal. O cálculo dessa instrução é definido por  $SLTU = C$ , resultante de uma subtração realizada internamente e avaliação do qualificador *carry* interno a ULA. O valor da condição SLTU é armazenado no bit menos significativo do registrador destino, sendo os outros zerados.

- SLTU Rst, RsA, RsB

```
if (GPR[RsA] < GPR[RsB]) GPR[Rst] ← 1
else GPR[Rst] ← 0
```

$I_{<15:12>}$	$I_{<11>}$	$I_{<10:8>}$	$I_{<7:5>}$	$I_{<4:2>}$	$I_{<1:0>}$
<i>Opcode</i>	<i>Imm</i>	<i>Rst</i>	<i>RsA</i>	<i>RsB</i>	<i>Op2</i>
0 1 0 0	0	r r r	r r r	r r r	0 0

- SLTU Rst, Immediate

```
if (GPR[RsA] < SEXT(Immediate)) GPR[Rst] ← 1
else GPR[Rst] ← 0
```



$I_{<15:12>}$	$I_{<11>}$	$I_{<10:8>}$	$I_{<7:0>}$
<i>Opcode</i>	<i>Imm</i>	<i>Rst</i>	<i>Immediate</i>
0 1 0 0	1	r r r	i i i i i i i i

### ADD - add

Soma dois valores e armazena o resultado em um registrador. No tipo I, o segundo valor possui extensão de sinal.

- ADD Rst, RsA, RsB

$$\text{GPR}[\text{Rst}] \leftarrow \text{GPR}[\text{RsA}] + \text{GPR}[\text{RsB}]$$

$I_{<15:12>}$	$I_{<11>}$	$I_{<10:8>}$	$I_{<7:5>}$	$I_{<4:2>}$	$I_{<1:0>}$
<i>Opcode</i>	<i>Imm</i>	<i>Rst</i>	<i>RsA</i>	<i>RsB</i>	<i>Op2</i>
0 1 0 1	0	r r r	r r r	r r r	0 0

- ADD Rst, Immediate

$$\text{GPR}[\text{Rst}] \leftarrow \text{GPR}[\text{Rst}] + \text{SEXT}(\text{Immediate})$$

$I_{<15:12>}$	$I_{<11>}$	$I_{<10:8>}$	$I_{<7:0>}$
<i>Opcode</i>	<i>Imm</i>	<i>Rst</i>	<i>Immediate</i>
0 1 0 1	1	r r r	i i i i i i i i

### SUB - subtract

Subtrai dois valores e armazena o resultado em um registrador. No tipo I, o segundo valor possui extensão de sinal.

- SUB Rst, RsA, RsB

$$\text{GPR}[\text{Rst}] \leftarrow \text{GPR}[\text{RsA}] - \text{GPR}[\text{RsB}]$$

$I_{<15:12>}$	$I_{<11>}$	$I_{<10:8>}$	$I_{<7:5>}$	$I_{<4:2>}$	$I_{<1:0>}$
<i>Opcode</i>	<i>Imm</i>	<i>Rst</i>	<i>RsA</i>	<i>RsB</i>	<i>Op2</i>
0 1 1 0	0	r r r	r r r	r r r	0 0

- SUB Rst, Immediate

$$\text{GPR}[\text{Rst}] \leftarrow \text{GPR}[\text{Rst}] - \text{SEXT}(\text{Immediate})$$

$I_{<15:12>}$	$I_{<11>}$	$I_{<10:8>}$	$I_{<7:0>}$
<i>Opcode</i>	<i>Imm</i>	<i>Rst</i>	<i>Immediate</i>
0 1 1 0	1	r r r	i i i i i i i i

### LDR - load register

Carrega uma constante de 8 bits em um registrador. O valor carregado possui extensão de sinal, o que facilita a carga de constantes de pequeno valor ( $\pm 128$ , em complemento de dois) com apenas uma instrução.

- LDR Rst, Immediate

$$\text{GPR}[\text{Rst}] \leftarrow \text{SEXT}(\text{Immediate})$$

$I_{<15:12>}$	$I_{<11>}$	$I_{<10:8>}$	$I_{<7:0>}$
<i>Opcode</i>	<i>Imm</i>	<i>Rst</i>	<i>Immediate</i>
1 0 0 0	1	r r r	i i i i i i i i

### LDC - load constant

Carrega uma constante em um registrador. O valor carregado não possui extensão de sinal. Antes de carregar o valor nos 8 bits menos significativos de um registrador, o mesmo tem seu conteúdo deslocado à esquerda, o que permite a carga de constantes de valores maiores que  $\pm 128$  com múltiplas instruções.

- LDC Rst, Immediate

$$\text{GPR}[\text{Rst}] \leftarrow (\text{GPR}[\text{Rst}] \ll 8) + \text{ZEXT}(\text{Immediate})$$

$I_{<15:12>}$	$I_{<11>}$	$I_{<10:8>}$	$I_{<7:0>}$
<i>Opcode</i>	<i>Imm</i>	<i>Rst</i>	<i>Immediate</i>
1 0 0 1	1	r r r	i i i i i i i i

## 1.4.2 Deslocamento

### LSR - logical shift right

Realiza a o deslocamento lógico por 1 bit à direita e armazena o resultado em um registrador.

- LSR Rst, RsA, r0

$$\text{GPR}[\text{Rst}] \leftarrow \text{GPR}[\text{RsA}] \gg 1$$

$I_{<15:12>}$	$I_{<11>}$	$I_{<10:8>}$	$I_{<7:5>}$	$I_{<4:2>}$	$I_{<1:0>}$
<i>Opcode</i>	<i>Imm</i>	<i>Rst</i>	<i>RsA</i>	<i>RsB</i>	<i>Op2</i>
0 0 0 0	0	r r r	r r r	0 0 0	0 1

### ASR - arithmetic shift right

Realiza a o deslocamento aritmético por 1 bit à direita e armazena o resultado em um registrador. O valor armazenado tem seu sinal mantido.

- ASR Rst, RsA, r0

$$\text{GPR}[\text{Rst}] \leftarrow \text{GPR}[\text{RsA}] \gg 1$$

$I_{<15:12>}$	$I_{<11>}$	$I_{<10:8>}$	$I_{<7:5>}$	$I_{<4:2>}$	$I_{<1:0>}$
<i>Opcode</i>	<i>Imm</i>	<i>Rst</i>	<i>RsA</i>	<i>RsB</i>	<i>Op2</i>
0 0 0 1	0	r r r	r r r	0 0 0	0 1

## 1.4.3 Carga e armazenamento

### LDB - load byte

Carrega um byte da memória. O endereço é obtido a partir do registrador base *RsB*. O valor é carregado na parte baixa do registrador destino *Rst*, e possui extensão de sinal.

- LDB Rst, r0, RsB

$$\text{GPR}[\text{Rst}] \leftarrow \text{SEXT}(\text{MEM}[\text{GPR}[\text{RsB}]]_{\langle 7:0 \rangle})$$

$I_{\langle 15:12 \rangle}$	$I_{\langle 11 \rangle}$	$I_{\langle 10:8 \rangle}$	$I_{\langle 7:5 \rangle}$	$I_{\langle 4:2 \rangle}$	$I_{\langle 1:0 \rangle}$
<i>Opcode</i>	<i>Imm</i>	<i>Rst</i>	<i>RsA</i>	<i>RsB</i>	<i>Op2</i>
0 0 0 0	0	r r r	0 0 0	r r r	1 0

### STB - store byte

Armazena um byte na memória. O endereço é obtido a partir do registrador base *RsB*. O valor armazenado encontra-se na parte baixa do registrador fonte *RsA*.

- STB r0, RsA, RsB

$$\text{MEM}[\text{GPR}[\text{RsB}]] \leftarrow \text{GPR}[\text{RsA}]_{\langle 7:0 \rangle}$$

$I_{\langle 15:12 \rangle}$	$I_{\langle 11 \rangle}$	$I_{\langle 10:8 \rangle}$	$I_{\langle 7:5 \rangle}$	$I_{\langle 4:2 \rangle}$	$I_{\langle 1:0 \rangle}$
<i>Opcode</i>	<i>Imm</i>	<i>Rst</i>	<i>RsA</i>	<i>RsB</i>	<i>Op2</i>
0 0 0 1	0	0 0 0	r r r	r r r	1 0

### LDW - load word

Carrega uma palavra da memória. O endereço é obtido a partir do registrador base *RsB* e deve estar alinhado ao tamanho da palavra (16 ou 32 bits). O valor é carregado no registrador destino *Rst*.

- LDW Rst, r0, RsB

$$\text{GPR}[\text{Rst}] \leftarrow \text{MEM}[\text{GPR}[\text{RsB}]]$$

$I_{\langle 15:12 \rangle}$	$I_{\langle 11 \rangle}$	$I_{\langle 10:8 \rangle}$	$I_{\langle 7:5 \rangle}$	$I_{\langle 4:2 \rangle}$	$I_{\langle 1:0 \rangle}$
<i>Opcode</i>	<i>Imm</i>	<i>Rst</i>	<i>RsA</i>	<i>RsB</i>	<i>Op2</i>
0 1 0 0	0	r r r	0 0 0	r r r	1 0

### STW - store word

Armazena uma palavra na memória. O endereço é obtido a partir do registrador base *RsB* e deve estar alinhado ao tamanho da palavra (16 ou 32 bits). O valor armazenado encontra-se no registrador fonte *RsA*.

- STW r0, RsA, RsB

$$\text{MEM}[\text{GPR}[\text{RsB}]] \leftarrow \text{GPR}[\text{RsA}]$$

$I_{\langle 15:12 \rangle}$	$I_{\langle 11 \rangle}$	$I_{\langle 10:8 \rangle}$	$I_{\langle 7:5 \rangle}$	$I_{\langle 4:2 \rangle}$	$I_{\langle 1:0 \rangle}$
<i>Opcode</i>	<i>Imm</i>	<i>Rst</i>	<i>RsA</i>	<i>RsB</i>	<i>Op2</i>
0 1 0 1	0	0 0 0	r r r	r r r	1 0

### 1.4.4 Desvios condicionais

#### BEZ - branch if equal zero

Realiza um desvio condicional, caso o valor de Fonte 1 seja zero. O endereço é obtido a partir do registrador base *RsB* ou relativo ao PC e deve estar alinhado ao tamanho de uma instrução (16 bits).

- BEZ r0, RsA, RsB

if (GPR[RsA] == zero) PC  $\leftarrow$  GPR[RsB]

$I_{\langle 15:12 \rangle}$	$I_{\langle 11 \rangle}$	$I_{\langle 10:8 \rangle}$	$I_{\langle 7:5 \rangle}$	$I_{\langle 4:2 \rangle}$	$I_{\langle 1:0 \rangle}$
<i>Opcode</i>	<i>Imm</i>	<i>Rst</i>	<i>RsA</i>	<i>RsB</i>	<i>Op2</i>
1 0 1 0	0	0 0 0	r r r	r r r	1 1

- BEZ Rst, Immediate

if (GPR[Rst] == zero) PC  $\leftarrow$  PC + SEXT(Immediate)

$I_{\langle 15:12 \rangle}$	$I_{\langle 11 \rangle}$	$I_{\langle 10:8 \rangle}$	$I_{\langle 7:0 \rangle}$
<i>Opcode</i>	<i>Imm</i>	<i>Rst</i>	<i>Immediate</i>
1 0 1 0	1	r r r	i i i i i i i i

#### BNZ - branch if not equal zero

Realiza um desvio condicional, caso o valor de Fonte 1 não seja zero. O endereço é obtido a partir do registrador base *RsB* ou relativo ao PC e deve estar alinhado ao tamanho de uma instrução (16 bits).

- BNZ r0, RsA, RsB

if (GPR[RsA] != zero) PC  $\leftarrow$  GPR[RsB]

$I_{\langle 15:12 \rangle}$	$I_{\langle 11 \rangle}$	$I_{\langle 10:8 \rangle}$	$I_{\langle 7:5 \rangle}$	$I_{\langle 4:2 \rangle}$	$I_{\langle 1:0 \rangle}$
<i>Opcode</i>	<i>Imm</i>	<i>Rst</i>	<i>RsA</i>	<i>RsB</i>	<i>Op2</i>
1 0 1 1	0	0 0 0	r r r	r r r	1 1

- BNZ Rst, Immediate

if (GPR[Rst] != zero) PC  $\leftarrow$  PC + SEXT(Immediate)

$I_{\langle 15:12 \rangle}$	$I_{\langle 11 \rangle}$	$I_{\langle 10:8 \rangle}$	$I_{\langle 7:0 \rangle}$
<i>Opcode</i>	<i>Imm</i>	<i>Rst</i>	<i>Immediate</i>
1 0 1 1	1	r r r	i i i i i i i i

A tabela a seguir apresenta um resumo das operações definidas na arquitetura. Importante observar que diversos *opcodes* não foram definidos, o que permite adição de novas instruções ao conjunto básico.

Instrução	Descrição	Opcode	Imm	Op2
AND	Logical product	0 0 0 0	x	0 0
OR	Logical sum	0 0 0 1	x	0 0
XOR	Logical difference	0 0 1 0	x	0 0
SLT	Set if less than	0 0 1 1	x	0 0
SLTU	Set if less than (unsigned)	0 1 0 0	x	0 0
ADD	Add	0 1 0 1	x	0 0
SUB	Subtract	0 1 1 0	x	0 0
LDR	Load register	1 0 0 0	1	0 0
LDC	Load constant	1 0 0 1	1	0 0
LSR	Logical shift right	0 0 0 0	0	0 1
ASR	Arithmetic shift right	0 0 0 1	0	0 1
LDB	Load byte	0 0 0 0	0	1 0
STB	Store byte	0 0 0 1	0	1 0
LDW	Load word	0 1 0 0	0	1 0
STW	Store word	0 1 0 1	0	1 0
BEZ	Branch if equal zero	1 0 1 0	x	1 1
BNZ	Branch if not equal zero	1 0 1 1	x	1 1

## 1.5 Características únicas

### 1.5.1 Carga de constantes

A carga de constantes pode ser realizada com as instruções LDR e LDC. A instrução LDR simplifica a carga de constantes com valor entre  $\pm 128$  e outras com valor negativo e maior magnitude. O objetivo de existir uma instrução específica para carga de valores pequenos é o fato da maior parte das constantes terem um valor nessa faixa, além de inicializar com a extensão de sinal a parte alta de um registrador. O valor -1 pode ser carregado diretamente com:

```
ldr r1,-1
```

Para constantes com valores fora da faixa de valores entre  $\pm 128$  uma sequência de instruções LDC (ou LDR + LDC) pode ser usada. Uma constante em uma arquitetura de 16 bits pode ser carregada pela seguinte sequência. O valor a ser carregado é  $1234_{16}$  e os bytes são carregados a partir do byte mais significativo<sup>6</sup>, sendo os valores especificados em decimal.

```
ldc r1,18
ldc r1,52
```

Para a carga da mesma constante em uma arquitetura de 32 bits, a sequência a seguir pode ser utilizada.

```
ldc r1,0
ldc r1,0
ldc r1,18
ldc r1,52
```

É importante observar que com a carga de todo o registrador qualquer informação antiga terá sido eliminada, uma vez que o registrador tem seu conteúdo deslocado à esquerda 8 bits a cada instrução. Uma maneira mais eficiente seria (desde que o valor do primeiro byte seja menor que 128):

<sup>6</sup>Mais detalhes sobre a ordem de bytes da arquitetura são apresentados na Seção 1.6.

```
ldr r1,18
ldc r1,52
```

O valor -31073 pode ser carregado com o par de instruções a seguir (assumindo que a instrução LDR utiliza uma constante sinalizada e LDC não):

```
ldr r1,-122
ldc r1,159
```

Outro exemplo seria a carga de constantes com valores de grande magnitude (32 bits). No exemplo, o valor a ser carregado é  $12345678_{16}$  (ou  $305419896_{10}$ ).

```
ldc r1,0x12
ldc r1,0x34
ldc r1,0x56
ldc r1,0x78
```

Para o caso de uma arquitetura de 32 bits, de uma a quatro instruções podem ser utilizadas, sendo que o número de instruções varia de acordo com a magnitude do valor da constante. Para uma versão de 16 bits, duas instruções LDC podem ser utilizadas para a carga de constantes fora da faixa de valor  $\pm 128$ .

### 1.5.2 Extensão de sinal

Para que valores imediatos (instruções do tipo I) possam ser utilizados para aritmética, é necessário que a sinalização adequada seja mantida (em complemento de dois). Para implementar a extensão de sinal, o valor do oitavo bit do campo imediato (bit 7) é replicado para todos os bits mais significativos de Fonte 2. O comportamento da extensão de sinal pode ser descrito como  $\text{SEXT}(\text{Immediate}) \leftarrow \text{Immediate}_{\langle 7 \rangle} \dots \text{Immediate}_{\langle 7:0 \rangle}$ . As únicas operações do tipo I que não utilizam extensão de sinal, ou seja utilizam extensão por zero, são as instruções AND, OR e LDC.

### 1.5.3 Desvios condicionais

São definidas duas instruções de desvios condicionais (BEZ e BNZ) na arquitetura, que comparam o valor de um registrador com zero e realizam desvios condicionalmente. O motivo para a definição dessas instruções, e não instruções mais genéricas que comparam o valor de um registrador com qualquer valor (como BEQ e BNE) é simples. No tipo de instrução R, três registradores são referenciados. Se dois valores a serem comparados estivessem em registrador, e mais um registrador de endereços fosse referenciado na mesma instrução, seriam necessárias três portas de leitura no banco de registradores. Além disso, seria necessário o uso de um multiplexador adicional para modificar a semântica dos campos *Rst*, *RsA* e *RsB* em instruções de desvio.

### 1.5.4 Outras operações

Algumas operações elementares como complemento, deslocamentos à esquerda e outros tipos de desvios são implementados na arquitetura com o uso de pseudo operações. Em algumas operações, não existe vantagem alguma em incluir *hardware* adicional para o seu suporte, uma

vez que as mesmas podem ser sintetizadas diretamente por outras equivalentes. Um exemplo é o deslocamento à esquerda, que pode ser obtido somando-se um valor a ele mesmo, não sendo necessária uma instrução separada para implementar esse comportamento.

Outras operações podem ser sintetizadas com sequências de poucas instruções elementares. Mais detalhes sobre tais operações são apresentadas no Capítulo 2.

## 1.6 Tipos de dados

Viking é uma arquitetura *big-endian*, ou seja, tipos compostos por múltiplos bytes possuem o endereço alinhado com o byte mais significativo. Dessa forma, o primeiro byte de uma instrução (mais significativo) é capaz de conter informação suficiente para definir operações que resultem em instruções com tamanho maior que 16 bits, uma possível extensão do formato de instruções. Na arquitetura Viking existem dois tipos de dados:

- Um *byte* possui 8 bits. Em operações de carga e armazenamento o byte mais significativo de uma palavra (dados, bits  $\langle 31:24 \rangle$  para 32 bits ou bits  $\langle 15:8 \rangle$  para 16 bits) é acessado quando o endereço estiver alinhado (endereço, bits  $\langle 1:0 \rangle = 0$  para 32 bits ou bit  $\langle 0 \rangle = 0$  para 16 bits) e o byte menos significativo é acessado quando os bits do endereço forem  $\langle 1:0 \rangle = 3$  (para palavras de 32 bits) e  $\langle 0 \rangle = 1$  (para palavras de 16 bits).
- Uma *palavra* possui 32 ou 16 bits (dependendo da implementação). Esse tipo possui seu byte mais significativo acessado na parte alta da palavra em operações de carga e armazenamento quando o endereço estiver alinhado (bits  $\langle 1:0 \rangle = 0$  para 32 bits ou bit  $\langle 0 \rangle = 0$  para 16 bits). Não são definidos acessos desalinhados para esse tipo.

## 1.7 Convenções de chamada de função

### 1.7.1 Pilha

Não há mecanismos ou instruções específicas para o gerenciamento da pilha. O programador é responsável por fazer a gerência manualmente, utilizando o registrador *r7* (*sp*) para essa finalidade. Por convenção, a pilha cresce do endereço mais alto para o endereço mais baixo, e esta deve ser inicializada com o endereço do topo da pilha no início do programa. Para implementar o comportamento de instruções estilo *PUSH* e *POP*, pode ser usado o seguinte padrão de código:

```
add sp,-2          # PUSH r1
stw r0,r1,sp
...
ldw r1,r0,sp       # POP r1
add sp,2
```

Importante observar que no código foi considerada uma implementação de 16 bits da arquitetura. Caso fossem utilizados registradores de 32 bits, seria necessário alocar / desalocar 4 bytes na pilha, e não 2 como apresentado no exemplo.

### 1.7.2 Registradores

Um conjunto de 8 registradores de propósito geral é definido na arquitetura. Por questões de interoperabilidade, as seguintes convenções são definidas para o uso de tais registradores. Importante observar que os nomes alternativos podem ser utilizados para designar os papéis de registradores específicos e tornar o código de montagem mais legível.

Registrador	Nome	Apelido	Papel	Preservado
0	r0	at	Temporário (montador)	Não
1	r1	r1	Variável local	Chamado
2	r2	r2	Variável local	Chamado
3	r3	r3	Variável local	Chamado
4	r4	r4	Variável local	Chamado
5	r5	sr	Temporário	Não
6	r6	lr	Endereço de retorno	Chamador
7	r7	sp	Apontador de pilha	Sim

Nos formatos de instruções em que um dos registradores especificado é fixo, deve-se utilizar a notação *r0*. Nos outros casos, o registrador 0 deve ser referenciado por *at*. O registrador *at* é reservado para a síntese de pseudo operações, e deve ser utilizado diretamente pelo programador apenas em situações em que não estão envolvidas pseudo operações. Os registradores *r1* a *r4* são de propósito geral e podem ser utilizados para avaliação de expressões e passagem de parâmetros. O registrador *sr* é um registrador temporário, e pode ser utilizado para qualquer finalidade. Para chamada de procedimentos e manipulação da pilha são utilizados os registradores *lr* e *sp* respectivamente.

Caso necessário, os registradores *sr* e *lr* podem ser utilizados como registradores de propósito geral. Para que o registrador *lr* possa ser utilizado com esse fim, seu conteúdo deve ser colocado na pilha no início da função, e restaurado no final antes de efetuado o retorno de função. Quando tratados como registradores de propósito geral, *sr* e *lr* devem ser referenciados por seus nomes *r5* e *r6*, ficando assim os registradores *r1* a *r6* (6 registradores) disponíveis para uso geral.

### 1.7.3 Chamada e retorno de funções

Em função do número reduzido de registradores na arquitetura, a passagem de parâmetros ocorre normalmente pela pilha. Apenas em casos onde não é desejável a manipulação da pilha (pequenas funções, por exemplo) os registradores *r1* a *r4* podem ser utilizados para essa finalidade. Nesse caso, é responsabilidade tanto da função chamadora quanto da função chamada definirem o protocolo adequado.

Não existem instruções nativas para o suporte de chamada e retorno de funções. Assim, para realizar a passagem de parâmetros pela pilha são necessárias as seguintes convenções:

- Usar o registrador *r5* (*scratch register*, *sr*) para o retorno de valores em funções. Se mais valores de retorno forem necessários, deve-se utilizar a pilha;
- Usar o registrador *r6* (*link register*, *lr*) como um registrador de endereço de retorno, e gerenciar o mesmo usando a pilha no caso de chamadas recursivas;
- Usar o registrador *r7* (*stack pointer*, *sp*) como apontador de pilha e fazer a sua gerência manualmente.



Uma chamada de função envolve gerenciar a passagem e retorno de parâmetros e endereços de chamada e retorno de função. Considerando as limitações da arquitetura, o seguinte protocolo pode ser usado:

1. Colocar os parâmetros na pilha (em ordem inversa);
2. Salvar  $lr$  na pilha;
3. Carregar  $lr$  com o endereço de retorno (um rótulo definido após a instrução de desvio que salta para a função chamada);
4. Carregar  $sr$  com o endereço da função a ser chamada;
5. Saltar para  $sr$  (chamada de função). Na função:
  - (a) Salvar  $r1$  até  $r4$  na pilha, se necessário;
  - (b) (Fazer o que for necessário);
  - (c) Escrever o resultado pelos parâmetros (ponteiros) ou em  $sr$ ;
  - (d) Restaurar registradores  $r1$  até  $r4$ , se necessário;
  - (e) Saltar para  $lr$  (retorno);
6. Na função chamadora, restaurar  $lr$  da pilha;
7. Liberar da pilha os parâmetros.

## Capítulo 2

# Síntese de pseudo instruções

Neste Capítulo são apresentadas diversas instruções que não fazem parte da arquitetura Viking, mas que podem ser sintetizadas de maneira simples. As operações apresentadas correspondem a instruções tipicamente encontradas em arquiteturas RISC, e servem para facilitar o desenvolvimento de programas em linguagem de montagem ou para a simplificação das listagens resultantes do processo de compilação.

Nas tabelas de instruções são apresentados o formato da instrução (pseudo operação) e a sua equivalência em uma sequência de instruções suportadas pela arquitetura. Em instruções que necessitam de um registrador temporário, *at* é utilizado para esse fim. O registrador *lr* é utilizado como endereço de retorno.

### 2.1 Pseudo operações básicas

Instruções de complemento são sintetizadas com operações XOR e SUB. Deslocamentos à esquerda são sintetizados com operações ADD. A carga de constantes é sintetizada de maneira trivial pelo montador, no entanto uma sequência mais otimizada pode ser gerada, como apresentado na Seção 1.5.1. O parâmetro *const* da pseudo operação LDI pode ser tanto um valor numérico quanto um rótulo, tendo seu valor resolvido pelo montador.

Operações de carga e armazenamento e desvios podem ser especificadas com apenas dois registradores, uma vez que para essas instruções um dos registradores não é utilizado fazendo com que o formato com três registradores se torne pouco intuitivo. Os parâmetros *addr* das operações BEZ e BNZ podem ser rótulos, sendo que essas operações fazem uso do registrador *at* para a carga do endereço. Isso simplifica o código de montagem pois o programador não precisa carregar o endereço manualmente. Outras operações que fazem uso de rótulos são LDB, STB, LDW e STW. A operação HCF não é definida pela arquitetura, e possui funcionalidade apenas no contexto de simulação (a simulação é abortada).

Nos formatos de pseudo operações suportadas pelo montador e pseudo operações adicionais, o registrador *r1* é exemplificado como registrador destino ou fonte da operação, enquanto *r2* é fonte.

Instrução	Descrição	Formato	Equivalência
NOP	No operation	<code>nop</code>	<code>and r0,r0,r0</code>
NOT	One's complement	<code>not r1</code>	<code>xor r1,-1</code>
NEG	Two's complement	<code>neg r1</code>	<code>xor r1,-1</code> <code>add r1,1</code>
LSR	Logical shift right	<code>lsr r1,r2</code>	<code>lsr r1,r2,r0</code>
ASR	Arithmetic shift right	<code>asr r1,r2</code>	<code>asr r1,r2,r0</code>
LSL	Logical shift left	<code>lsl r1,r2</code>	<code>add r1,r2,r2</code>
LDI	Load immediate	<code>ldi r1,const</code>	<code>ldc r1,byte0</code> <code>ldc r1,byte1</code> ...
BEZ	Branch if equal zero	<code>bez r1,r2</code>	<code>bez r0,r1,r2</code>
		<code>bez r1,addr</code>	<code>ldi at,addr</code> <code>bez r0,r1,at</code>
BNZ	Branch if not equal zero	<code>bnz r1,r2</code>	<code>bnz r0,r1,r2</code>
		<code>bnz r1,addr</code>	<code>ldi at,addr</code> <code>bnz r0,r1,at</code>
LDB	Load byte	<code>ldb r1,r2</code>	<code>ldb r1,r0,r2</code>
		<code>ldb r1,addr</code>	<code>ldi at,addr</code> <code>ldb r1,r0,at</code>
STB	Store byte	<code>stb r1,r2</code>	<code>stb r0,r1,r2</code>
		<code>stb r1,addr</code>	<code>ldi at,addr</code> <code>stb r0,r1,at</code>
LDW	Load word	<code>ldw r1,r2</code>	<code>ldw r1,r0,r2</code>
		<code>ldw r1,addr</code>	<code>ldi at,addr</code> <code>ldw r1,r0,at</code>
STW	Store word	<code>stw r1,r2</code>	<code>stw r0,r1,r2</code>
		<code>stw r1,addr</code>	<code>ldi at,addr</code> <code>stw r0,r1,at</code>
HCF	Halt and catch fire	<code>hcf</code>	0x0003 (padrão)

## 2.2 Operações de deslocamento

Nas operações de deslocamento que envolvem múltiplos bits o registrador *r1* é exemplificado como fonte e destino e *r2* contém o número de bits a serem deslocados. O conteúdo de *r2* também é modificado como resultado do processamento.

Instrução	Descrição	Formato	Equivalência
LSRM	Logical shift right multiple	<code>lsrm r1,r2</code>	<code>lsr r1,r1,r0</code> <code>sub r2,1</code> <code>bnz r2,-6</code>
ASRM	Arithmetic shift right multiple	<code>asrm r1,r2</code>	<code>asr r1,r1,r0</code> <code>sub r2,1</code> <code>bnz r2,-6</code>
LSLM	Logical shift left multiple	<code>lslm r1,r2</code>	<code>add r1,r1,r0</code> <code>sub r2,1</code> <code>bnz r2,-6</code>

## 2.3 Pseudo operações não suportadas pelo montador

### 2.3.1 Testes, seleção e desvios (condicionais)

Em pseudo operações que envolvem testes, os registradores *r2* e *r3* são exemplificados como operandos e *r1* como alvo. As operações SLT e SLTU já fazem parte do conjunto de instruções básico, e por isso não foram apresentadas na tabela.

Instrução	Descrição	Formato	Equivalência
SEQ	Set if equal	seq <i>r1,r2,r3</i>	sub <i>r1,r2,r3</i> sltu <i>r1,1</i>
SNE	Set if not equal	sne <i>r1,r2,r3</i>	sub <i>r1,r2,r3</i> xor <i>at,at,at</i> sltu <i>r1,at,r1</i>
SGE	Set if greater equal	sge <i>r1,r2,r3</i>	slt <i>r1,r2,r3</i> ldr <i>at,1</i> sub <i>r1,at,r1</i>
SGEU	Set if greater equal (unsigned)	sgeu <i>r1,r2,r3</i>	sltu <i>r1,r2,r3</i> ldr <i>at,1</i> sub <i>r1,at,r1</i>

Nos formatos de desvios condicionais, os registradores *r1* e *r2* são exemplificados como operandos, sendo o valor de *r1* não preservado. Um endereço é definido no rótulo *addr*.

Instrução	Descrição	Formato	Equivalência
BEQ	Branch if equal	beq <i>r1,r2,addr</i>	ldi <i>at,addr</i> sub <i>r1,r1,r2</i> bez <i>r0,r1,at</i>
BNE	Branch if not equal	bne <i>r1,r2,addr</i>	ldi <i>at,addr</i> sub <i>r1,r1,r2</i> bnz <i>r0,r1,at</i>
BLT	Branch if less than	blt <i>r1,r2,addr</i>	ldi <i>at,addr</i> slt <i>r1,r1,r2</i> bnz <i>r0,r1,at</i>
BGE	Branch if greater equal	bge <i>r1,r2,addr</i>	ldi <i>at,addr</i> slt <i>r1,r1,r2</i> bez <i>r0,r1,at</i>
BLTU	Branch if less than (unsigned)	bltu <i>r1,r2,addr</i>	ldi <i>at,addr</i> sltu <i>r1,r1,r2</i> bnz <i>r0,r1,at</i>
BGEU	Branch if greater equal (unsigned)	bgeu <i>r1,r2,addr</i>	ldi <i>at,addr</i> sltu <i>r1,r1,r2</i> bez <i>r0,r1,at</i>

### 2.3.2 Operações condicionais equivalentes

Outras operações condicionais são equivalentes às definidas anteriormente, sendo apenas necessário inverter a ordem dos operandos. Por exemplo, a instrução BLE é a mesma que BGE porém com os operandos invertidos.

Instrução	Descrição	Formato	Equivalência
SGT	Set if greater equal	<code>sgt r1,r2,r3</code>	<code>slt r1,r3,r2</code>
SLE	Set if less equal	<code>sle r1,r2,r3</code>	<code>sge r1,r3,r2</code>
SGTU	Set if greater than (unsigned)	<code>sgtu r1,r2,r3</code>	<code>sltu r1,r3,r2</code>
SLEU	Set if less equal (unsigned)	<code>sleu r1,r2,r3</code>	<code>sgeu r1,r3,r2</code>
BGT	Branch if greater than	<code>bgt r1,r2,r3</code>	<code>blt r2,r1,r3</code>
BLE	Branch if less equal	<code>ble r1,r2,r3</code>	<code>bge r2,r1,r3</code>
BGTU	Branch if greater than (unsigned)	<code>bgtu r1,r2,r3</code>	<code>bltu r2,r1,r3</code>
BLEU	Branch if less equal (unsigned)	<code>bleu r1,r2,r3</code>	<code>bgeu r2,r1,r3</code>

### 2.3.3 Desvios incondicionais

Desvios incondicionais, assim como operações de chamada e retorno de subrotina podem ser trivialmente emuladas. Assume-se que  $r7$  ( $sp$ ) seja sempre diferente de zero.

Instrução	Descrição	Formato	Equivalência
JMP	Jump	<code>jmp addr</code>	<code>ldi at,addr</code> <code>bnz r0,r7,at</code>
JAL	Jump and link	<code>jal addr</code>	<code>ldi at,addr</code> <code>ldi lr,raddr</code> <code>bnz r0,r7,at</code>
JMPR	Jump register	<code>jmp r1</code>	<code>bnz r0,r7,r1</code>
JALR	Jump and link register	<code>jalr r1</code>	<code>ldi lr,raddr</code> <code>bnz r0,r7,r1</code>
RET	Return	<code>ret</code>	<code>bnz r0,r7,lr</code>

### 2.3.4 Operações aritméticas adicionais

Para operações de multiplicação, divisão e resto são necessárias chamadas para funções que emulam tais instruções. Nessas operações, os registradores  $r2$  e  $r3$  são exemplificados como operandos e  $r1$  como alvo. As rotinas *mulsi3* (multiplicação), *divsi3* (divisão) e *modsi3* (resto) são apresentadas no Apêndice B.

Instrução	Descrição	Formato	Equivalência
MUL / DIV / REM	Multiply / Divide / Division remainder	<code>mul r1,r2,r3 /</code> <code>div r1,r2,r3 / rem</code> <code>r1,r2,r3</code>	<code>sub sp,2</code> <code>stw r0,r2,sp</code> <code>sub sp,2</code> <code>stw r0,r3,sp</code> <code>sub sp,2</code> <code>stw r0,lr,sp</code> <code>ldi lr,raddr</code> <code>ldi sr,mulsi3 /</code> <code>divsi3 / modsi3</code> <code>bnz r0,r7,sr</code> <code>ldw lr,r0,sp</code> <code>add sp,6</code> <code>add r1,r0,sr</code>

## Capítulo 3

# Montagem de código e simulação

### 3.1 Montador

O montador possui uma sintaxe bastante simples, não sendo necessário definir regiões separadas para código e dados e diretivas tradicionalmente utilizadas em montadores de outras arquiteturas. O programa montador foi descrito com a linguagem Python, em função de sua facilidade natural de manipular texto e poder servir como referência para implementações mais completas e com um desempenho melhor.

#### 3.1.1 Formato da linguagem de montagem

Rótulos são utilizados para declarar pontos específicos (deslocamentos) no código, como destinos de saltos, endereço de entrada de funções ou procedimentos e também endereços de estruturas de dados (variáveis e vetores). O montador é responsável por resolver o valor dos rótulos, permitindo que as referências à memória assumam um valor numérico para a codificação das instruções em linguagem de máquina.

Instruções são representadas por seus mnemônicos, e em sua maioria possuem parâmetros que especificam o modo de endereçamento utilizado (R ou I) e operandos. Os mnemônicos que representam instruções, assim como as referências à registradores, são traduzidos pelo montador. Algumas poucas pseudo-operações não possuem parâmetros, como NOP e HCF. As regras para um programa de montagem válido são:

- Comentários devem ser iniciados por um caracter ponto e vírgula (;) à esquerda, sem tabulações.
- Rótulos devem ser declarados com alinhamento à esquerda, sem tabulações, e sem finalizador (dois pontos).
- Instruções devem ser alinhadas à esquerda, com uma única tabulação.
- Instruções devem ser representadas por dois campos: mnemônico e parâmetros (se existirem). O separador dos dois campos pode ser um espaço ou uma tabulação.
- Os elementos que compõem parâmetros de uma instrução devem ser separados por vírgula e sem espaços.

- Rótulos sem parâmetros definem endereços (deslocamentos) no código, e com parâmetros estruturas de dados e sua posição na memória.
- Estruturas de dados são definidas por dois tipos básicos (byte e inteiro). No tipo byte, os valores são representados por um conjunto de bytes e no tipo inteiro podem ser definidos por apenas um valor (variável) ou uma lista de valores separados por um espaço (vetor de inteiros).
- Valores das estruturas de dados podem ser bytes (*string*) delimitados por aspas ou valores numéricos, representados em decimal (123), hexadecimal (0x123), octal (0o123) ou binário (0b1010).
- Caracteres especiais aceitos em *strings* são `\t`, `\n` e `\r`.
- Instruções e dados podem ser misturados.

Para a montagem de código, são realizadas três passadas em sequência. Cada uma possui um papel fundamental na transformação do programa em linguagem de montagem para código de máquina. A sequência para a montagem de um programa com relação às passadas pelo código fonte em linguagem de montagem é a seguinte:

1. Pseudo-operações são convertidas para operações básicas equivalentes ou sequências (padrões) de instruções suportadas pela arquitetura;
2. Rótulos são resolvidos (convertidos) para endereços e uma tabela de símbolos é montada;
3. Instruções e dados são montados (traduzidos), um a um, a partir da listagem gerada no passo anterior e da tabela de símbolos.

### 3.1.2 Sintaxe de linha de comando

A entrada e saída padrão devem ser utilizadas para processar um arquivo em linguagem de montagem e armazenar o código objeto gerado. Além disso, o script do montador deve ser invocado com o interpretador Python (versão 2.7):

```
$ python assemble16.py < input.asm > output.out
```

O seguinte código em linguagem de montagem,

Listing 3.1: ninetoone.asm

```
1 main
2     ldi r1,9
3     ldi r2,32
4 loop
5     ldw sr,writei
6     stw r1,sr
7     ldw sr,writec
8     stw r2,sr
9     sub r1,1
10    bnz r1,loop
11    hcf
12
13 writec 0xf000
14 writei 0xf002
```

após ser processado pelo montador, resulta no seguinte código objeto:

Listing 3.2: ninetoone.out

<pre> 1 0000 9900 2 0002 9909 3 0004 9a00 4 0006 9a20 5 0008 9800 6 000a 9824 7 000c 4502 8 000e 5036 9 0010 9800 </pre>	<pre> 10 0012 9822 11 0014 4502 12 0016 5056 13 0018 6901 14 001a 9800 15 001c 9808 16 001e b020 17 0020 0003 18 0022 f000 19 0024 f002 </pre>
--	--

O arquivo de entrada *input.asm* será processado e o código objeto (pronto para ser executado no simulador) será armazenado em *output.txt*. Uma listagem completa é obtida (para depuração do código, por exemplo), se o script for executado com o parâmetro *debug*:

```
$ python assemble16.py debug < input.asm > output.out
```

O resultado será uma listagem contendo além dos endereços e código objeto, os rótulos e código intermediário do processo de montagem. O simulador não pode executar essa listagem diretamente, no entanto.

Listing 3.3: ninetoone\_debug.out

<pre> 1 main 2 0000 9900      ldc0  r1,9 3 0002 9909      ldc1  r1,9 4 0004 9a00      ldc0  r2,32 5 0006 9a20      ldc1  r2,32 6 loop 7 0008 9800      ldc0  at,writei 8 000a 9824      ldc1  at,writei 9 000c 4502      ldw  sr,r0,at 10 000e 5036     stw  r0,r1,sr </pre>	<pre> 11 0010 9800      ldc0  at,writec 12 0012 9822      ldc1  at,writec 13 0014 4502      ldw  sr,r0,at 14 0016 5056      stw  r0,r2,sr 15 0018 6901      sub  r1,l 16 001a 9800      ldc0  at,loop 17 001c 9808      ldc1  at,loop 18 001e b020      bnz  r0,r1,at 19 0020 0003      hcf  r0,r0,r0 20 21 0022 f000      writec 0xf000 22 0024 f002      writei 0xf002 </pre>
--	---

Caso ocorra algum erro de montagem, o script irá terminar silenciosamente. Erros de montagem podem ser verificados no código objeto gerado, onde nas linhas em que ocorreram erros será apresentado um padrão *\*\*\*\* ????.* O código objeto resultante será rejeitado pelo simulador caso exista algum erro na montagem.

Diversos arquivos de código fonte podem ser combinados (concatenados) e usados como uma única entrada para o montador. A sintaxe é:

```
$ cat fonte1.asm fonte2.asm fonte3.asm | python assemble16.py > output.out
```

## 3.2 Simulador

Assim como o programa montador, o simulador foi implementado na linguagem Python. Apesar da simulação ser bastante lenta em função do interpretador Python, a descrição mostrou-se adequada para a verificação do comportamento da arquitetura. Essa implementação de referência é simples de ser entendida, o que permite um porte fácil do simulador para outras linguagens de alto desempenho (como C, por exemplo).



### 3.2.1 Mapa de memória

O simulador implementa o modelo de execução da arquitetura Viking, incluindo uma memória e mecanismos básicos de entrada e saída. O espaço de endereçamento é compartilhado entre dados e instruções, por questões de simplicidade. Os espaços de endereçamento possuem algumas diferenças entre os simuladores da arquitetura de 16 e 32 bits.

Papel	16 bits	32 bits
Código + dados (início)	0x0000	0x00000000
Apontador de pilha	0xdffe	0x000ffffc
Saída (character)	0xf000	0xf0000000
Saída (inteiro)	0xf002	0xf0000004
Entrada (character)	0xf004	0xf0000008
Entrada (inteiro)	0xf006	0xf000000c

No início da simulação, o apontador de pilha (*sp*) é inicializado para o topo da pilha, que coincide com o final da memória. A execução do programa começa a partir do endereço zero, após o programa ser carregado para a memória.

### 3.2.2 Sintaxe de linha de comando

Assim como o montador, a entrada e saída padrão são usadas pelo simulador para a leitura do código objeto e dispositivos de entrada e saída apresentados no mapa de memória. Para a execução de um programa, o simulador deve ser invocado da seguinte forma:

```
$ python run16.py < output.out
```

```
[program (code + data): 38 bytes]
[memory size: 57344]
9 8 7 6 5 4 3 2 1
[ok]
112 cycles
```

Nesse caso, *output.out* foi gerado no processo de montagem e é usado como entrada para o simulador. Caso seja necessário executar o programa instrução por instrução, pode-se usar o parâmetro *debug*:

```
$ python run16.py debug < output.out
```

Case seja necessário montar o programa e executá-lo no simulador, é possível invocar o montador e direcionar sua saída à entrada do simulador, através de um *pipe*. Com isso, evita-se a necessidade de criação de um arquivo intermediário, e pode-se executar o programa a partir de seu código de montagem:

```
$ python assemble16.py < input.asm | python run16.py
```

# Apêndice A

## Exemplos

Listing A.1: hello\_world.asm

```
1 main
2     ldw sr, writec
3     ldi r4, str
4     ldi r3, loop
5 loop
6     ldb r2, r4
```

```
7     stw r2, sr
8     add r4, 1
9     bnz r2, r3
10    hcf
11
12 writec 0xf000
13 str "hello world!"
```

Listing A.2: fibonacci.asm

```
1 main
2     xor r1, r1, r1
3     ldi r2, 1
4     ldi r4, 21
5 fib_loop
6     ldw sr, writei
7     stw r1, sr
8     ldw sr, writec
9     ldi r3, 32
10    stw r3, sr
```

```
11
12    add r3, r1, r2
13    and r1, r2, r2
14    and r2, r3, r3
15
16    sub r4, 1
17    bnz r4, fib_loop
18    hcf
19
20 writec 0xf000
21 writei 0xf002
```

Listing A.3: function\_call.asm

```
1 main
2     ldi r1, str1
3     sub sp, 2
4     stw r1, sp
5     sub sp, 2
6     stw lr, sp
7     ldi lr, ret_print1
8     ldi sr, print_str
9     bnz r7, sr
10 ret_print1
11    ldw lr, sp
12    add sp, 4
13
14    ldi r1, str2
15    sub sp, 2
16    stw r1, sp
17    sub sp, 2
18    stw lr, sp
```

```
19    ldi lr, ret_print2
20    ldi sr, print_str
21    bnz r7, sr
22 ret_print2
23    ldw lr, sp
24    add sp, 4
25
26    hcf
27
28 print_str
29    ldw sr, writec
30    sub sp, 2
31    stw r1, sp
32    sub sp, 2
33    stw r2, sp
34
35    and r1, sp, sp
36    add r1, 6
37    ldw r1, r1
```

```

38 print_loop
39     ldb r2,r1
40     stw r2,sr
41     add r1,1
42     bnz r2,-8
43
44     ldw r2,sp
45     add sp,2

```

Listing A.4: mult.asm

```

1  main
2      ldw r2,readi
3      ldw r2,r2
4      ldw r3,readi
5      ldw r3,r3
6
7      sub sp,2
8      stw r2,sp
9      sub sp,2
10     stw r3,sp
11     sub sp,2
12     stw lr,sp

```

Listing A.5: bubble\_sort.asm

```

1  main
2      ldi sr,array1
3      sub sp,2
4      stw sr,sp
5      ldw sr,array1_sz
6      lsl sr,sr
7      sub sp,2
8      stw sr,sp
9      sub sp,2
10     stw lr,sp
11     ldi lr,ret_sort1
12     ldi sr,sort
13     bnz r7,sr
14 ret_sort1
15     ldw lr,sp
16     add sp,4
17
18     ldi r1,array1
19     ldw r2,array1_sz
20 print_array1
21     ldw r3,r1
22     ldw r4,writei
23     stw r3,r4
24     ldi r3,32
25     ldw r4,wrotec
26     stw r3,r4
27     add r1,2
28     sub r2,1
29     bnz r2,print_array1
30     ldi r3,10
31     stw r3,r4

```

```

46     ldw r1,sp
47     add sp,2
48     bnz r7,lr
49
50
51 writec 0xf000
52 str1   "this is the first call\n"
53 str2   "and this is the second!\n"

```

```

13     ldi lr,ret_addr
14     ldi sr,mulsi3
15     bnz r7,sr
16 ret_addr
17     ldw lr,sp
18     add sp,6
19
20     and r1,sr,sr
21     ldw sr,writei
22     stw r1,sr
23     hcf
24
25 writei 0xf002
26 readi 0xf006

```

```

32
33     ldi sr,array2
34     sub sp,2
35     stw sr,sp
36     ldw sr,array2_sz
37     lsl sr,sr
38     sub sp,2
39     stw sr,sp
40     sub sp,2
41     stw lr,sp
42     ldi lr,ret_sort2
43     ldi sr,sort
44     bnz r7,sr
45 ret_sort2
46     ldw lr,sp
47     add sp,4
48
49     ldi r1,array2
50     ldw r2,array2_sz
51 print_array2
52     ldw r3,r1
53     ldw r4,writei
54     stw r3,r4
55     ldi r3,32
56     ldw r4,wrotec
57     stw r3,r4
58     add r1,2
59     sub r2,1
60     bnz r2,print_array2
61     ldi r3,10
62     stw r3,r4
63

```

```

64     hcf
65
66 sort
67     sub sp,2
68     stw r1,sp
69     sub sp,2
70     stw r2,sp
71     sub sp,2
72     stw r3,sp
73     sub sp,2
74     stw r4,sp
75     xor r1,r1,r1
76 beg_loop_i
77     and r4,sp,sp
78     add r4,10
79     ldw r4,r4
80     sub r4,2
81     slt r4,r4,r1
82     bnz r4,end_loop_i
83     and r2,r1,r1
84     add r2,2
85 beg_loop_j
86     and r4,sp,sp
87     add r4,10
88     ldw r4,r4
89     sub r4,2
90     slt r4,r4,r2
91     bnz r4,end_loop_j
92     and sr,sp,sp
93     add sr,12
94     ldw r3,sr
95     add r3,r3,r1
96     ldw r3,r3
97     ldw r4,sr
98     add r4,r4,r2
99     ldw r4,r4
100    sub sp,2
101    stw r3,sp
102    slt r3,r3,r4
103    bnz r3,no_swap
104    and sr,sp,sp
105    add sr,14
106    ldw r3,sr
107    add r3,r3,r1
108    stw r4,r3
109    ldw r4,sr
110    add r4,r4,r2
111    ldw r3,sp
112    stw r3,r4
113 no_swap
114    add sp,2
115    add r2,2
116    bnz r7,beg_loop_j
117 end_loop_j
118    add r1,2
119    bnz r7,beg_loop_i
120 end_loop_i
121    ldw r4,sp
122    add sp,2
123    ldw r3,sp
124    add sp,2
125    ldw r2,sp
126    add sp,2
127    ldw r1,sp
128    add sp,2
129    bnz r7,lr
130
131 writec    0xf000
132 writei    0xf002
133 array1    -5 3 23 -64 34 3 65 7 10 -4 10
134 array1_sz 11
135 array2    13121 6686 12335 6172 -13028
            -4379 -3953 16045 -7613 -12561 -7188
            -7141 -6281 8039 -12760 -2041 6212
            -146 -3087 9151 -14015 7819 6590
            -13079 549 13277 9033 -8114 -3338
            -5071
136 array2_sz 30

```

## Apêndice B

# Rotinas *mulsi3*, *divsi3*, *modsi3* e *udivmodsi4*

Listing B.1: mulsi3.asm

```
1 mulsi3
2   sub sp,2
3   stw r1,sp
4   sub sp,2
5   stw r2,sp
6   sub sp,2
7   stw r3,sp
8
9   and r3,sp,sp
10  add r3,10
11  ldw r2,r3
12  sub r3,2
13  ldw r3,r3
14
15  xor r1,r1,r1
16  bez r3,14
17  and sr,r3,r3
18  and sr,1
19  bez sr,2
20  add r1,r1,r2
21  lsl r2,r2
22  lsr r3,r3
23  bnz r7,-16
24
25  and sr,r1,r1
26  add sp,2
27  ldw r3,sp
28  add sp,2
29  ldw r2,sp
30  add sp,2
31  ldw r1,sp
32
33  bnz r7,lr
```

Listing B.2: divsi3.asm

```
1 divsi3
2   sub sp,2
3   stw r1,sp
4   sub sp,2
5   stw r2,sp
6   sub sp,2
7   stw r3,sp
8
9   and r2,sp,sp
10  add r2,10
11  ldw r1,r2
12  sub r2,2
13  ldw r2,r2
14  xor r3,r3,r3
15
16  xor at,at,at
17  slt sr,r1,at
18  bez sr,4
19  sub r1,at,r1
20  or r3,1
21  slt sr,r2,at
22  bez sr,4
23  sub r2,at,r2
24  xor r3,1
25
26  sub sp,2
27  stw r1,sp
28  sub sp,2
29  stw r2,sp
30  sub sp,2
31  ldr sr,0
32  stw sr,sp
33  sub sp,2
34  stw lr,sp
```

```

35     ldi lr,ret_divsi3
36     ldi sr,udivmodsi4
37     bnz r7,sr
38 ret_divsi3
39     ldw lr,sp
40     add sp,8
41     bez r3,4
42     xor at,at,at
43     sub sr,at,sr

```

Listing B.3: modsi3.asm

```

1  modsi3
2     sub sp,2
3     stw r1,sp
4     sub sp,2
5     stw r2,sp
6     sub sp,2
7     stw r3,sp
8
9     and r2,sp,sp
10    add r2,10
11    ldw r1,r2
12    sub r2,2
13    ldw r2,r2
14    xor r3,r3,r3
15
16    xor at,at,at
17    slt sr,r1,at
18    bez sr,4
19    sub r1,at,r1
20    or  r3,1
21    slt sr,r2,at
22    bez sr,4
23    sub r2,at,r2
24    xor r3,1
25

```

Listing B.4: udivmodsi4.asm

```

1  udivmodsi4
2     sub sp,2
3     stw r1,sp
4     sub sp,2
5     stw r2,sp
6     sub sp,2
7     stw r3,sp
8     sub sp,2
9     stw r4,sp
10
11    ldr r3,1
12    xor r4,r4,r4
13
14    and r2,sp,sp
15    add r2,14
16    ldw r1,r2
17    sub r2,2

```

```

44
45    add sp,2
46    ldw r3,sp
47    add sp,2
48    ldw r2,sp
49    add sp,2
50    ldw r1,sp
51
52    bnz r7,lr
26    sub sp,2
27    stw r1,sp
28    sub sp,2
29    stw r2,sp
30    sub sp,2
31    ldr sr,1
32    stw sr,sp
33    sub sp,2
34    stw lr,sp
35    ldi lr,ret_modsi3
36    ldi sr,udivmodsi4
37    bnz r7,sr
38 ret_modsi3
39    ldw lr,sp
40    add sp,8
41    bez r3,4
42    xor at,at,at
43    sub sr,at,sr
44
45    add sp,2
46    ldw r3,sp
47    add sp,2
48    ldw r2,sp
49    add sp,2
50    ldw r1,sp
51
52    bnz r7,lr

```

```

18    ldw r2,r2
19
20    sltu sr,r2,r1
21    bez sr,8
22    bez r3,6
23    lsl r2,r2
24    lsl r3,r3
25    bnz r7,-12
26    sltu sr,r1,r2
27    bnz sr,4
28    sub r1,r1,r2
29    add r4,r4,r3
30    lsr r3,r3
31    lsr r2,r2
32    bnz r3,-14
33
34    and sr,sp,sp
35    add sr,10

```

36	ldw	sr, sr	44	ldw	r3, sp
37	bez	sr, 4	45	add	sp, 2
38	and	sr, r1, r1	46	ldw	r2, sp
39	bez	sr, 2	47	add	sp, 2
40	and	sr, r4, r4	48	ldw	r1, sp
41			49	add	sp, 2
42	ldw	r4, sp	50	bnz	r7, lr
43	add	sp, 2			