Viking CPU - Manual de referência v0.5

Sérgio Johann Filho

19 de julho de 2018

Sumário

1 A arquitetura Viking 2
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 6
1.4.1 Computação 6
1.4.2 Deslocamento e rotação 10
1.4.3 Carga e armazenamento 11
1.4.4 Desvios condicionais 12
1.5 Detalhes sobre a codificação de instruções 13
1.6 Características únicas 14
1.6.1 Carga de constantes 14
1.6.2 Extensão de sinal 15
1.6.3 Desvios condicionais 15
1.6.4 Soma e subtração com *carry* 16
1.6.5 Deslocamento com *carry* 16
1.6.6 Comparações com *carry* 17
1.6.7 Outras operações 17
1.7 Tipos de dados 18

2 Síntese de pseudo operações 19
2.1 Pseudo operações básicas 19
2.2 Operações de deslocamento 20
2.3 Pseudo operações não suportadas pelo montador 21
2.3.1 Testes, seleção e desvios (condicionais) 21
2.3.2 Operações condicionais equivalentes 21
2.3.3 Desvios incondicionais 21
2.3.4 Operações aritméticas adicionais 22

3 Programando com o processador Viking 23
3.1 Controle de fluxo do programa 23
3.1.1 Seleção 23
3.1.2 Repetição 25
3.2 Acesso à memória - variáveis 26
3.3 Acesso à memória - vetores 27
3.4 Chamadas de função e convenções de chamada 28
3.4.1 Pilha 28
3.4.2 Registradores 28
3.4.3 Chamada e retorno de funções 29

4 Montagem de código e simulação 31
4.1 Montador 31
4.1.1 Formato da linguagem de montagem 31
4.1.2 Sintaxe de linha de comando 32

4.2 Simulador 33
4.2.1 Mapa de memória 34
4.2.2 Sintaxe de linha de comando 34

A Exemplos 35

B Rotinas *mulsi3*, *divsi3*, *modsi3* e *udivmodsi4* 38

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[1], 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[2] 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 (20 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, ADC, SUB, SBC, LDR, LDC)
2. *Deslocamento e rotação* (LSR, ASR, ROR)
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.

[1] 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.

[2] A arquitetura possui uma excessão, que é a geração e inclusão condicional de *carry* para operações aritméticas e de deslocamento.

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 tabela abaixo e detalhadas na Seção 3.4 para a chamada de funções. Os registradores *r0* (*at*) e *r7* (*sp*) são respectivamente utilizados como *temporário* e *ponteiro de pilha*. Esses registradores não devem ser tratados da mesma forma que os outros. O temporário é usado por pseudo operações (apresentadas na Seção 2) e o ponteiro de pilha para armazenamento de dados e chamadas de função. Outro papel desse registrador é a implementação de desvios incondicionais, uma vez que é seguro assumir que seu valor nunca será zero durante a execução normal de um programa.

Registrador	**Nome**	**Apelido**	**Papel**
0	r0	at	Especial
1	r1	r1	Uso geral
2	r2	r2	Uso geral
3	r3	r3	Uso geral
4	r4	r4	Uso geral
5	r5	sr	Uso geral
6	r6	lr	Uso geral
7	r7	sp	Especial

Além dos 8 registradores de propósito geral (GPRs), é definido na arquitetura um registrador com a finalidade de 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.

$I_{<15:12>}$	$I_{<11>}$	$I_{<10:8>}$	$I_{<7:5>}$	$I_{<4:2>}$	$I_{<1:0>}$
Opcode	*Imm*	*Rst*	*RsA*	*RsB*	*Op2*
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 4. 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*[3].

Operação	**Significado**
`add r3,r1,r2`	r3 = r1 + r2
`ldb r3,r0,r2`	r3 = MEM[r2]
`stw r0,r1,r2`	MEM[r2] = r1
`and r2,r3,r4`	r2 = r3 and r4
`bez r0,r2,r3`	if (r2 == zero) PC = r3
`slt r3,r1,r2`	if (r1 < r2) r3 = 1, else r3 = 0
`lsr r5,r3,r0`	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.

$I_{<15:12>}$	$I_{<11>}$	$I_{<10:8>}$	$I_{<7:0>}$
Opcode	*Imm*	*Rst*	*Immediate*
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:

[3]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.

- *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[4] e representado em complemento de 2). Dessa forma, é possível realizar desvios relativos ao PC de ±128 bytes[5], 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.

Operação	**Significado**
`add r5,10`	r5 = r5 + 10
`or r2,1`	r2 = r2 or 1
`xor r5,-1`	r5 = r5 xor -1 = not r5
`ldr r3,5`	r3 = 5
`ldc r3,10`	r3 = (r3 << 8) or 10
`slt r4,10`	if (r4 < 10) r4 = 1, else r4 = 0
`bez r4,28`	if (r4 == 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[6] 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.

[4] A implementação de extensão de sinal é apresentada na Seção 1.6.2.

[5] No futuro o campo *Immediate* poderá codificar apenas a magnitude alinhada, o que aumenta o alcance dos desvios relativos para ±256 bytes.

[6] No Capítulo 2 são apresentadas pseudo operações que emulam o modo de endereçamento direto.

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

```
GPR[Rst] ← GPR[RsA] and GPR[RsB]
```

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

- AND Rst, Immediate

```
GPR[Rst] ← GPR[Rst] and ZEXT(Immediate)
```

$I_{<15:12>}$	$I_{<11>}$	$I_{<10:8>}$	$I_{<7:0>}$
Opcode	*Imm*	*Rst*	*Immediate*
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

```
GPR[Rst] ← GPR[RsA] or GPR[RsB]
```

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

- OR Rst, Immediate

```
GPR[Rst] ← GPR[Rst] or ZEXT(Immediate)
```

$I_{<15:12>}$	$I_{<11>}$	$I_{<10:8>}$	$I_{<7:0>}$
Opcode	*Imm*	*Rst*	*Immediate*
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

  `GPR[Rst] ← GPR[RsA] xor GPR[RsB]`

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

- XOR Rst, Immediate

  `GPR[Rst] ← GPR[Rst] xor SEXT(Immediate)`

$I_{<15:12>}$	$I_{<11>}$	$I_{<10:8>}$	$I_{<7:0>}$
Opcode	*Imm*	*Rst*	*Immediate*
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 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`

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

$I_{<15:12>}$	$I_{<11>}$	$I_{<10:8>}$	$I_{<7:0>}$
Opcode	*Imm*	*Rst*	*Immediate*
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
  ```

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

$I_{<15:12>}$	$I_{<11>}$	$I_{<10:8>}$	$I_{<7:0>}$
Opcode	*Imm*	*Rst*	*Immediate*
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

  ```
  GPR[Rst] ← GPR[RsA] + GPR[RsB]
  ```

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

- ADD Rst, Immediate

  ```
  GPR[Rst] ← GPR[Rst] + SEXT(Immediate)
  ```

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

ADC - add with carry

Soma dois valores e armazena o resultado em um registrador. A soma inclui o *carry* gerado pela última instrução.

- ADC Rst, RsA, RsB

  `GPR[Rst] ← GPR[RsA] + GPR[RsB] + Carry`

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

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

  `GPR[Rst] ← GPR[RsA] - GPR[RsB]`

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

- SUB Rst, Immediate

  `GPR[Rst] ← GPR[Rst] - SEXT(Immediate)`

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

SBC - subtract with carry

Subtrai dois valores e armazena o resultado em um registrador. A subtração inclui o *carry* gerado pela última instrução.

- SBC Rst, RsA, RsB

  `GPR[Rst] ← GPR[RsA] - GPR[RsB] - Carry`

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

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 (±128, em complemento de dois) com apenas uma instrução.

- LDR Rst, Immediate

  `GPR[Rst]` ← `SEXT(Immediate)`

$I_{<15:12>}$	$I_{<11>}$	$I_{<10:8>}$	$I_{<7:0>}$
Opcode	*Imm*	*Rst*	*Immediate*
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 por 8 bits, o que permite a carga de constantes de valores maiores que ±128 com múltiplas instruções.

- LDC Rst, Immediate

  `GPR[Rst]` ← $GPR[Rst]_{<7/24:0>}$ `& ZEXT(Immediate)`

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

1.4.2 Deslocamento e rotação

LSR - logical shift right

Realiza a o deslocamento lógico por 1 bit à direita e armazena o resultado em um registrador. O valor de *carry* é modificado com o valor do *bit* perdido.

- LSR Rst, RsA, r0

  `GPR[Rst]` ← `'0' &` $GPR[RsA]_{<15/31:1>}$

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

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. O valor de *carry* é modificado com o valor do *bit* perdido.

- ASR Rst, RsA, r0

  `GPR[Rst]` ← $GPR[RsA]_{15/31}$ `&` $GPR[RsA]_{<15/31:1>}$

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

ROR - rotate right trough carry

Realiza a rotação por 1 bit à direita e armazena o resultado em um registrador. O valor inserido no bit mais significativo é o valor de *carry* gerado na última instrução. O novo valor de *carry* é modificado com o valor do *bit* perdido.

- ROR Rst, RsA, r0

  `GPR[Rst]` $\leftarrow$ `Carry &` $GPR[RsA]_{<15/31:1>}$

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

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

  `GPR[Rst]` $\leftarrow$ `SEXT(MEM[GPR[RsB]]`$_{<7:0>}$`)`

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

  `MEM[GPR[RsB]]` $\leftarrow$ `GPR[RsA]`$_{<7:0>}$

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

  `GPR[Rst]` $\leftarrow$ `MEM[GPR[RsB]]`

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

```
MEM[GPR[RsB]] ← GPR[RsA]
```

$I_{<15:12>}$	$I_{<11>}$	$I_{<10:8>}$	$I_{<7:5>}$	$I_{<4:2>}$	$I_{<1:0>}$
Opcode	*Imm*	*Rst*	*RsA*	*RsB*	*Op2*
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 ← GPR[RsB]
```

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

- BEZ Rst, Immediate

```
if (GPR[Rst] == zero) PC ← PC + SEXT(Immediate)
```

$I_{<15:12>}$	$I_{<11>}$	$I_{<10:8>}$	$I_{<7:0>}$
Opcode	*Imm*	*Rst*	*Immediate*
1 1 0 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 ← GPR[RsB]
```

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

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

- BNZ Rst, Immediate

  `if (GPR[Rst] != zero) PC ← PC + SEXT(Immediate)`

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. Além disso, alguns opcodes são reaproveitados para instruções semelhantes, como instruções que fazem ou não o uso do qualificador *carry*.

Instrução	**Descrição**	**Opcode**	**Imm**	**Op2**
AND	Logical product	0 0 0 0	x	x x
OR	Logical sum	0 0 0 1	x	x x
XOR	Logical difference	0 0 1 0	x	x x
SLT	Set if less than	0 0 1 1	x	x x
SLTU	Set if less than (unsigned)	0 1 0 0	x	x x
ADD	Add	0 1 0 1	x	x x
ADC	Add with carry	0 1 0 1	0	0 1
SUB	Subtract	0 1 1 0	x	x x
SBC	Subtract with carry	0 1 1 0	0	0 1
LDR	Load register	1 0 0 0	1	x x
LDC	Load constant	1 0 0 1	1	x x
LSR	Logical shift right	1 0 1 0	0	0 0
ASR	Arithmetic shift right	1 0 1 0	0	0 1
ROR	Rotate right trough carry	1 0 1 0	0	1 0
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 1 0 0	x	x x
BNZ	Branch if not equal zero	1 1 0 1	x	x x

1.5 Detalhes sobre a codificação de instruções

As decisões abaixo foram tomadas durante o projeto do conjunto de instruções, com o principal objetivo de simplificar o projeto de hardware (incluindo a decodificação de instruções) e permitir a implementação de software eficiente. Em particular, o uso do qualificador *carry* simplifica operações aritméticas e comparações que compreendem múltiplas palavras.

1. As instruções mais comuns possuem tipos I e R, expandindo a funcionalidade do conjunto de instruções;

   - AND, OR, XOR, SLT, SLTU, ADD, SUB, BEZ, BNZ - tipo R

- ANDI, ORI, XORI, SLTI, SLTUI, ADDI, SUBI, BEZI, BNZI - tipo I[7]

2. Algumas instruções possuem apenas um único modo de endereçamento;
   - ADC, SBC, LSR, ASR, ROR, LDB, STB, LDW, STW - modo registrador
   - LDR, LDC - modo imediato
3. Algumas instruções compartilham o mesmo *Opcode*. Nesses casos, a desambiguação ocorre pelos campos *Imm* = 0 e *Op2* = 00 para instruções mais comuns ou 01 para operações de computação com *carry*;
   - ADD, ADDI, ADC - 0101
   - SUB, SUBI, SBC - 0110
4. Operações de deslocamento e rotação (LSR, ASR e ROR) são um caso especial, e compartilham o mesmo *opcode*. A desambiguação é feita pelos campos *Imm* = 0 e *Op2* de acordo com a instrução;
5. Todas as operações de carga e armazenamento utilizam os campos *Imm* = 0 e *Op2* = 10. Quatro *Opcodes* são reciclados;
6. Desvios são codificados com *Opcode* >= 1100;
7. *Opcodes* 0111, 1011, 1110 e 1111 estão disponíveis;
8. *Opcodes* onde *Imm* = 0 e *Op2* = 11 estão disponíveis.

1.6 Características únicas

1.6.1 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 ±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 ±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[8], sendo os valores especificados em decimal.

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

[7] A nomenclatura apresentada para essas instruções não existe no processador Viking, sendo seu tipo definido pelo formato da instrução. No entanto, o comportamento dessas instruções é semelhante a casos encontrados em outros processadores, como MIPS por exemplo.

[8] Mais detalhes sobre a ordem de bytes da arquitetura são apresentados na Seção 1.7.

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):

```
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 ± 128.

1.6.2 Extensão de sinal

Para que valores imediados (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 `SEXT(Immediate)` $\leftarrow$ $\texttt{Immediate}_{\texttt{<7>}}$ ... $\texttt{Immediate}_{\texttt{<7:0>}}$. 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.

1.6.3 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.

1.6.4 Soma e subtração com *carry*

A arquitetura define modificadores para as operações de soma (ADD) e subtração (SUB) para a inclusão de um valor de *carry* gerado pela última operação aritmética. Tais operações são definidas pelas instruções ADC (*Add with carry*) e SBC (*Subtract with carry*), e podem ser utilizadas para simplificar o cálculo de valores de maior magnitude que o suportado pela largura natural da arquitetura. Por exemplo, supondo que deseja-se realizar uma soma em uma arquitetura de 16 bits, onde dois valores de 32 bits estão dispostos em quatro registradores (no exemplo, o primeiro operando está armazenado em *r1:r0* (parte alta e baixa, respectivamente) e o segundo operando está em *r3:r2*). A operação pode ser realizada da seguinte forma:

```
add r2,r2,r0
adc r3,r3,r1
```

Na primeira instrução, as partes baixas são somadas e o resultado é armazenado em *r2*, gerando-se também o *carry*. Na segunda instrução, as partes altas são somadas, juntamente com o valor do *carry* da última operação e o resultado é armazenado em *r3*.

É importante observar que todas as instruções da arquitetura modificam o estado do qualificador *carry* mas apenas as instruções de soma (ADD, ADC), subtração (SUB, SBC) e deslocamento ou rotação (LSR, ASR, ROR) fazem uso do qualificador (gerado pela última instrução) e podem definir seu valor em 1, caso ocorra *carry*. Em todos os outros casos, *carry* possui o valor 0 após a execução da instrução.

1.6.5 Deslocamento com *carry*

São definidas três operações de deslocamento na arquitetura, incluindo o deslocamento lógico à direita (LSR), deslocamento aritmético à direita (ASR) e rotação à direita (ROR). Em todos os casos, o bit deslocado define o valor do qualificador *carry* após a execução da instrução. A instrução de rotação utiliza o valor de *carry* gerado na última instrução como o valor incluído em seu bit mais significativo, e modifica o valor de *carry* com o valor do bit deslocado (perdido). A instrução de rotação (juntamente com as instruções de deslocamento) pode ser utilizada para a manipulação de valores maiores que o naturalmente suportado pela arquitetura. Por exemplo, supondo que a arquitetura seja de 16 bits e um valor de 32 bits esteja disposto em dois registradores (*r2:r1*), a operação de deslocamento em 32 bits pode ser realizada da seguinte forma (essencialmente, dividindo o valor de 32 bits por 2 e mantendo o valor sinalizado):

```
asr r2,r2
ror r1,r1
```

As operações de deslocamento e rotação à esquerda são sintetizadas com o uso das instruções ADD e ADC, respectivamente. Seguindo o exemplo anterior, poderia-se realizar a mesma operação, porém com o deslocamento à esquerda, da seguinte forma (essencialmente, multiplicando o valor de 32 bits por 2):

```
add r1,r1
adc r2,r2
```

1.6.6 Comparações com *carry*

Operações de comparação também podem utilizar *carry* com o objetivo de permitir de maneira simplificada a avaliação entre operandos maiores que a largura natural da arquitetura. Uma instrução semelhante a *Set if less than (unsigned)* pode ser sintetizada pelo seguinte código (operandos de 32 bits):

```
xor r0,r0,r0
sub r1,r1,r3
sbc r2,r2,r4
adc r1,r0,r0
```

No exemplo apresentado, se o resultado contido nos registradores *r2:r1* for menor que *r4:r3* (sem sinal), o valor 1 será armazenado em *r1* e 0 caso contrário. Para uma versão que envolve operandos sinalizados em complemento de 2, é necessário apenas inverter o sinal dos dois operandos (bit mais significativo) e realizar o teste como se os números não fossem sinalizados:

```
ldr r0,0x80
ldc r0,0x00
xor r2,r2,r0
xor r4,r4,r0
xor r0,r0,r0
sub r1,r1,r3
sbc r2,r2,r4
adc r1,r0,r0
```

Em ambos os casos não são necessários desvios[9] adicionais (além do desvio posterior ao teste, não mostrado nos exemplos) para a implementação das comparações em comandos de seleção e repetição.

1.6.7 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

[9] Em arquiteturas RISC, muitas vezes não existem qualificadores de operação (como o *carry*), o que complica consideravelmemente a implementação de operações que envolvem precisão maior em função de desvios condicionais. O uso de desvios, além de aumentar o tamanho do código, implica em degradação da execução em função de sua influência no *pipeline* de execução.

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 apresentados no Capítulo 2.

1.7 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 <31:24> para 32 bits ou bits <15:8> para 16 bits) é acessado quando o endereço estiver alinhado (endereço, bits <1:0> = 0 para 32 bits ou bit <0> = 0 para 16 bits) e o byte menos significativo é acessado quando os bits do endereço forem <1:0> = 3 (para palavras de 32 bits) e <0> = 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 <1:0> = 0 para 32 bits ou bit <0> = 0 para 16 bits). Não são definidos acessos desalinhados para esse tipo.

Capítulo 2

Síntese de pseudo operações

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 ADD. Deslocamentos e rotações à esquerda são sintetizados com operações ADD e ADC. A carga de constantes é sintetizada de maneira trivial pelo montador, no entanto uma sequência mais otimizada pode ser gerada, como apresentado na Seção 1.6.1. 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, o registrador *r1* é exemplificado como registrador destino ou fonte da operação, enquanto *r2* é fonte. Em casos onde o endereço necessita ser calculado em função de um rótulo, *at* é fonte.

Instrução	Descrição	Formato	Equivalência
NOP	No operation	`nop`	`and r0,r0,r0`
NOT	One's complement	`not r1`	`xor r1,-1`
NEG	Two's complement	`neg r1`	`xor r1,-1` `add r1,1`
MOV	Move register	`mov r1,r2`	`and r1,r2,r2`
LSR	Logical shift right	`lsr r1,r2`	`lsr r1,r2,r0`
ASR	Arithmetic shift right	`asr r1,r2`	`asr r1,r2,r0`
ROR	Rotate right trough carry	`ror r1,r2`	`ror r1,r2,r0`
LSL	Logical shift left	`lsl r1,r2`	`add r1,r2,r2`
ROL	Rotate left trough carry	`rol r1,r2`	`adc r1,r2,r2`
LDI	Load immediate	`ldi r1,`*`const`*	`ldr r1,`*`byte0`* `ldc r1,`*`byte1`* `...`
BEZ	Branch if equal zero	`bez r1,r2`	`bez r0,r1,r2`
		`bez r1,`*`addr`*	`ldi at,`*`addr`* `bez r0,r1,at`
BNZ	Branch if not equal zero	`bnz r1,r2`	`bnz r0,r1,r2`
		`bnz r1,`*`addr`*	`ldi at,`*`addr`* `bnz r0,r1,at`
LDB	Load byte	`ldb r1,r2`	`ldb r1,r0,r2`
		`ldb r1,`*`addr`*	`ldi at,`*`addr`* `ldb r1,r0,at`
STB	Store byte	`stb r1,r2`	`stb r0,r1,r2`
		`stb r1,`*`addr`*	`ldi at,`*`addr`* `stb r0,r1,at`
LDW	Load word	`ldw r1,r2`	`ldw r1,r0,r2`
		`ldw r1,`*`addr`*	`ldi at,`*`addr`* `ldw r1,r0,at`
STW	Store word	`stw r1,r2`	`stw r0,r1,r2`
		`stw r1,`*`addr`*	`ldi at,`*`addr`* `stw r0,r1,at`
HCF	Halt and catch fire	`hcf`	`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	`lsrm r1,r2`	`bez r2,6`
			`lsr r1,r1,r0`
			`sub r2,1`
			`bnz r2,-6`

ASRM	Arithmetic shift right multiple	`asrm r1,r2`	`bez r2,6`
			`asr r1,r1,r0`
			`sub r2,1`
			`bnz r2,-6`

LSLM	Logical shift left multiple	`lslm r1,r2`	`bez r2,6`
			`add r1,r1,r1`
			`sub r2,1`
			`bnz r2,-6`

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 r1,r2,r3`	`sub r1,r2,r3` `sltu r1,1`

SNE	Set if not equal	`sne r1,r2,r3`	`sub r1,r2,r3`
			`sltu r1,1`
			`xor r1,1`

SGE	Set if greater equal	`sge r1,r2,r3`	`slt r1,r2,r3` `xor r1,1`
SGEU	Set if greater equal (unsigned)	`sgeu r1,r2,r3`	`sltu r1,r2,r3` `xor r1,1`

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 r1,r2,`*`addr`*	`ldi at,`*`addr`*
			`sub r1,r1,r2`
			`bez r0,r1,at`

BNE	Branch if not equal	`bne r1,r2,`*`addr`*	`ldi at,`*`addr`*
			`sub r1,r1,r2`
			`bnz r0,r1,at`

BLT	Branch if less than	`blt r1,r2,`*`addr`*	`ldi at,`*`addr`*
			`slt r1,r1,r2`
			`bnz r0,r1,at`

BGE	Branch if greater equal	`bge r1,r2,`*`addr`*	`ldi at,`*`addr`*
			`slt r1,r1,r2`
			`bez r0,r1,at`

BLTU	Branch if less than (unsigned)	`bltu r1,r2,`*`addr`*	`ldi at,`*`addr`*
			`sltu r1,r1,r2`
			`bnz r0,r1,at`

BGEU	Branch if greater equal (unsigned)	`bgeu r1,r2,`*`addr`*	`ldi at,`*`addr`*
			`sltu r1,r1,r2`
			`bez r0,r1,at`

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.

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

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

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 / *oper*	Equivalência

MUL / DIV / REM	Multiply / Divide / Division remainder	`mul r1,r2,r3` (*`mulsi3`*) / `div r1,r2,r3` (*`divsi3`*) / `rem r1,r2,r3` (*`modsi3`*)	`sub sp,2`
			`stw r0,r2,sp`
			`sub sp,2`
			`stw r0,r3,sp`
			`sub sp,2`
			`stw r0,lr,sp`
			`ldi lr,`*`raddr`*
			`ldi sr,`*`oper`*
			`bnz r0,r7,sr`
			`raddr`
			`ldw lr,r0,sp`
			`add sp,6`
			`and r1,sr,sr`

Capítulo 3

Programando com o processador Viking

Algumas estruturas de controle básicas para a programação do processador são apresentadas nas próximas seções. Nos exemplos apresentados serão usadas apenas instruções suportadas nativamente pela arquitetura e pseudo operações básicas suportadas pelo montador, com o objetivo de ilustrar padrões simples para construção de código.

3.1 Controle de fluxo do programa

As estruturas de controle básicas de linguagem de alto nível como seleção e repetição podem ser implementadas para o controle de fluxo de execução com apenas quatro instruções (SUB, SLT, BNZ e BEZ) na arquitetura Viking.

3.1.1 Seleção

Igual a (==) e diferente de (!=)

Em um comando de seleção que utiliza uma comparação por igualdade (*if (a == b)*), são utilizadas instruções SUB e BEZ. A idéia é que se dois valores forem iguais (nesse caso, as variáveis a e b estão armazenadas nos registradores *r1* e *r2* respectivamente) a subtração de ambos resultará em zero, e o desvio do fluxo de controle (condicional) será executado (*if*).

```
    sub r3,r1,r2
    bez r3,if
else
    ...
if
    ...
```

Quando a comparação for por não igualdade (*if (a != b)*), utiliza-se uma instrução BNZ. Neste caso, sempre que o resultado da subtração for diferente de zero (ou seja, se os valores de a e b forem diferentes) o desvio será executado.

```
    sub r3,r1,r2
    bnz r3,if
else
    ...
if
    ...
```

Menor que (<) e maior ou igual a (>=)

Para a implementação de seleção para uma comparação por menor que (*if (a < b)*) as instruções SLT (ou SLTU, caso os valores a serem comparados não forem sinalizados) e BNZ são utilizadas. Se o valor de *r1* for menor que *r2*, o resultado da comparação será diferente de zero, e o salto será executado.

```
    slt r3,r1,r2
    bnz r3,if
else
    ...
if
    ...
```

Em uma comparação por maior ou igual a (*if (a >= b)*) a lógica é a mesma, porém utiliza-se uma instrução BEZ. Deve-se lembrar que se um número não for menor que outro (<) ele é maior ou igual ao outro número (>=), então a única diferença entre as duas comparações deve ser a instrução de salto.

```
    slt r3,r1,r2
    bez r3,if
else
    ...
if
    ...
```

Maior que (>) e menor ou igual a (<=)

Para a implementação de seleção para uma comparação por maior que (*if (a > b)*) as instruções SLT (ou SLTU, caso os valores a serem comparados não forem sinalizados) e BNZ são utilizadas. Se o valor de *r2* for menor que *r1*, (ou seja, *r1* for maior que *r2*) o resultado da comparação será diferente de zero, e o salto será executado.

```
    slt r3,r2,r1
    bnz r3,if
else
    ...
if
    ...
```

Em uma comparação por menor ou igual a (*if (a <= b)*) a lógica é a mesma, porém utiliza-se uma instrução BEZ. Deve-se lembrar que se um número não for maior que outro (>) ele é menor

ou igual ao outro número (<=), então a única diferença entre as duas comparações deve ser a instrução de salto.

```
    slt r3,r2,r1
    bez r3,if
else
    ...
if
    ...
```

Alternativas para menor ou igual a (<=) e maior ou igual a (>=)

Versões alternativas para as operações de seleção maior ou igual a (>=) e menor ou igual a (<=) podem ser usadas. Essas versões utilizam mais instruções, porém são mais simples de serem verificadas mentalmente. Nessas versões, os testes são realizados de forma independente - primeiramente o teste por menor que (<) é realizado (usando-se SLT e BNZ) pois cobre a maior parte dos casos, e posteriormente o teste por igualdade (==) é realizado (usando-se SUB e BEZ). A idéia é que qualquer uma das condições possa fazer com que o fluxo de execução seja desviado.

O exemplo abaixo realiza o teste para menor ou igual a (*if (a <= b)*). Para o teste de maior ou igual a (*if (a >= b)*, basta inverter a ordem de *r1* e *r2* na primeira instrução (SLT).

```
    slt r3,r1,r2
    bnz r3,if
    sub r3,r1,r2
    bez r3,if
else
    ...
if
    ...
```

3.1.2 Repetição

Estruturas de controle de repetição em linguagem de montagem possuem uma estrutura semelhante à estruturas de seleção, com a diferença de que normalmente o fluxo de execução será redirecionado a um ponto do código percorrido anteriormente de maneira iterativa. Além da operação de repetição (como um *for*, *while* ou *do .. while*), muitas vezes são utilizadas comandos do tipo *break* (que quebra o laço incondicionalmente) e *continue* (que desvia incondicionalmente para a próxima iteração do laço). Em todos os casos, são utilizadas estruturas semelhantes às apresentadas anteriormente.

Repetição incondicional

Um comando simples de repetição incondicional pode ser implementado de acordo com o padrão a seguir. Nesse exemplo, assume-se que o registrador *r7* (ou *sp*) nunca tenha um valor zero. Esse exemplo ilustra uma construção semelhante ao um laço *while (1) { ... }*.

```
while
    ...
```

```
    bnz r7,while
endwhile
```

Repetição condicional

A implementação de um comando de repetição semelhante a *while (a < b) { ... }* é mostrado a seguir. Nesse exemplo, as variáveis *a* e *b* estão armazenadas nos registradores *r1* e *r2* respectivamente.

```
while
    slt r3,r1,r2
    bez r3,endwhile
    ...
    bnz r7,while
endwhile
```

Importante observar no exemplo anterior que se a comparação $a < b$ for falsa (ou seja, zero), o fluxo de execução será desviado para o final do laço. Enquanto $a < b$, o primeiro desvio não será tomado, o corpo da repetição será executado e o último comando de desvio (incondicional) irá desviar o fluxo de execução para o início do laço.

Para a implementação de um comando de repetição do tipo *do ... while (a < b)* basta que o teste seja realizado no final do laço. Nesse exemplo, se a comparação $a < b$ for verdadeira, o fluxo de execução será desviado para o início do laço.

```
while
    ...
    slt r3,r1,r2
    bnz r3,while
endwhile
```

3.2 Acesso à memória - variáveis

Apenas um número limitado de registradores está presente na arquitetura Viking. Parte desses registradores são usados para fins específicos (como apresentado na Seção 3.4.2), restando na maior parte dos casos apenas os registradores *r1* a *r5* como temporários para o armazenamento de variáveis.

A arquitetura realiza o acesso à memória de dados apenas com instruções carga e armazenamento (*load / store*). Dessa forma, os operandos precisam serem trazidos da memória, uma vez que as operações lógicas e aritméticas são realizadas apenas nos registradores internos. Por exemplo, para realizar uma operação de soma entre duas variáveis e armazenar o resultado em uma terceira variável (C = A + B), é necessário um padrão semelhante ao apresentado abaixo, que realiza 3 acessos à memória de dados.

```
    ldw r1,A
    ldw r2,B
    add r3,r1,r2
    stw r3,C
```

```
    ...
A   123
B   333
C   0
```

Caso uma variável seja utilizada frequentemente (um contador em um laço, por exemplo), pode-se fixar temporariamente o uso de um dos registradores para evitar operações de acesso à memória indesejáveis (carga da variável contador, incremento do contador e armazenamento da variável contador).

3.3 Acesso à memória - vetores

O acesso à vetores pode ser realizado com o uso de ponteiros. Um ponteiro nada mais é que uma variável (valor inteiro) que armazena um endereço de memória. Dessa forma, o ponteiro é utilizado para referenciar uma posição de memória. Esse endereço pode ser qualquer posição na memória, então um ponteiro pode referenciar o conteúdo de uma variável, ou elemento de um vetor.

Um detalhe importante para acesso à memória utilizando o conceito de ponteiros é que é necessário que o tipo de dados apontado seja conhecido. Por exemplo, na arquitetura Viking inteiros possuem 2 ou 4 bytes (2 em uma arquitetura de 16 bits, como no exemplo abaixo) e vetores de caracteres (*strings*) possuem 1 byte por elemento. Precisamos levar isso em conta para calcular o deslocamento na memória durante o acesso à vetores.

Para o cálculo do deslocamento, usa-se a fórmula $d = i*ts$, onde d é o deslocamento, i é o índice do vetor e ts é o tamanho do tipo de dado armazenado no vetor. Sabendo-se o deslocamento, é possível encontrar o endereço de memória efetivo de um determinado elemento em um índice i de um vetor. Esse elemento i pode ser acessado por um ponteiro que contém o endereço do primeiro elemento do vetor somado ao deslocamento (formando um endereço efetivo).

No exemplo abaixo, o quarto elemento de um vetor será acessado e nele armazenado o valor 123 (*vetor[3] = 123*. Assume-se que *vetor* possui 5 elementos do tipo inteiro de 16 bits.

```
    ldi r4,vet
    add r4,3
    add r4,3
    ldi r3,123
    stw r3,r4
    ...
vet 0 0 0 0 0
```

No código acima, o endereço do primeiro elemento do vetor é carregado em *r4* com a instrução LDI. O endereço efetivo é calculado somando-se o índice (nesse caso 3) duas vezes (ou seja, 3 multiplicado por 2 em função do tipo inteiro) ao endereço inicial. O acesso ao vetor é realizado pela instrução STW, que armazena o valor de *r3* no endereço efetivo armazenado em *r4*.

Para se realizar o acesso à um vetor de caracteres, o índice não precisa ser multiplicado pelo tamanho do tipo. Além disso usam-se instruções LDB e STB para a leitura e escrita. No exemplo abaixo o elemento *vet[10]* (um espaço em branco) é substituído por uma quebra de linha ('\n').

```
    ldi r4,vet
    add r4,10
    ldi r3,0xa
    stb r3,r4
    ...
vet "fight fire with fire"
```

3.4 Chamadas de função e convenções de chamada

3.4.1 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:

```
    sub sp,2          # PUSH r1
    stw r1,sp
    ...
    ldw r1,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.

3.4.2 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	Ponteiro de pilha	Sim

Nos formatos de instruções em que um dos registradores especificado é fixo, deve-se utilizar a notação *r0*. Pseudo operações podem ser usadas nesse caso para que a referência a *r0* seja omitida, uma vez que essa referência trata-se de um detalhe da arquitetura que não precisa ser exposto ao programador. 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.

3.4.3 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 ponteiro 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 4

Montagem de código e simulação

4.1 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.

4.1.1 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 seguido por um espaço (;) à esquerda, sem tabulações. Apenas caracteres da língua inglesa são reconhecidos.
- 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 ou alvo de desvios) no código, e com parâmetros definem estruturas de dados e sua posição inicial 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 numérico (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 $\backslash t$, $\backslash n$ e $\backslash r$. *Strings* definem implicitamente o terminador $\backslash 0$.
- 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.

4.1.2 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 4.1: 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
```

```
writec  0xf000
writei  0xf002
```

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

Listing 4.2: ninetoone.out

```
0000 9900
0002 9909
0004 9a00
0006 9a20
0008 9800
000a 9824
000c 4502
000e 5036
0010 9800
0012 9822
0014 4502
0016 5056
0018 6901
001a 9800
001c 9808
001e b020
0020 0003
0022 f000
0024 f002
```

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 4.3: ninetoone_debug.out

```
main
0000 9900      ldc0  r1,9
0002 9909      ldc1  r1,9
0004 9a00      ldc0  r2,32
0006 9a20      ldc1  r2,32
loop
0008 9800      ldc0  at,writei
000a 9824      ldc1  at,writei
000c 4502      ldw  sr,r0,at
000e 5036      stw  r0,r1,sr
0010 9800      ldc0  at,writec
0012 9822      ldc1  at,writec
0014 4502      ldw  sr,r0,at
0016 5056      stw  r0,r2,sr
0018 6901      sub  r1,1
001a 9800      ldc0  at,loop
001c 9808      ldc1  at,loop
001e b020      bnz  r0,r1,at
0020 0003      hcf  r0,r0,r0

0022 f000 writec  0xf000
0024 f002 writei  0xf002
```

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
```

4.2 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. Esse 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).

4.2.1 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`
Ponteiro de pilha	`0xdffe`	`0x000ffffc`
Saída (caracter)	`0xf000`	`0xf0000000`
Saída (inteiro)	`0xf002`	`0xf0000004`
Entrada (caracter)	`0xf004`	`0xf0000008`
Entrada (inteiro)	`0xf006`	`0xf000000c`

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

4.2.2 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

```
main
  ldw sr,writec
  ldi r4,str
  ldi r3,loop
loop
  ldb r2,r4
  stw r2,sr
  add r4,1
  bnz r2,r3
  hcf

writec  0xf000
str "hello world!"
```

Listing A.2: fibonacci.asm

```
main
  xor r1,r1,r1
  ldi r2,1
  ldi r4,21
fib_loop
  ldw sr,writei
  stw r1,sr
  ldw sr,writec
  ldi r3,32
  stw r3,sr

  add r3,r1,r2
  and r1,r2,r2
  and r2,r3,r3

  sub r4,1
  bnz r4,fib_loop
  hcf

writec  0xf000
writei  0xf002
```

Listing A.3: function_call.asm

```
main
  ldi r1,str1
  sub sp,2
  stw r1,sp
  sub sp,2
  stw lr,sp
  ldi lr,ret_print1
  ldi sr,print_str
  bnz r7,sr
ret_print1
  ldw lr,sp
  add sp,4

  ldi r1,str2
  sub sp,2
  stw r1,sp
  sub sp,2
  stw lr,sp
  ldi lr,ret_print2
  ldi sr,print_str
  bnz r7,sr
ret_print2
  ldw lr,sp
  add sp,4

  hcf

print_str
  ldw sr,writec
  sub sp,2
  stw r1,sp
  sub sp,2
  stw r2,sp

  and r1,sp,sp
  add r1,6
  ldw r1,r1
```

```
print_loop
  ldb r2,r1
  stw r2,sr
  add r1,1
  bnz r2,-8

  ldw r2,sp
  add sp,2
  ldw r1,sp
  add sp,2
  bnz r7,lr


writec  0xf000
str1  "this is the first call\n"
str2  "and this is the second!\n"
```

Listing A.4: mult.asm

```
main
  ldw r2,readi
  ldw r2,r2
  ldw r3,readi
  ldw r3,r3

  sub sp,2
  stw r2,sp
  sub sp,2
  stw r3,sp
  sub sp,2
  stw lr,sp
  ldi lr,ret_addr
  ldi sr,mulsi3
  bnz r7,sr
ret_addr
  ldw lr,sp
  add sp,6

  and r1,sr,sr
  ldw sr,writei
  stw r1,sr
  hcf

writei  0xf002
readi 0xf006
```

Listing A.5: bubble_sort.asm

```
main
  ldi lr,ret_print1
  bnz r7,print_numbers
ret_print1
  ldi lr,ret_sort
  bnz r7,sort
ret_sort
  ldi lr,ret_print2
  bnz r7,print_numbers
ret_print2
  hcf

print_numbers
  ldi r1,0
  ldw r2,N
  ldi r3,numbers
loop_print
  ldw r4,r3
  stw r4,0xf002
  ldi r4,32
  stw r4,0xf000
  add r1,1
  add r3,2
  sub r5,r1,r2
  bnz r5,loop_print
  ldi r4,10
  stw r4,0xf000
  bnz r7,lr

sort
  ldw r1,i
loop_i
  ldw r3,N
  sub r3,1
  slt r3,r1,r3
  bez r3,end_i

  xor r0,r0,r0
  add r2,r1,r0
  add r2,1
  stw r2,j
loop_j
  ldw r1,i
  ldw r3,N
  slt r3,r2,r3
  bez r3,end_j

  ldi r4,numbers
  add r3,r4,r1
  add r3,r3,r1
  ldw r1,r3
  add r4,r4,r2
  add r4,r4,r2
  ldw r2,r4

  slt r1,r2,r1
  bez r1,skip

  ldw r1,r3
  stw r1,r4
  stw r2,r3
skip
  ldw r2,j
```

```
64    add  r2,1
65    stw  r2,j
66    bnz  r7,loop_j
67  end_j
68    ldw  r1,i
69    add  r1,1
70    stw  r1,i
71    bnz  r7,loop_i
72  end_i
73    bnz  r7,lr
74
75  i  0
76  j  0
77  N  10
78  numbers -5 8 -22 123 77 -1 99 -33 10 12
```

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
```

```
    ldi  lr,ret_divsi3
    ldi  sr,udivmodsi4
    bnz  r7,sr
ret_divsi3
    ldw  lr,sp
    add  sp,8
    bez  r3,4
    xor  at,at,at
    sub  sr,at,sr
```

Listing B.3: modsi3.asm

```
modsi3
    sub  sp,2
    stw  r1,sp
    sub  sp,2
    stw  r2,sp
    sub  sp,2
    stw  r3,sp

    and  r2,sp,sp
    add  r2,10
    ldw  r1,r2
    sub  r2,2
    ldw  r2,r2
    xor  r3,r3,r3

    xor  at,at,at
    slt  sr,r1,at
    bez  sr,4
    sub  r1,at,r1
    or   r3,1
    slt  sr,r2,at
    bez  sr,4
    sub  r2,at,r2
    xor  r3,1

    sub  sp,2
    stw  r1,sp
    sub  sp,2
    stw  r2,sp
    sub  sp,2
    ldr  sr,1
    stw  sr,sp
    sub  sp,2
    stw  lr,sp
    ldi  lr,ret_modsi3
    ldi  sr,udivmodsi4
    bnz  r7,sr
ret_modsi3
    ldw  lr,sp
    add  sp,8
    bez  r3,4
    xor  at,at,at
    sub  sr,at,sr

    add  sp,2
    ldw  r3,sp
    add  sp,2
    ldw  r2,sp
    add  sp,2
    ldw  r1,sp

    bnz  r7,lr
```

Listing B.4: udivmodsi4.asm

```
udivmodsi4
    sub  sp,2
    stw  r1,sp
    sub  sp,2
    stw  r2,sp
    sub  sp,2
    stw  r3,sp
    sub  sp,2
    stw  r4,sp

    ldr  r3,1
    xor  r4,r4,r4

    and  r2,sp,sp
    add  r2,14
    ldw  r1,r2
    sub  r2,2
    ldw  r2,r2

    sltu   sr,r2,r1
    bez  sr,8
    bez  r3,6
    lsl  r2,r2
    lsl  r3,r3
    bnz  r7,-12
    sltu   sr,r1,r2
    bnz  sr,4
    sub  r1,r1,r2
    add  r4,r4,r3
    lsr  r3,r3
    lsr  r2,r2
    bnz  r3,-14

    and  sr,sp,sp
    add  sr,10

    add  sp,2
    ldw  r3,sp
    add  sp,2
    ldw  r2,sp
    add  sp,2
    ldw  r1,sp

    bnz  r7,lr
```

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