Viking CPU - Manual de referência v0.2

Sérgio Johann Filho

29 de setembro de 2017

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 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

2 Síntese de pseudo instruções 17
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

3 Montagem de código e simulação 21
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

A Rotinas *mulsi3*, *divsi3*, *modsi3* e *udivmodsi4* 25

B Exemplos 28

C Montador - código fonte 31

D Simulador - código fonte 37

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 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.

[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.

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.

$I_{<15:12>}$	$I_{<11>}$	$I_{<10:8>}$	$I_{<7:5>}$	$I_{<4:2>}$	$I_{<1:0>}$
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 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*[2].

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:

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

[2] 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.

[3] A implementação de extensão de sinal é apresentada na Seção 1.5.2.

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

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[5] 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

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

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

$I_{<15:12>}$	$I_{<11>}$	$I_{<10:8>}$	$I_{<7:5>}$	$I_{<4:2>}$	$I_{<1:0>}$
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

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

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, o que permite a carga de constantes de valores maiores que ±128 com múltiplas instruções.

- LDC Rst, Immediate

```
GPR[Rst] ← (GPR[Rst] << 8) + 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

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

```
GPR[Rst] ← GPR[RsA] >> 1
```

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

```
GPR[Rst] ← GPR[RsA] >> 1
```

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

  `GPR[Rst] ← 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]] ← 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] ← 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 0 1 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 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 ← 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 0 1 1	0	0 0 0	r r r	r r r	1 1

- BNZ 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 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 ±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[6], 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):

[6]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 ± 128.

1.5.2 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.

1.5.3 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 <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.

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	`nop`	`and r0,r0,r0`
NOT	One's complement	`not r1`	`xor r1,-1`
NEG	Two's complement	`neg r1`	`xor at,at,at` `sub r1,at,r1`
LSR	Logical shift right	`lsr r1,r2`	`lsr r1,r2,r0`
ASR	Arithmetic shift right	`asr r1,r2`	`asr r1,r2,r0`
LSL	Logical shift left	`lsl r1,r2`	`add r1,r2,r2`
LDI	Load immediate	`ldi r1,`*`const`*	`ldc 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`	`lsr r1,r1,r0`
			`sub r2,1`
			`bnz r2,-6`

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

LSLM	Logical shift left multiple	`lslm r1,r2`	`add r1,r1,r0`
			`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`
			`xor at,at,at`
			`sltu r1,at,r1`

SGE	Set if greater equal	`sge r1,r2,r3`	`slt r1,r2,r3`
			`ldr at,1`
			`sub r1,at,r1`

SGEU	Set if greater equal (unsigned)	`sgeu r1,r2,r3`	`sltu r1,r2,r3`
			`ldr at,1`
			`sub r1,at,r1`

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.

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`

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	`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 A.

Instrução	Descrição	Formato	Equivalência

MUL / DIV / REM	Multiply / Divide / Division remainder	`mul r1,r2,r3 / div r1,r2,r3 / rem r1,r2,r3`	`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,`*`mulsi3 / divsi3 / modsi3`*
			`bnz r0,r7,sr`
			`ldw lr,r0,sp`
			`add sp,6`
			`add r1,r0,sr`

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 $\backslash t$, $\backslash n$ e $\backslash r$.
- 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

```
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 3.3: 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
```

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. 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).

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 (caracter)	`0xf000`	`0xf0000000`
Saída (inteiro)	`0xf002`	`0xf0000004`
Entrada (caracter)	`0xf004`	`0xf0000008`
Entrada (inteiro)	`0xf006`	`0xf000000c`

No início da simulação, o apontador 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.

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

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

Listing A.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 A.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 A.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

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

    bnz  r7,lr
```

Listing A.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

    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
```

```
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
```

Apêndice B

Exemplos

Listing B.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 B.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 B.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 B.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 B.5: bubble_sort.asm

```
main
  ldi sr,array1
  sub sp,2
  stw sr,sp
  ldw sr,array1_sz
  lsl sr,sr
  sub sp,2
  stw sr,sp
  sub sp,2
  stw lr,sp
  ldi lr,ret_sort1
  ldi sr,sort
  bnz r7,sr
ret_sort1
  ldw lr,sp
  add sp,4

  ldi r1,array1
  ldw r2,array1_sz
print_array1
  ldw r3,r1
  ldw r4,writei
  stw r3,r4
  ldi r3,32
  ldw r4,writec
  stw r3,r4
  add r1,2
  sub r2,1
  bnz r2,print_array1
  ldi r3,10
  stw r3,r4


  ldi sr,array2
  sub sp,2
  stw sr,sp
  ldw sr,array2_sz
  lsl sr,sr
  sub sp,2
  stw sr,sp
  sub sp,2
  stw lr,sp
  ldi lr,ret_sort2
  ldi sr,sort
  bnz r7,sr
ret_sort2
  ldw lr,sp
  add sp,4

  ldi r1,array2
  ldw r2,array2_sz
print_array2
  ldw r3,r1
  ldw r4,writei
  stw r3,r4
  ldi r3,32
  ldw r4,writec
  stw r3,r4
  add r1,2
  sub r2,1
  bnz r2,print_array2
  ldi r3,10
  stw r3,r4

```

```
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 C

Montador - código fonte

Listing C.1: assemble16.py

```
#!/usr/bin/python

import sys, string

codes = {
  "and":0x0000, "or":0x1000, "xor":0x2000, "slt":0x3000,
  "sltu":0x4000, "add":0x5000, "sub":0x6000, "ldr":0x8000,
  "ldc":0x9000, "lsr":0x0001, "asr": 0x1001,
  "ldb":0x0002, "stb":0x1002, "ldw":0x4002, "stw":0x5002,
  "bez":0xa000, "bnz":0xb000,
  "ldc0":0x9000, "ldc1":0x9000, "hcf":0x0003        # special ops
}

lookup = {
  "r0":0, "r1":1, "r2":2, "r3":3,
  "r4":4, "r5":5, "r6":6, "r7":7,
  "at":0, "sr":5, "lr":6, "sp":7
}

def is_number(s):
  try:
    int(s)
    return True
  except ValueError:
    return False

def tohex(n):
  return "%s" % ("0000%x" % (n & 0xffff))[-4:]

def getval(s) :
  "return numeric value of a symbol or number"
  if not s : return 0             # empty symbol - zero
  a = lookup.get(s)             # get value or None if not in lookup
  if a == None : return int(s, 0)           # just a number (prefix can be 0x.. 0o
    .. 0b..)
  else : return a

def pass1(program) :
```

```
38    "process pseudo operations"
39    i = 0
40    for lin in program :
41      flds = string.split(lin)
42      if flds :
43        if flds[0] == ";" :
44          program[i] = '\n'
45        if flds[0] == "nop" :
46          program[i] = "\tand r0,r0,r0\n"
47        if flds[0] == "hcf" :
48          program[i] = "\thcf r0,r0,r0\n"
49        if len(flds) > 1 :
50          parts  = string.split(flds[1],",")
51          if flds[0] == "not" :
52            program[i] = "\txor " + parts[0] + ",-1\n"
53          if flds[0] == "neg" :
54            program[i] = "\txor at,at,at\n"
55            program[i].insert(i+1, "\tsub " + parts[0] + ",at," + parts[0] + "\n")
56          if flds[0] == "lsr" :
57            program[i] = "\tlsr " + parts[0] + "," + parts[1] + "," + "r0\n"
58          if flds[0] == "asr" :
59            program[i] = "\tasr " + parts[0] + "," + parts[1] + "," + "r0\n"
60          if flds[0] == "lsl" :
61            program[i] = "\tadd " + parts[0] + "," + parts[1] + "," + parts[1] + "\n
      "
62          if flds[0] == "ldi" :
63            program[i] = "\tldc0  " + flds[1] + "\n";
64            program.insert(i+1, "\tldc1 " + flds[1] + "\n");
65          if flds[0] == "ldb" and len(parts) == 2 :
66            if lookup.get(parts[1]) == None :
67              program[i] = "\tldc0  at," + parts[1] + "\n"
68              program.insert(i+1, "\tldc1 at," + parts[1] + "\n");
69              program.insert(i+2, "\tldb  " + parts[0] + ",r0,at\n");
70            else :
71              program[i] = "\tldb " + parts[0] + ",r0," + parts[1] + "\n"
72          if flds[0] == "stb" and len(parts) == 2 :
73            if lookup.get(parts[1]) == None :
74              program[i] = "\tldc0  at," + parts[1] + "\n"
75              program.insert(i+1, "\tldc1 at," + parts[1] + "\n");
76              program.insert(i+2, "\tstb  r0," + parts[0] + ",at\n");
77            else :
78              program[i] = "\tstb r0," + parts[0] + "," + parts[1] + "\n"
79          if flds[0] == "ldw" and len(parts) == 2 :
80            if lookup.get(parts[1]) == None :
81              program[i] = "\tldc0  at," + parts[1] + "\n"
82              program.insert(i+1, "\tldc1 at," + parts[1] + "\n");
83              program.insert(i+2, "\tldw  " + parts[0] + ",r0,at\n");
84            else :
85              program[i] = "\tldw " + parts[0] + ",r0," + parts[1] + "\n"
86          if flds[0] == "stw" and len(parts) == 2 :
87            if lookup.get(parts[1]) == None :
88              program[i] = "\tldc0  at," + parts[1] + "\n"
89              program.insert(i+1, "\tldc1 at," + parts[1] + "\n");
90              program.insert(i+2, "\tstw  r0," + parts[0] + ",at\n");
91            else :
92              program[i] = "\tstw r0," + parts[0] + "," + parts[1] + "\n"
```

```
93          if flds[0] == "bez" and len(parts) == 2 :
94            if lookup.get(parts[1]) == None :
95              if is_number(parts[1]) == False :
96                program[i] = "\tldc0  at," + parts[1] + "\n"
97                program.insert(i+1, "\tldc1 at," + parts[1] + "\n");
98                program.insert(i+2, "\tbez  r0," + parts[0] + ",at\n");
99            else :
100             program[i] = "\nbez r0," + parts[0] + "," + parts[1] + "\n"
101         if flds[0] == "bnz" and len(parts) == 2 :
102           if lookup.get(parts[1]) == None :
103             if is_number(parts[1]) == False :
104               program[i] = "\tldc0  at," + parts[1] + "\n"
105               program.insert(i+1, "\tldc1 at," + parts[1] + "\n");
106               program.insert(i+2, "\tbnz  r0," + parts[0] + ",at\n");
107           else :
108             program[i] = "\nbnz r0," + parts[0] + "," + parts[1] + "\n"
109         if flds[0] == "lsrm" and len(parts) == 2 and is_number(parts[1]) == False
    :
110           program[i] = "\tlsr " + parts[0] + "," + parts[0] + ",r0\n"
111           program.insert(i+1, "\tsub  " + parts[1] + ",1\n")
112           program.insert(i+2, "\tbnz  " + parts[1] + ",-6\n")
113         if flds[0] == "asrm" and len(parts) == 2 and is_number(parts[1]) == False
    :
114           program[i] = "\tasr " + parts[0] + "," + parts[0] + ",r0\n"
115           program.insert(i+1, "\tsub  " + parts[1] + ",1\n")
116           program.insert(i+2, "\tbnz  " + parts[1] + ",-6\n")
117         if flds[0] == "lslm" and len(parts) == 2 and is_number(parts[1]) == False
    :
118           program[i] = "\tadd " + parts[0] + "," + parts[0] + "," + + parts[0] + "
    \n"
119           program.insert(i+1, "\tsub  " + parts[1] + ",1\n")
120           program.insert(i+2, "\tbnz  " + parts[1] + ",-6\n")
121     i += 1
122
123 def pass2(program) :
124   "determine addresses for labels and add to the lookup dictionary"
125   global lookup
126   pc = 0
127   for lin in program :
128     flds = string.split(lin)
129     if not flds : continue            # just an empty line
130     if lin[0] > ' ' :
131       symb = flds[0]              # a symbol - save its address in lookup
132       lookup[symb] = pc
133       flds2 = ' '.join(flds[1:])
134       if flds2 :
135         if flds2[0] == '"' and flds2[-1] == '"' :
136           flds2 = flds2[1:-1]
137           flds2 = flds2.replace("\\t", chr(0x09))
138           flds2 = flds2.replace("\\n", chr(0x0a))
139           flds2 = flds2.replace("\\r", chr(0x0d))
140           flds2 = flds2 + ' '
141           while (len(flds2) % 2) != 0 :
142             flds2 = flds2 + ' '
143         pc = pc + len(flds2)
144       else:
```

```
145             flds = flds[1:]
146             for f in flds :
147               pc = pc + 2
148       else :
149         pc = pc + 2
150
151 def assemble(flds) :
152   "assemble instruction to machine code"
153   opval = codes.get(flds[0])
154   symb = lookup.get(flds[0])
155   if symb != None :
156     return symb
157   else :
158     if opval == None : return int(flds[0], 0)     # just a number (prefix can be 0
      x.. 0o.. 0b..)
159     parts  = string.split(flds[1],",")        # break opcode fields
160     if len(parts) == 2 :
161       parts = [0,parts[0],parts[1]]
162       if (flds[0] == "ldc0") :        # ldc0 .. ldc1 are special steps of ldc
163         return (opval | 0x0800 | (getval(parts[1]) << 8) | ((getval(parts[2]) >>
      8) & 0xff))
164       else :
165         return (opval | 0x0800 | (getval(parts[1]) << 8) | (getval(parts[2]) & 0
      xff))
166     if len(parts) == 3 :
167       parts = [0,parts[0],parts[1],parts[2]]
168       return (opval | (getval(parts[1]) << 8) | (getval(parts[2]) << 5) | (getval(
      parts[3]) << 2))
169
170 def pass3(program) :
171   "translate assembly code and symbols to machine code"
172   args = sys.argv[1:]
173   if args :
174     args = args[0]
175   else :
176     args = ''
177
178   pc = 0
179
180   if args == "debug" :
181     for lin in program :
182       flds = string.split(lin)
183       if lin[0] > ' ' : flds = flds[1:]     # drop symbol if there is one
184       if not flds : print (lin),        # print now if only a symbol
185       else :
186         try :
187           flds2 = ' '.join(flds)
188           if flds2[0] == '"' and flds2[-1] == '"' :
189             flds2 = flds2[1:-1]
190             flds2 = flds2.replace("\\t", chr(0x09))
191             flds2 = flds2.replace("\\n", chr(0x0a))
192             flds2 = flds2.replace("\\r", chr(0x0d))
193             flds2 = flds2 + '\0'
194             while (len(flds2) % 2) != 0 :
195               flds2 = flds2 + '\0'
196             flds3 = ''
```

```
197             while True :
198               flds3 += (str((int(ord(flds2[0])) << 8) | int(ord(flds2[1])))) + ' '
199               flds2 = flds2[2:]
200               if flds2 == '' : break
201             flds3 = string.split(flds3)
202             instruction = assemble(flds3)
203             print ("%04x %s    %s" % (pc, tohex(instruction), lin)),
204             pc = pc + 2
205             flds3 = flds3[1:]
206             for f in flds3 :
207               instruction = assemble(flds3)
208               print ("%04x %s" % (pc, tohex(instruction)))
209               pc = pc + 2
210               flds3 = flds3[1:]
211             flds = ''
212           else :
213             if codes.get(flds[0]) == None :
214               data = assemble(flds)
215               print ("%04x %s %s" % (pc, tohex(data), lin)),
216               pc = pc + 2
217               flds = flds[1:]
218               for f in flds :
219                 data = assemble(flds)
220                 print ("%04x %s" % (pc, tohex(data)))
221                 pc = pc + 2
222                 flds = flds[1:]
223             else :
224               instruction = assemble(flds)
225               print ("%04x %s    %s" % (pc, tohex(instruction), lin)),
226               pc = pc + 2
227         except :
228           print ("**** ????    %s" % lin),
229   else :
230     for lin in program :
231       flds = string.split(lin)
232       if lin[0] > ' ' : flds = flds[1:]     # drop symbol if there is one
233       if not flds : continue
234       try :
235         flds2 = ' '.join(flds)
236         if flds2[0] == '"' and flds2[-1] == '"' :
237           flds2 = flds2[1:-1]
238           flds2 = flds2.replace("\\t", chr(0x09))
239           flds2 = flds2.replace("\\n", chr(0x0a))
240           flds2 = flds2.replace("\\r", chr(0x0d))
241           flds2 = flds2 + '\0'
242           while (len(flds2) % 2) != 0 :
243             flds2 = flds2 + '\0'
244           flds3 = ''
245           while True :
246             flds3 += (str((int(ord(flds2[0])) << 8) | int(ord(flds2[1])))) + ' '
247             flds2 = flds2[2:]
248             if flds2 == '' : break
249           flds3 = string.split(flds3)
250           instruction = assemble(flds3)
251           print ("%04x %s" % (pc, tohex(instruction)))
252           pc = pc + 2
```

```
253             flds3 = flds3[1:]
254             for f in flds3 :
255               instruction = assemble(flds3)
256               print ("%04x %s" % (pc, tohex(instruction)))
257               pc = pc + 2
258               flds3 = flds3[1:]
259             flds = ''
260           else :
261             if codes.get(flds[0]) == None :
262               data = assemble(flds)
263               print ("%04x %s" % (pc, tohex(data)))
264               pc = pc + 2
265               flds = flds[1:]
266               for f in flds :
267                 data = assemble(flds)
268                 print ("%04x %s" % (pc, tohex(data)))
269                 pc = pc + 2
270                 flds = flds[1:]
271             else :
272               instruction = assemble(flds)
273               print ("%04x %s" % (pc, tohex(instruction)))
274               pc = pc + 2
275         except :
276           print ("**** ????    %s" % lin),
277
278 def main() :
279   program = sys.stdin.readlines()
280   pass1(program)
281   pass2(program)
282   pass3(program)
283
284 if __name__ == "__main__" : main()
```

Apêndice D

Simulador - código fonte

Listing D.1: run16.py

```
#!/usr/bin/python

import sys, string

context = [
  0x0000, 0x0000, 0x0000, 0x0000,   # r0 - r3
  0x0000, 0x0000, 0x0000, 0x0000,   # r4 - r7
  0x0000, 0x0000          # pc, stack limit
]

memory = []

def tohex(n):
  return "%s" % ("0000%x" % (n & 0xffff))[-4:]

def check(program) :
  for lin in program :
    flds = string.split(lin)
    if len(flds) != 2 :
      return 1
    for f in flds :
      if f == '****' :
        return 1
  return 0

def load(program) :
  lines = 0
  # load program into memory
  for lin in program :
    flds = string.split(lin)
    data = int(flds[1], 16)
    memory.append(data)
    lines += 1
  print ("[program (code + data): %d bytes]" % (len(memory) * 2))

  # set the stack limit to the end of program section
  context[9] = (len(memory) * 2) + 2
  # fill the rest of memory with zeroes
```

```
39    for i in range(lines, 28672) :
40      memory.append(0)
41    # set the stack pointer to the last memory position
42    context[7] = len(memory) * 2 - 2
43    print ("[memory size: %d]" % (len(memory) * 2))
44
45  def cycle() :
46    pc = context[8]
47    # fetch an instruction from memory
48    instruction = memory[pc >> 1]
49
50    # predecode the instruction (extract opcode fields)
51    opc = (instruction & 0xf000) >> 12
52    imm = (instruction & 0x0800) >> 11
53    rst = (instruction & 0x0700) >> 8
54    rs1 = (instruction & 0x00e0) >> 5
55    rs2 = (instruction & 0x001c) >> 2
56    op2 = instruction & 0x0003
57    immediate = instruction & 0x00ff
58
59    # it's halt and catch fire, halt the simulator
60    if instruction == 0x0003 : return 0
61
62    # decode and execute
63    if imm == 0 :
64      if context[rs1] > 0x7fff : rs1 = context[rs1] - 0x10000
65      else : rs1 = context[rs1]
66      if context[rs2] > 0x7fff : rs2 = context[rs2] - 0x10000
67      else : rs2 = context[rs2]
68    else :
69      if context[rst] > 0x7fff : rs1 = context[rst] - 0x10000
70      else : rs1 = context[rst]
71      if immediate > 0x7f : immediate -= 0x100
72      rs2 = immediate
73
74    if ((imm == 0 and (op2 == 0 or op2 == 3)) or imm == 1) :
75      if opc == 0 :
76            if (imm == 1) : rs2 &= 0xff
77            context[rst] = rs1 & rs2
78      elif opc == 1 :
79            if (imm == 1) : rs2 &= 0xff
80            context[rst] = rs1 | rs2
81      elif opc == 2 :   context[rst] = rs1 ^ rs2
82      elif opc == 3 :
83            if rs1 < rs2 : context[rst] = 1
84            else : context[rst] = 0
85      elif opc == 4 :
86            if (rs1 & 0xffff) < (rs2 & 0xffff) : context[rst] = 1
87            else : context[rst] = 0
88      elif opc == 5 :   context[rst] = (rs1 & 0xffff) + (rs2 & 0xffff)
89      elif opc == 6 :   context[rst] = (rs1 & 0xffff) - (rs2 & 0xffff)
90      elif opc == 8 :   context[rst] = rs2
91      elif opc == 9 :   context[rst] = (context[rst] << 8) | (rs2 & 0xff)
92      elif opc == 10 :
93            if (imm == 1) :
94              if rs1 == 0 : pc = pc + rs2;
```

```
95            else :
96              if rs1 == 0 : pc = rs2 - 2
97      elif opc == 11 :
98            if (imm == 1) :
99              if rs1 != 0 : pc = pc + rs2;
100           else :
101             if rs1 != 0 : pc = rs2 - 2
102     else :       print ("[error (invalid computation / branch instruction)]")
103   elif (imm == 0 and op2 == 1) :
104     if opc == 0 :    context[rst] = (rs1 & 0xffff) >> 1
105     elif opc == 1 :    context[rst] = rs1 >> 1
106     else :       print ("[error (invalid shift instruction)]")
107   elif (imm == 0 and op2 == 2) :
108     if opc == 0 :
109           if (rs2 & 0x1) :
110             byte = memory[rs2 >> 1] & 0xff
111           else :
112             byte = memory[rs2 >> 1] >> 8
113
114           if byte > 0x7f : context[rst] = byte - 0x100
115           else : context[rst] = byte
116     elif opc == 1 :
117           if (rs2 & 0x1) :
118             memory[rs2 >> 1] = (memory[rs2 >> 1] & 0xff00) | (rs1 & 0xff)
119           else :
120             memory[rs2 >> 1] = (memory[rs2 >> 1] & 0x00ff) | ((rs1 & 0xff) << 8)
121     elif opc == 4 :
122           if (rs2 & 0xffff) == 0xf004 :     # emulate an input character device (
      address: 61444)
123             content[rst] = chr(raw_input('char? '));
124           elif (rs2 & 0xffff) == 0xf006 :     # emulate an input integer device (
      address: 61446)
125             context[rst] = int(raw_input('int? '));
126           else :
127             context[rst] = memory[rs2 >> 1]
128     elif opc == 5 :
129           if (rs2 & 0xffff) == 0xf000 :     # emulate an output character device (
      address: 61440)
130             sys.stdout.write(chr(rs1 & 0xff))
131           elif (rs2 & 0xffff) == 0xf002 :     # emulate an output integer device (
      address: 61442)
132             sys.stdout.write(str(rs1))
133           else :
134             memory[rs2 >> 1] = rs1
135     else :       print ("[error (invalid load/store instruction)]")
136   else :       print ("[error (invalid instruction)]")
137
138   # increment the program counter
139   pc = pc + 2
140   context[8] = pc
141   # fix the stored word to the matching hardware size
142   context[rst] &= 0xffff
143
144   return 1
145
146 def run(program) :
```

```
147   codes = {
148     0x0000:"and", 0x1000:"or", 0x2000:"xor", 0x3000:"slt",
149     0x4000:"sltu", 0x5000:"add", 0x6000:"sub", 0x8000:"ldr",
150     0x9000:"ldc", 0x0001:"lsr", 0x1001:"asr",
151     0x0002:"ldb", 0x1002:"stb", 0x4002:"ldw", 0x5002:"stw",
152     0xa000:"bez", 0xb000:"bnz"
153   }
154   cycles = 0;
155   args = sys.argv[1:]
156
157   while True :
158     inst = memory[context[8] >> 1]
159     last_pc = context[8]
160
161     if not cycle() : break
162     cycles += 1
163
164     if context[7] < context[9] :
165       print ("stack overflow detected!")
166       break;
167
168     if args :
169       if (inst & 0x0800) :
170         print ("pc: %04x instruction: %s r%d,%d" % (last_pc, codes[inst & 0xf000],
      (inst & 0x0700) >> 8, (inst & 0x00ff)))
171       else :
172         print ("pc: %04x instruction: %s r%d,r%d,r%d" % (last_pc, codes[inst & 0
      xf003], (inst & 0x0700) >> 8, (inst & 0x00e0) >> 5, (inst & 0x001c) >> 2))
173       print ("r0: [%04x] r1: [%04x] r2: [%04x] r3: [%04x]" % (context[0], context
      [1], context[2], context[3]))
174       print ("r4: [%04x] r5: [%04x] r6: [%04x] r7: [%04x]\n" % (context[4],
      context[5], context[6], context[7]))
175       a = raw_input()
176   print ("\n[ok]")
177   print ("%d cycles" % cycles)
178
179
180
181 def main() :
182   program = sys.stdin.readlines()
183   if (check(program)) :
184     print ("[program has errors]")
185   else :
186     load(program)
187     sys.stdin = open('/dev/tty')
188     run(program)
189
190 if __name__ == "__main__" : main()
```