Processador TRM - Manual de referência

Sérgio Johann Filho

25 de julho de 2025

Essa mensagem de agredecimento foi intencionalmente deixada em branco.

Sumário

Capítulo 1

A função de um computador

Este capítulo apresenta a função de um sistema computacional, apresentando conceitos básicos que serão úteis para o entendimento do restante desse manual.

1.1 Um sistema computacional típico

Um sistema computacional típico consiste dos seguintes elementos principais:

- Uma unidade central de processamento (CPU)
- Memória para armazenamento do programa
- Portas de entrada e saída

A memória é usada para armazenar *instruções*, que são porções de informação que definem as atividades a serem executadas pela CPU, assim como *dados*, que são porções de informação que são processados pelas instruções executadas. Um agrupamento de instruções e dados logicamente relacionados e armazenados na memória são chamados de *programa*. A CPU lê cada instrução da memória em uma sequência determinada e utiliza essa sequência para realizar o processamento, executando as instruções. Se as instruções lidas e executadas pela CPU forem coerentes e lógicas, o processamento do programa irá produzir resultado válido.

A CPU pode acessar qualquer informação na memória de forma eficiente, no entanto a memória nem sempre possui tamanho o suficiente para armazenar toda a informação necessária para uma aplicação específica. Isso pode ser resolvido por meio de uma ou mais *portas*. A CPU pode endereçar essas portas e acessar dados por meio destas, de maneira que torna-se possível enviar e receber dados de dispositivos periféricos. As portas podem ser de entrada ou saída. Uma porta de entrada pode ser usada para leitura de informação externa (de um dispositivo periférico como um teclado, por exemplo) e uma porta de saída pode ser usada para escrita de informação para o exterior (para um periférico como um monitor, por exemplo).

Em suma, a CPU é resposável por unificar todas as partes de um sistema computacional. Ela controla as funções realizadas por outros componentes, sequenciando todos os eventos. A CPU deve poder buscar instruções, decodificar estas e as executar. Ela também pode referenciar a memória para acessos (leitura e escrita de variáveis, por exemplo), assim como portas para comunicação com dispositivos periféricos.

1.2 A arquitetura de uma CPU

Geralmente uma unidade central de processamento consiste das seguintes unidades funcionais interconectadas:

- Registradores
- Unidade lógica e aritmética (ALU)
- Unidade de controle

Cada computador possui um *tamanho de palavra* que é uma característica da máquina em questão. A largura da palavra de um computador é determinada pelo tamanho dos elementos de armazenamento interno (registradores) e caminhos de interconexão (barramentos). Por exemplo, um computador que pode armazenar e transferir 16 bits de informação por vez possui um tamanho de palavra de 16 bits, e é referenciado como um processador de 16 bits. Dados e instruções são armazenados na memória com um formato de múltiplos de 8 bits (um byte), ou seja, 8 bits, 16 bits, 24 bits, etc. Um computador cujo tamanho da palavra seja um múltiplo de 8 bits geralmente processa de maneira mais eficiente dados com esse tamanho. Em algumas implementações, uma instrução possui exatamente o tamanho da palavra da arquitetura, em outras, uma instrução é composta por uma sequência de bytes ou múltiplas palavras.

O formato das instruções depende da maneira como a arquitetura é definida, o que inclui o número de instruções, tamanho da palavra, número de *registradores*, modos de endereçamento, estratégia de sequenciamento para execução de instruções, entre outros detalhes. Muitos desses detalhes são expostos no *formato de instrução*, composto por diversos campos de bits que podem ser decodificados processador durante o processo de execução de cada instrução.

Registradores são unidades para o armazenamento de informação de forma temporária. Alguns registradores, como o contador de programa (PC) e outros possuem um papel especial. Outros registradores, possuem um propósito mais genérico e normalmente podem ser usados de forma flexível pelo programador.

1.2.1 Registradores de propósito geral (GPRs)

Os registradores de propósito geral armazenam operandos a serem manipulados pela *unidade lógica e aritmética* (ALU), bem como o resultado do processamento de uma instrução. Uma instrução típica pode solicitar uma operação se soma de dois valores à ALU, e armazenar o resultado. De forma geral, os registradores de propósito geral (GPRs) são implementados em um vetor (banco) de registradores. Em uma operação típica, são selecionados dois registradores com os operandos (fontes) a serem processadors e um registrador (destino) para o armazenamento do resultado.

De maneira geral, são incluídos em uma CPU registradores que podem ser usados para o armazenamento de operandos e resultados intermediários do processamento. A disponibilidade de registradores de propósito geral elimina a necessidade de acessos adicionais à memória, o que aumenta a velocidade do processamento e a sua eficiência, uma vez que acessos à memória tendem a ser mais lentos que acessos à registradores[1].

[1] Acessos adicionais à memória podem ser complicados, uma vez que dependem do desempenho do sistema de memória.

1.2.2 Contador de programa (PC)

As instruções que compõem um programa são armazenadas na memória e a CPU referencia o conteúdo da memória para acessar as mesmas durante a execução do programa. Para acessar uma instrução é necessário que a CPU saiba o endereço da próxima instrução a ser trazida da memória.

No processador é mantido uma referência ao endereço da próxima instrução, sendo tal referência armazenada em um registrador chamado *contador de programa* (PC). Esse registrador é atualizado antes da busca de cada instrução.

As instruções de um programa são armazenadas sequencialmente na memória, de tal forma que endereços menores armazenam as primeiras instruções de um programa, já endereços maiores instruções posteriores. Essa regra pode ser modificada caso sejam utilizadas instrução de desvio (ou saltos) no programa. Uma instrução de desvio (transferência de controle) contém uma referência ao endereço da instrução a ser executada no caso do desvio ser tomado. A próxima instrução pode ser armazenada em qualquer lugar da memória. Durante a execução de uma instrução de desvio, o processador substitui o conteúdo do registrador PC com o endereço da próxima instrução, seja ela o alvo do desvio (caso tomado) ou a instrução seguinte (caso não tomado).

Um tipo especial de transferência de fluxo ocorre durante a execução de subprogramas. Nesse caso, o processador deve lembrar do endereço que deve retornar ao término do subprograma e pode assim continuar a execução do programa principal.

O processador possui uma maneira especial de executar subprogramas. Antes de realizar o desvio para o subprograma, é necessário calcular o endereço de retorno e armazenar o mesmo em um registrador ou na memória, em um espaço conhecido como *pilha*. Após, o endereço do subprograma é colocado no PC por meio de uma instrução de desvio incondicional (salto) e a primeira instrução do subprograma pode ser buscada na memória. A última instrução em um subprograma é uma instrução de retorno. Essa instrução também é um desvio incondicional, que restaura o valor do PC para o endereço salvo anteriormente em um registrador ou na pilha antes de retomar a execução do programa principal. Registradores ou a pilha também são utilizados para passagem de argumentos à subprogramas ou para o retorno do resultado do processamento.

1.2.3 Registrador de endereço

A CPU pode utilizar registradores para o armazenamento de enderereços de memória a serem acessados para a transferência de dados. Se um registrador de endereço for programável[2], o programa pode criar um endereço no registrador antes de executar uma operação de referência à memória (uma instrução que lê ou escreve dados na memória). Essas operações implementam o acesso à variáveis e constantes.

Registradores de endereço podem ser implementados no próprio banco de registradores, ou seja, podem ser utilizados registradores de propósito geral para essa finalidade.

1.2.4 Unidade lógica e aritmética

A unidade lógica e aritmética (ALU) é uma parte da CPU responsável pela operação aritmética e lógica sobre dados. A ALU possui um circuito *somador*, responsável por combinar o conteúdo de dois registradores de acordo com as regras de aritmética binária. Isso permite que o processador realize operações aritméticas sobre dados obtidos da memória ou portas de entrada.

[2]Por meio de uma ou mais instruções o programador pode modificar o conteúdo do registrador.

Utilizando apenas um somador, um bom programador pode escrever rotinas capazes de realizar a subtração, multiplicação e divisão, o que define capacidade aritmética completa ao computador. Na prática, a maioria das ALUs implementam também outras funções, como operações booleanas lógicas e operações de deslocamento.

Além disso, a ALU possui sinais (*flags*) que especificam determinadas condições durante a execução de operações lógicas e aritméticas[3]. È possível dessa forma realizar a comparação de valores e desvios condicionais, dependendo dos valores de tais sinais após o cálculo do resultado da última operação.

1.2.5 Decodificação de instruções e unidade de controle

O processador busca uma instrução na memória de programa fornecendo à mesma um endereço armazenado no *contador de programa* (PC). A memória retorna a instrução para o processador, que a armazena no *registrador de instrução* e passa a processar essa por meio de uma sequência de passos.

Após a busca, a instrução armazenada é decodificada, sendo essa a principal responsabilidade da *unidade de controle*. De acordo com o tipo de instrução, a unidade de controle será responsável por gerar sinais para a ativação de partes do processador, responsáveis pela execução de cada instrução.

A instrução armazenada no registrador de instrução pode ser decodificada e utilizada para ativar as diferentes partes internas do *caminho de dados*. Cada parte interna está associada com a execução de uma instrução particular, e a habilitação ou não de cada parte, assim como o sequenciamento das operações é responsável pelo processamento aritmético, lógico ou de transferência, que define o resultado da operação de uma instrução.

1.3 Operação de um processador

Todas as operações de um processador ocorrem de forma cíclica, ou seja, o processador busca uma instrução, realiza sua decodificação, a executa, produzindo o resultado da operação e volta a buscar a próxima instrução, repetindo o ciclo. A execução de uma instrução, composta por essas etapas é chamada de *ciclo de instrução*.

1.3.1 Busca e execução de instrução

A primeira operação no ciclo de instrução é chamado de *busca de instrução*, que compreende em trazer a próxima instrução a ser executada da memória. A CPU sinaliza uma operação de leitura e o conteúdo do contador de programa (PC) é enviado à memória, que responde com a instrução endereçada ou parte dessa[4]. A primeira parte de uma instrução é colocada no registrador de instrução, e caso a instrução consista de mais partes, ciclos adicionais de busca são realizados. No momento em que toda a instrução tiver sido buscada, o contador de programa (PC) é incrementado e a instrução é decodificada. A operação especificada na instrução será executada nos estágios seguintes, sendo que a instrução pode realizar acessos adicionais à memória (no caso de instruções que referenciam dados, ou portas de entrada e saída). Em instruções sem acessos adicionais, como operações aritméticas ou lógicas que endereçam apenas registradores, o próximo passo é a execução, que computará o resultado.

[3]Nem sempre os sinais de condição são apresentados ao programador, ou seja, podem ser apenas utilizados internamente durante o processamento.

[4]Algumas instruções precisam de múltiplos acessos à memória para serem buscadas.

1.3.2 Acessos adicionais à memória

Uma operação de busca é um tipo especial de operação de leitura realizada na memória, onde a instrução é trazida para o processador para ser executada. Uma instrução pode solicitar acessos adicionais à memória (para acesso à dados). Nesse caso, a CPU gera novamente uma operação de leitura ou escrita à memória, enviando o endereço solicitado pela instrução à memória. Em uma operação de leitura, o dado é transferido para um registrador de propósito geral e em uma operação de escrita, o dado armazenado em um registrador é escrito na memória.

Capítulo 2

A arquitetura TRM

O processador TRM (Tiny RISC machine) possui uma arquitetura planejada de acordo com a filosofia RISC. Essa arquitetura foi definida com o objetivo de oferecer ao programador um conjunto de instruções básico, de maneira que uma quantidade reduzida de recursos de hardware seja necessária para implementar seu conjunto de instruções[1]. Entre as decisões de projeto, foi utilizado um banco de registradores com propósito geral, não há qualificadores de estado (flags) para operações, não foram definidas unidades para multiplicação e divisão e há apenas um conjunto limitado de modos de endereçamento.

Um pequeno conjunto de instruções foi definido na arquitetura TRM (25 instruções básicas). Apesar do pequeno número de instruções, estas são poderosas o suficiente para realizar todas as operações de máquinas com um maior número de instruções. Para isso, em algumas situações uma instrução pode ser utilizada de forma pouco ortodoxa ou uma combinação de instruções pode ser utilizada para implementação de um comportamento mais elaborado. As instruções são separadas nas seguintes classes:

1. *Lógica e aritmética* (and, or, xor, add, sub)
2. *Comparação* (tlt, tge, tbl, tae, teq, tne)
3. *Deslocamento* (lsl, lsr, asr)
4. *Transferência de dados* (ldw, ldb, lbu, stw, stb)
5. *Transferência de controle* (blt, bge, bbl, bae, beq, bne)

Seguindo a filosofia RISC ao extremo, as instruções são definidas em uma representação binária que utiliza apenas um formato de instrução[2]. Dessa forma, a lógica necessária para a decodificação de instruções é reduzida significativamente, comparado ao que seria necessário para decodificação de instruções com múltiplos formatos (como nas arquiteturas ARM e RISC-V por exemplo) ou instruções com tamanho variável (como na arquitetura x64 por exemplo).

2.1 Registradores

Assim como outros processadores RISC, o TRM utiliza operações de transferência de dados (*load / store*) para acesso à dados. Para que operações lógicas e aritméticas,

[1] O conjunto de instruções foi definido com o intuito de minimizar a complexidade da arquitetura e ao mesmo tempo permitir a síntese de operações mais complexas por meio de poucas operações simples.

[2] Uma instrução pode ou não ser seguida de uma palavra para o armazenamento de um valor imediato / literal (constante).

comparação, deslocamento, entre outras possam ser execututadas, é necessário que os operandos[3] sejam primeiramente trazidos da memória em um ou mais registradores de propósito geral (GPRs), ou que sejam utilizadas constantes no próprio fluxo de instruções.

São definidos 16 registradores (*r0 - r15*) e estes podem ser utilizados para qualquer finalidade[4], sendo apenas recomendado seu uso em função das convenções apresentadas abaixo. Ao utilizar diferentes registradores em um programa, um programador tipicamente referencia os mesmos por seu nome, onde é indicada a sua função.

Registrador	**Nome**	**Função**
`r0`	`zr`	Constante zero
`r1`	`a0`	Argumentos
`r2`	`a1`	
`r3`	`a2`	
`r4`	`a3`	
`r5`	`v0`	Variáveis locais
`r6`	`v1`	
`r7`	`v2`	
`r8`	`v3`	
`r9`	`v4`	
`r10`	`v5`	
`r11`	`v6`	
`r12`	`v7`	
`r13`	`v8/fp`	Ponteiro de moldura
`r14`	`sp`	Ponteiro de pilha
`r15`	`lr`	Endereço de retorno

Além dos 16 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 decodificada, o PC avança para a próxima posição[5]. Instruções de transferência de fluxo (desvios) podem fazer com que o PC seja atualizado com o destino do desvio, caso este seja tomado. As instruções possuem um tamanho de 16 bits, portanto o contador de programa é incrementado com esse tamanho.

2.2 Formato de instruções

Existe apenas um formato de instrução na arquitetura TRM, sendo que uma instrução pode conter um campo imediato opcional, armazenado na palavra seguinte no fluxo de instruções. No formato, são definidos campos que descrevem a instrução, a presença ou não de um campo imediato e dois registradores, utilizados como fonte ou destino das operações.

Os campos *Opcode* e *Opcode 2* definem o tipo de operação e a operação específica respectivamente. O campo *Imm* define se a operação possui ou não um campo imediato.

[3] Operandos são valores utilizados para entrada para uma computação.

[4] Exceto *r0*, que possui o seu valor constante zero.

[5] Instruções que possuem um operando imadiato armazenam esse operando como uma constante na próxima posição de memória após a instrução, dessa forma essas instruções causam um avanço de duas posições no PC.

$I_{<15:13>}$	$I_{<12:9>}$	$I_{<8>}$	$I_{<7:4>}$	$I_{<3:0>}$
Opcode	*Opcode 2*	*Imm*	*Reg A*	*Reg B*
x x x	x x x x	i	r r r r	r r r r

Os campos *Reg A* e *Reg B* definem os registradores a serem utilizados na operação.

2.2.1 Instruções regulares

Em instruções regulares (sem um campo imediato), são referenciados dois registradores, sendo estes utilizados como fonte ou destino da operação, dependendo da instrução específica. Exemplos:

Operação	**Significado**
`add r1,r2`	`r1 = r1 + r2`
`xor r1,r1`	`r1 = 0`
`ldw r1,r2`	`r1 = MEM[r2]`
`blt r1,r2`	`pc = r1 < 0 ? r2 : pc + 2`

2.2.2 Instruções com campo imediato

Em instruções com um campo imediato, são referenciados dois registradores, sendo estes utilizados como fonte ou destino da operação, dependendo da instrução específica. Além disso, um campo imediato (constante) associado à instrução é usado como fonte. Exemplos:

Operação	**Significado**
`add r1,r1,1`	`r1 = r1 + 1`
`lsl r1,r2,3`	`r1 = r2 << 3`
`bge r1,r2,12340`	`pc = r1 >= r2 ? 12340 : pc + 2`
`stw r1,r2,12`	`MEM[r2 + 12] = r1`

2.3 O conjunto de instruções

O conjunto de instruções definido na arquitetura é apresentado a seguir. Diversos códigos de operação são reservados para extensões futuras. As instruções possuem um tamanho de 16 bits, associado a uma boa densidade do código.

Em operações lógicas e aritméticas, comparações e operações de deslocamento que possuem o formato de instrução regular (sem campo imediato), *Reg A* é utilizado como operando *fonte 1* e como destino para o resultado da operação e *Reg B* é utilizado como operando *fonte 2*. Nas operações com formato imediato, *Reg A* é utilizado como *destino* para o resultado, *Reg B* é utilizado como operando *fonte 1* e a palavra seguinte à instrução contendo o valor imaditato é utilizada como operando *fonte 2*.

Em operações de transferência de dados que possuem o formato de instrução regular, *Reg A* é utilizado como operando *fonte 1* (instruções de armazenamento) e como *destino* para o resultado da operação (instruções de carga) e *Reg B* é utilizado como operando *fonte 2*, contendo o endereço da memória. Nas operações com formato imediato, Os registradores possuem o mesmo papel, no entanto é somado um valor imediato ao valor

armazenado no operando *fonte 2* (*Reg B*), sendo realizado o cálculo do endereço *base + deslocamento*.

Em operações de transferência de controle que possuem o formato de instrução regular (sem campo imediato), *Reg A* é utilizado como operando *fonte 1*, *zero* é utilizado como *fonte 2* e *Reg B* é utilizado como endereço da memória. Nas operações com formato imediato, *Reg A* é utilizado *fonte 1*, *Reg B* é utilizado como operando *fonte 2* e a palavra seguinte à instrução contendo o valor imaditato é utilizada como endereço da memória.

A tabela abaixo apresenta um quadro das instruções implementadas na arquitetura[6]. São apresentados os nomes e códigos de operação, a descrição e o resultado da instrução nos formatos regular e imediato.

Instrução	Opcode	Descrição	Regular	Imediato
and	000 0000	logical and	`ra = ra & rb`	`ra = rb & imm`
or	000 0010	logical or	`ra = ra | rb`	`ra = rb | imm`
xor	000 0011	logical exclusive or	`ra = ra ^ rb`	`ra = rb ^ imm`
add	000 0100	arithmetic add	`ra = ra + rb`	`ra = rb + imm`
sub	000 0101	arithmetic subtract	`ra = ra - rb`	`ra = rb - imm`
tlt	101 0000	test if less than	`ra = ra < rb ? 1 : 0`	`ra = rb < imm ? 1 : 0`
tge	101 0001	test if greater or equal	`ra = ra >= rb ? 1 : 0`	`ra = rb >= imm ? 1 : 0`
tbl	101 0100	test if below	`ra = ra < rb ? 1 : 0`	`ra = rb < imm ? 1 : 0`
tae	101 0101	test if above or equal	`ra = ra >= rb ? 1 : 0`	`ra = rb >= imm ? 1 : 0`
teq	101 1000	test if equal	`ra = ra == rb ? 1 : 0`	`ra = rb == imm ? 1 : 0`
tne	101 1001	test if not equal	`ra = ra != rb ? 1 : 0`	`ra = rb != imm ? 1 : 0`
lsl	001 1000	shift left	`ra = ra << rb`	`ra = rb << imm`
lsr	001 1010	shift right	`ra = ra >>> rb`	`ra = rb >>> imm`
asr	001 1011	arithmetic shift right	`ra = ra >> rb`	`ra = rb >> imm`
ldw	010 0000	load word	`ra = mem[rb]`	`ra = mem[rb + imm]`
ldb	010 0010	load byte	`ra = mem[rb]`	`ra = mem[rb + imm]`
lbu	010 0011	load byte unsigned	`ra = mem[rb]`	`ra = mem[rb + imm]`
stw	010 0100	store word	`mem[rb] = ra`	`mem[rb + imm] = ra`
stb	010 0110	store byte	`mem[rb] = ra`	`mem[rb + imm] = ra`
blt	100 0000	branch if less than	`pc = ra < 0 ? rb : pc + 2`	`pc = ra < rb ? imm : pc + 2`
bge	100 0001	branch if greater or equal	`pc = ra >= 0 ? rb : pc + 2`	`pc = ra >= rb ? imm : pc + 2`
bbl	100 0100	branch if below	`pc = ra < 0 ? rb : pc + 2`	`pc = ra < rb ? imm : pc + 2`
bae	100 0101	branch if above or equal	`pc = ra >= 0 ? rb : pc + 2`	`pc = ra >= rb ? imm : pc + 2`
beq	100 1000	branch if equal	`pc = ra == 0 ? rb : pc + 2`	`pc = ra == rb ? imm : pc + 2`
bne	100 1001	branch if not equal	`pc = ra != 0 ? rb : pc + 2`	`pc = ra != rb ? imm : pc + 2`
hlt	111 1100	halt execution (simulador)		

2.3.1 Lógica e aritmética

AND - logical and (bitwise logical product)

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

[6] A instrução *hlt* não é uma instrução real da arquitetura. Ela é uma pseudo instrução utilizada para parar a execução de programas no simulador.

$I_{<15:13>}$	$I_{<12:9>}$	$I_{<8>}$	$I_{<7:4>}$	$I_{<3:0>}$
Opcode	*Opcode 2*	*Imm*	*Reg A*	*Reg B*
0 0 0	0 0 0 0	i	r r r r	r r r r

- AND Reg A, Reg B

  ```
  GPR[Reg A] ← GPR[Reg A] and GPR[Reg B]
  ```

- AND Reg A, Reg B, Immediate

  ```
  GPR[Reg A] ← GPR[Reg B] and Immediate
  ```

OR - logical or (bitwise logical sum)

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

$I_{<15:13>}$	$I_{<12:9>}$	$I_{<8>}$	$I_{<7:4>}$	$I_{<3:0>}$
Opcode	*Opcode 2*	*Imm*	*Reg A*	*Reg B*
0 0 0	0 0 1 0	i	r r r r	r r r r

- OR Reg A, Reg B

  ```
  GPR[Reg A] ← GPR[Reg A] or GPR[Reg B]
  ```

- OR Reg A, Reg B, Immediate

  ```
  GPR[Reg A] ← GPR[Reg B] or Immediate
  ```

XOR - logical exclusive or (bitwise logical difference)

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

$I_{<15:13>}$	$I_{<12:9>}$	$I_{<8>}$	$I_{<7:4>}$	$I_{<3:0>}$
Opcode	*Opcode 2*	*Imm*	*Reg A*	*Reg B*
0 0 0	0 0 1 1	i	r r r r	r r r r

- XOR Reg A, Reg B

  ```
  GPR[Reg A] ← GPR[Reg A] xor GPR[Reg B]
  ```

- XOR Reg A, Reg B, Immediate

  ```
  GPR[Reg A] ← GPR[Reg B] xor Immediate
  ```

ADD - arithmetic add

Soma dois valores e armazena o resultado em um registrador.

$I_{<15:13>}$	$I_{<12:9>}$	$I_{<8>}$	$I_{<7:4>}$	$I_{<3:0>}$
Opcode	*Opcode 2*	*Imm*	*Reg A*	*Reg B*
0 0 0	0 1 0 0	i	r r r r	r r r r

- ADD Reg A, Reg B

```
GPR[Reg A] ← GPR[Reg A] + GPR[Reg B]
```

- ADD Reg A, Reg B, Immediate

```
GPR[Reg A] ← GPR[Reg B] + Immediate
```

SUB - arithmetic subtract

Subtrai dois valores e armazena o resultado em um registrador.

$I_{<15:13>}$	$I_{<12:9>}$	$I_{<8>}$	$I_{<7:4>}$	$I_{<3:0>}$
Opcode	*Opcode 2*	*Imm*	*Reg A*	*Reg B*
0 0 0	0 1 0 1	i	r r r r	r r r r

- SUB Reg A, Reg B

```
GPR[Reg A] ← GPR[Reg A] - GPR[Reg B]
```

- SUB Reg A, Reg B, Immediate

```
GPR[Reg A] ← GPR[Reg B] - Immediate
```

2.3.2 Comparação

TLT - test if less than

Testa se o primeiro valor é menor que o segundo. O teste resulta em verdadeiro (1) ou falso (0), sendo o resultado armazenado em um registrador.

$I_{<15:13>}$	$I_{<12:9>}$	$I_{<8>}$	$I_{<7:4>}$	$I_{<3:0>}$
Opcode	*Opcode 2*	*Imm*	*Reg A*	*Reg B*
1 0 1	0 0 0 0	i	r r r r	r r r r

- TLT Reg A, Reg B

```
GPR[Reg A] ← GPR[Reg A] < GPR[Reg B] ? 1 :  0
```

- TLT Reg A, Reg B, Immediate

```
GPR[Reg A] ← GPR[Reg B] < Immediate ?  1 :  0
```

TGE - test if greater or equal

Testa se o primeiro valor é maior ou igual ao o segundo. O teste resulta em verdadeiro (1) ou falso (0), sendo o resultado armazenado em um registrador.

$I_{<15:13>}$	$I_{<12:9>}$	$I_{<8>}$	$I_{<7:4>}$	$I_{<3:0>}$
Opcode	*Opcode 2*	*Imm*	*Reg A*	*Reg B*
1 0 1	0 0 0 1	i	r r r r	r r r r

- TGE Reg A, Reg B

```
GPR[Reg A] ← GPR[Reg A] >= GPR[Reg B] ? 1 :  0
```

- TGE Reg A, Reg B, Immediate

```
GPR[Reg A] ← GPR[Reg B] >= Immediate ?  1 :  0
```

TBL - test if below

Testa se o primeiro valor é menor que o segundo (valores não sinalizados). O teste resulta em verdadeiro (1) ou falso (0), sendo o resultado armazenado em um registrador.

$I_{<15:13>}$	$I_{<12:9>}$	$I_{<8>}$	$I_{<7:4>}$	$I_{<3:0>}$
Opcode	*Opcode 2*	*Imm*	*Reg A*	*Reg B*
1 0 1	0 1 0 0	i	r r r r	r r r r

- TBL Reg A, Reg B

```
GPR[Reg A] ← GPR[Reg A] < GPR[Reg B] ? 1 :  0
```

- TBL Reg A, Reg B, Immediate

```
GPR[Reg A] ← GPR[Reg B] < Immediate ?  1 :  0
```

TAE - test if above or equal

Testa se o primeiro valor é maior ou igual ao segundo (valores não sinalizados). O teste resulta em verdadeiro (1) ou falso (0), sendo o resultado armazenado em um registrador.

$I_{<15:13>}$	$I_{<12:9>}$	$I_{<8>}$	$I_{<7:4>}$	$I_{<3:0>}$
Opcode	*Opcode 2*	*Imm*	*Reg A*	*Reg B*
1 0 1	0 1 0 1	i	r r r r	r r r r

- TAE Reg A, Reg B

```
GPR[Reg A] ← GPR[Reg A] >= GPR[Reg B] ? 1 :  0
```

- TAE Reg A, Reg B, Immediate

```
GPR[Reg A] ← GPR[Reg B] >= Immediate ?  1 :  0
```

TEQ - test if equal

Testa se o primeiro valor é igual ao segundo. O teste resulta em verdadeiro (1) ou falso (0), sendo o resultado armazenado em um registrador.

$I_{<15:13>}$	$I_{<12:9>}$	$I_{<8>}$	$I_{<7:4>}$	$I_{<3:0>}$
Opcode	*Opcode 2*	*Imm*	*Reg A*	*Reg B*
1 0 1	1 0 0 0	i	r r r r	r r r r

- TEQ Reg A, Reg B

```
GPR[Reg A] ← GPR[Reg A] == GPR[Reg B] ? 1 :  0
```

- TEQ Reg A, Reg B, Immediate

```
GPR[Reg A] ← GPR[Reg B] == Immediate ?  1 :  0
```

$I_{<15:13>}$	$I_{<12:9>}$	$I_{<8>}$	$I_{<7:4>}$	$I_{<3:0>}$
Opcode	*Opcode 2*	*Imm*	*Reg A*	*Reg B*
1 0 1	1 0 0 1	i	r r r r	r r r r

TNE - test if not equal

Testa se o primeiro valor é diferente do segundo. O teste resulta em verdadeiro (1) ou falso (0), sendo o resultado armazenado em um registrador.

- TNE Reg A, Reg B

```
GPR[Reg A] ← GPR[Reg A] != GPR[Reg B] ? 1 :  0
```

- TNE Reg A, Reg B, Immediate

```
GPR[Reg A] ← GPR[Reg B] != Immediate ?  1 :  0
```

2.3.3 Deslocamento

LSL - logical shift left

Realiza a o deslocamento lógico à esquerda e armazena o resultado em um registrador. Zeros são inseridos na parte menos significativa do resultado.

$I_{<15:13>}$	$I_{<12:9>}$	$I_{<8>}$	$I_{<7:4>}$	$I_{<3:0>}$
Opcode	*Opcode 2*	*Imm*	*Reg A*	*Reg B*
0 0 1	1 0 0 0	i	r r r r	r r r r

- LSL Reg A, Reg B

```
GPR[Reg A] ← GPR[Reg A] << GPR[Reg B]
```

- LSL Reg A, Reg B, Immediate

```
GPR[Reg A] ← GPR[Reg B] << Immediate
```

LSR - logical shift right

Realiza a o deslocamento lógico à direita e armazena o resultado em um registrador. Zeros são inseridos na parte mais significativa do resultado.

$I_{<15:13>}$	$I_{<12:9>}$	$I_{<8>}$	$I_{<7:4>}$	$I_{<3:0>}$
Opcode	*Opcode 2*	*Imm*	*Reg A*	*Reg B*
0 0 1	1 0 1 0	i	r r r r	r r r r

- LSR Reg A, Reg B

```
GPR[Reg A] ← GPR[Reg A] >>> GPR[Reg B]
```

- LSR Reg A, Reg B, Immediate

```
GPR[Reg A] ← GPR[Reg B] >>> Immediate
```

$I_{<15:13>}$	$I_{<12:9>}$	$I_{<8>}$	$I_{<7:4>}$	$I_{<3:0>}$
Opcode	*Opcode 2*	*Imm*	*Reg A*	*Reg B*
0 0 1	1 0 1 1	i	r r r r	r r r r

ASR - arithmetic shift right

Realiza a o deslocamento aritmético à direita e armazena o resultado em um registrador. Zeros ou uns são inseridos na parte mais significativa do resultado, preservando o sinal.

- ASR Reg A, Reg B

  `GPR[Reg A] ← GPR[Reg A] >> GPR[Reg B]`

- ASR Reg A, Reg B, Immediate

  `GPR[Reg A] ← GPR[Reg B] >> Immediate`

2.3.4 Transferência de dados

LDW - load word

Transfere uma palavra da memória para um registrador. O endereço acessado deve ser alinhado.

$I_{<15:13>}$	$I_{<12:9>}$	$I_{<8>}$	$I_{<7:4>}$	$I_{<3:0>}$
Opcode	*Opcode 2*	*Imm*	*Reg A*	*Reg B*
0 1 0	0 0 0 0	i	r r r r	r r r r

- LDW Reg A, Reg B

  `GPR[Reg A] ← MEM[Reg B]`

- LDW Reg A, Reg B, Immediate

  `GPR[Reg A] ← MEM[Reg B + Immediate]`

LDB - load byte

Transfere um byte da memória para um registrador. O registrador de destino que armazena o resultado tem o byte carregado na parte menos significativa, sendo o sinal preservado.

$I_{<15:13>}$	$I_{<12:9>}$	$I_{<8>}$	$I_{<7:4>}$	$I_{<3:0>}$
Opcode	*Opcode 2*	*Imm*	*Reg A*	*Reg B*
0 1 0	0 0 1 0	i	r r r r	r r r r

- LDB Reg A, Reg B

  `GPR[Reg A] ← SEXT(MEM[Reg B]`$_{<7:0>}$`)`

- LDB Reg A, Reg B, Immediate

  `GPR[Reg A] ← SEXT(MEM[Reg B + Immediate]`$_{<7:0>}$`)`

LBU - load byte unsigned

Transfere um byte da memória para um registrador. O registrador de destino que armazena o resultado tem o byte carregado na parte menos significativa, e a parte mais significativa preenchida com zeros.

$I_{<15:13>}$	$I_{<12:9>}$	$I_{<8>}$	$I_{<7:4>}$	$I_{<3:0>}$
Opcode	*Opcode 2*	*Imm*	*Reg A*	*Reg B*
0 1 0	0 0 1 1	i	r r r r	r r r r

- LBU Reg A, Reg B

  `GPR[Reg A] ← ZEXT(MEM[Reg B]`$_{<7:0>}$`)`

- LBU Reg A, Reg B, Immediate

  `GPR[Reg A] ← ZEXT(MEM[Reg B + Immediate]`$_{<7:0>}$`)`

STW - store word

Transfere uma palavra para memória a partir de um registrador. O endereço acessado deve ser alinhado.

$I_{<15:13>}$	$I_{<12:9>}$	$I_{<8>}$	$I_{<7:4>}$	$I_{<3:0>}$
Opcode	*Opcode 2*	*Imm*	*Reg A*	*Reg B*
0 1 0	0 1 0 0	i	r r r r	r r r r

- STW Reg A, Reg B

  `MEM[Reg B] ← GPR[Reg A]`

- STW Reg A, Reg B, Immediate

  `MEM[Reg B + Immediate] ← GPR[Reg A]`

STB - load byte

Transfere um byte para memória a partir de um registrador. O registrador de fonte tem o byte carregado da parte menos significativa.

$I_{<15:13>}$	$I_{<12:9>}$	$I_{<8>}$	$I_{<7:4>}$	$I_{<3:0>}$
Opcode	*Opcode 2*	*Imm*	*Reg A*	*Reg B*
0 1 0	0 1 1 0	i	r r r r	r r r r

- STB Reg A, Reg B

  `MEM[Reg B] ← GPR[Reg A]`$_{<7:0>}$

- STB Reg A, Reg B, Immediate

  `MEM[Reg B + Immediate] ← GPR[Reg A]`$_{<7:0>}$

2.3.5 Transferência de controle

BLT - branch if less than

Testa se o primeiro valor é menor que o segundo. Caso o teste seja verdadeiro, o valor do PC é atualizado com o endereço alvo.

$I_{<15:13>}$	$I_{<12:9>}$	$I_{<8>}$	$I_{<7:4>}$	$I_{<3:0>}$
Opcode	*Opcode 2*	*Imm*	*Reg A*	*Reg B*
1 0 0	0 0 0 0	i	r r r r	r r r r

- BLT Reg A, Reg B

```
PC ← GPR[Reg A] < 0 ?  GPR[Reg B] : PC + 2
```

- BLT Reg A, Reg B, Immediate

```
PC ← GPR[Reg A] < GPR[Reg B] ? Immediate :  PC + 2
```

BGE - branch if greater or equal

Testa se o primeiro valor é maior ou igual ao segundo. Caso o teste seja verdadeiro, o valor do PC é atualizado com o endereço alvo.

$I_{<15:13>}$	$I_{<12:9>}$	$I_{<8>}$	$I_{<7:4>}$	$I_{<3:0>}$
Opcode	*Opcode 2*	*Imm*	*Reg A*	*Reg B*
1 0 0	0 0 0 1	i	r r r r	r r r r

- BGE Reg A, Reg B

```
PC ← GPR[Reg A] >= 0 ?  GPR[Reg B] : PC + 2
```

- BGE Reg A, Reg B, Immediate

```
PC ← GPR[Reg A] >= GPR[Reg B] ? Immediate :  PC + 2
```

BBL - branch if below

Testa se o primeiro valor é menor que o segundo (valores não sinalizados). Caso o teste seja verdadeiro, o valor do PC é atualizado com o endereço alvo.

$I_{<15:13>}$	$I_{<12:9>}$	$I_{<8>}$	$I_{<7:4>}$	$I_{<3:0>}$
Opcode	*Opcode 2*	*Imm*	*Reg A*	*Reg B*
1 0 0	0 1 0 0	i	r r r r	r r r r

- BBL Reg A, Reg B

```
PC ← GPR[Reg A] < 0 ?  GPR[Reg B] : PC + 2
```

- BBL Reg A, Reg B, Immediate

```
PC ← GPR[Reg A] < GPR[Reg B] ? Immediate :  PC + 2
```

$I_{<15:13>}$	$I_{<12:9>}$	$I_{<8>}$	$I_{<7:4>}$	$I_{<3:0>}$
Opcode	*Opcode 2*	*Imm*	*Reg A*	*Reg B*
1 0 0	0 1 0 1	i	r r r r	r r r r

BAE - branch if above or equal

Testa se o primeiro valor é maior ou igual ao segundo (valores não sinalizados). Caso o teste seja verdadeiro, o valor do PC é atualizado com o endereço alvo.

- BAE Reg A, Reg B

```
PC ← GPR[Reg A] >= 0 ?  GPR[Reg B] : PC + 2
```

- BAE Reg A, Reg B, Immediate

```
PC ← GPR[Reg A] >= GPR[Reg B] ? Immediate :  PC + 2
```

BEQ - branch if equal

Testa se o primeiro valor é igual ao segundo. Caso o teste seja verdadeiro, o valor do PC é atualizado com o endereço alvo.

$I_{<15:13>}$	$I_{<12:9>}$	$I_{<8>}$	$I_{<7:4>}$	$I_{<3:0>}$
Opcode	*Opcode 2*	*Imm*	*Reg A*	*Reg B*
1 0 0	1 0 0 0	i	r r r r	r r r r

- BEQ Reg A, Reg B

```
PC ← GPR[Reg A] == 0 ?  GPR[Reg B] : PC + 2
```

- BEQ Reg A, Reg B, Immediate

```
PC ← GPR[Reg A] == GPR[Reg B] ? Immediate :  PC + 2
```

BNE - branch if not equal

Testa se o primeiro valor é diferente do segundo. Caso o teste seja verdadeiro, o valor do PC é atualizado com o endereço alvo.

$I_{<15:13>}$	$I_{<12:9>}$	$I_{<8>}$	$I_{<7:4>}$	$I_{<3:0>}$
Opcode	*Opcode 2*	*Imm*	*Reg A*	*Reg B*
1 0 0	1 0 0 1	i	r r r r	r r r r

- BNE Reg A, Reg B

```
PC ← GPR[Reg A] != 0 ?  GPR[Reg B] : PC + 2
```

- BNE Reg A, Reg B, Immediate

```
PC ← GPR[Reg A] != GPR[Reg B] ? Immediate :  PC + 2
```

2.4 Modos de endereçamento

2.4.1 Registrador

No modo de endereçamento a registrador, dois registradores são utilizados no formato de instrução. Os dois registradores são utilizados como operandos, sendo o primeiro também utilizado para o armazenamento do resultado da operação.

O tamanho total da instrução é de 2 bytes (1 palavra). A palavra contém a informação referente à instrução e os operandos. Exemplos:

Operação	**Significado**
`add r1,r2`	`r1 = r1 + r2`
`xor r1,r1`	`r1 = 0`
`asr r1,r2`	`r1 = r1 >> r2`
`teq r1,r2`	`r1 = r1 == r2 ? 1 : 0`

2.4.2 Registrador com imediato (literal)

No modo de endereçamento registrador com imediato, dois registradores são utilizados no formato de instrução, além de um valor literal. O segundo registrador e o valor literal são utilizados como operandos, sendo o primeiro registrador utilizado para o armazenamento do resultado da operação. Caso *r0* for utilizado como o primeiro registrador, o resultado da operação será descartado (exceto em desvios condicionais). Caso *r0* for utilizado como segundo registrador, a instrução poderá resultar em uma carga do valor literal no primeiro registrador, o que é útil para a carga de constantes. Caso o valor literal for zero, a instrução poderá resultar na cópia do valor de um registrador para outro.

O tamanho total da instrução é de 4 bytes (2 palavras). A primeira palavra contém a informação referente à instrução, o registrador para o armazenamento do resultado e o primeiro operando. A segunda palavra contém o valor literal, utilizado como segundo operando. Exemplos:

Operação	**Significado**
`and r1,r2,123`	`r1 = r2 and 123`
`add r1,r1,1`	`r1 = r1 + 1`
`xor r1,r1,-1`	`r1 = not r1`
`lsl r1,r2,3`	`r1 = r2 << 3`
`add r1,r0,12345`	`r1 = 12345`
`add r1,r2,0`	`r1 = r2`
`beq r0,r0,12345`	`pc = 12345`
`bge r1,r2,12345`	`pc = r1 >= r2 ? 12345 : pc + 2`
`tne r1,r2,3`	`r1 = r2 != 3 ? 1 : 0`

2.4.3 Indireto a registrador (base)

No modo de endereçamento indireto a registrador, dois registradores são utilizados no formato de instrução. O primeiro registrador é utilizado como a origem, destino ou como operando e o segundo registrador é utilizado como o endereço base referenciado. O operando zero pode ser utilizado de forma implícita em desvios condicionais. Caso *r0* for utilizado como o primeiro registrador, em operações de carga o valor será descartado,

em operações de armazenamento o valor zero será utilizado como origem e em operações de desvio este será realizado ou não de forma incondicional (comparação com zero).

O tamanho total da instrução é de 2 bytes (1 palavra). A palavra contém a informação referente à instrução, a origem (ou destino / operando) e o endereço base. Exemplos:

Operação	**Significado**
`ldw r1,r2`	`r1 = MEM[r2]`
`stw r1,r2`	`MEM[r2] = r1`
`beq r0,r1`	`pc = r1`
`blt r1,r2`	`pc = r1 < 0 ? r2 : pc + 2`

2.4.4 Indireto a registrador com deslocamento (base + deslocamento)

No modo de endereçamento indireto a registrador com deslocamento, dois registradores são utilizados no formato de instrução, além de um valor literal. O primeiro registrador é utilizado como origem ou destino e o segundo registrador e o valor literal são utilizados como operandos para uma operação de soma, efetivamente calculando o endereço base + deslocamento. Caso o registrador base for *r0*, o endereço é calculado de forma absoluta. Caso o valor literal for zero, o resultado é o mesmo da operação utilizando o modo de endereçamento anterior (indireto a registrador).

O tamanho total da instrução é de 4 bytes (2 palavras). A primeira palavra contém a informação referente à instrução, a origem (ou destino / operando) e o endereço base. A segunda palavra contém o valor literal, utilizado como deslocamente a partir da base. Exemplos:

Operação	**Significado**
`stw r1,r2,12`	`mem[r2 + 12] = r1`
`ldb r1,r0,1000`	`r1 = MEM[1000]`
`ldw r1,r2,-16`	`r1 = MEM[r2 - 16]`

Capítulo 3

Linguagem de montagem

A linguagem de montagem da arquitetura TRM define uma série de elementos em sua sintaxe. Tais elementos são usados para a implementação de programas. A sintaxe da linguagem será apresentada brevemente, assim como alguns exemplos, os quais ilustram os principais usos da linguagem.

3.1 Sintaxe

A sintaxe da linguagem de montagem é bastante simples, contendo apenas os elementos essenciais para criação de programas. Não é necessário definir regiões separadas para código e dados[1] e também não é necessário o uso de diretivas tradicionalmente utilizadas em montadores para outras arquiteturas.

3.1.1 Comentários

Comentários podem ser adicionados ao código de montagem, de acordo com algumas regras. O caracter ';' é utilizado para inserção de comentários, sendo o restante da linha o conteúdo do comentário. Após a inserção do caracter ';', o restante da linha será ignorado no processo de montagem. A inclusão de comentários no código deve ser feita apenas com caracteres da língua inglesa.

3.1.2 Endereços

Um endereço de memória é um identificador numérico único para uma localização específica na memória do computador. Endereços são valores que indicam a posição de objetos, como palavras contendo instruções ou dados na memória. Endereços são utilizados para criação de referências a objetos, de tal maneira que, caso a memória seja acessada nessa posição (derreferência), têm-se o acesso ao conteúdo do endereço.

3.1.3 Referências (símbolos)

Referências são nomes simbólicos, ou seja, uma abstração de endereços de memória, que são utilizados pelo programa em linguagem de montagem para que a ferramenta de montagem (montador) os resolva. Durante o processo de montagem do programa, as referências são substituídas pelo seu valor numérico. Dessa forma, uma referência é um nome que se dá a uma posição (ou endereço) de memória.

[1] A arquitetura TRM define um único espaço de endereços. No mesmo espaço de endereços são armazenadas instruções e dados.

Referências são utilizadas para declaração de pontos específicos no código, tais como destino de saltos, ponto de entrada em funções / subprogramas e também como declarações de estruturas de dados, como variáveis e vetores. A ferramenta para montagem é responsável por resolver as referências, permitindo que estas assumam no código objeto um valor numérico para sua codificação nas instruções montadas.

O valor numérico de uma referência será gerado em função do espaço alocado previamente a declaração da mesma. Esse espaço pode ser ocupado por declarações de instruções ou estruturas de dados. A referência, em si, não ocupa memória alguma[2], sendo apenas um símbolo no programa, ou seja, uma referência ao endereço da próxima palavra na listagem do código objeto gerado pelo montador. A seguir, é apresentado um exemplo do uso de referências em um programa:

```
ref1
    (instruções)
    ...
ref2
    (instruções)
    ...
ref3
    (dados)
    ...
```

No exemplo, são apresentados três símbolos (*ref1*, *ref2* e *ref3*). Referências devem ser declaradas com alinhamento à esquerda, sem tabulações ou espaços e sem um caracter finalizador (como dois pontos, por exemplo).

3.1.4 Instruções

Instruções são comandos que correspondem diretamente a operações que CPU deve executar no contexto de um programa. Cada linha de um programa em linguagem de montagem que realiza uma ação concreta é uma instrução. Instruções são representações legíveis do que se tornará o código de máquina após o processo de montagem.

Na sintaxe da linguagem de montagem, instruções são divididas em duas partes:

1. Mnemônico - é uma abreviação que representa a operação a ser realizada (por exemplo, '*add*' para uma operação de soma).

2. Operandos - definem o contexto no qual a operação será realizada (por exemplo, os registradores e valor imediato)[3].

A seguir, é apresentado um exemplo do uso de instruções em um programa. Note que algumas instruções fazem referência à símbolos, sendo necessário que o montador resolva as referências antes de realizar a montagem das instruções.

```
ref1
    add r1,r2,123
    xor r3,r4
    blt r6,r1,ref1
```

[2]Uma exceção é a declaração de uma variável contendo um símbolo (referência) para um ponto do programa.

[3]Os operandos em uma instrução são separados por vírgula.

```
    ...
ref2
    ldw r4,r0,ref3
    ...
ref3
    (dado)
    ...
```

Valores numéricos utilizados para definição de operandos literais podem ser especificados em base 10 (por exemplo: 123), base 8 (0123) ou base 16 (0x123). O valor literal também pode ser um símbolo, sendo seu valor numérico resolvido pelo montador.

3.1.5 Variáveis

Variáveis são declaradas por meio de rótulos, que definem uma referência a um espaço em memória contendo a informação representada por um valor numérico. A alocação e acesso à variáveis ocorre por meio da reserva do espaço em memória para o armazenamento do conteúdo de uma variável, a nomeação dessa variável (símbolo ou referência) e por fim instruções que manipulam o conteúdo. A alocação ocorre em tempo de montagem e o acesso é feito em tempo de execução, quando as instruções do programa executam.

Dois tipos de dados são definidos para a declaração de variáveis, sendo esses uma palavra (inteiro de 16 bits) e um byte (inteiro de 8 bits). Como pode ser observado, valores numéricos para variáveis do tipo byte são precedidos pelo caractere '$'. A alocação de bytes como variáveis é feita de forma alinhada[4]. Inteiros são acessados por meio das instruções *ldw* e *stw* (para leitura e escrita, respectivamente) e bytes por instruções *ldb*, *lbu* e *stb* (as duas primeiras usadas para leitura e a última para escrita).

```
ref1
    (instruções)
    ...
var1
    123
var2
    -456
var3
    $-1
```

No exemplo é apresentada a sintaxe para a declaração de variáveis. *var1* declara uma variável do tipo inteiro com valor positivo e *var2* declara um inteiro com valor negativo. É possível realizar comparações sinalizadas e não sinalizadas com os valores de variáveis, após sua carga em registradores. *var3* define uma variável do tipo byte. O seu valor pode ser interpretado de duas formas, dependendo da instrução utilizada em uma operação de carga. Por exemplo, se a instrução *ldb* for utilizada para carga do valor, este terá o sinal preservado (no caso, -1) e a faixa representável está entre -128 e 127. Caso a instrução *lbu* for utilizada, o valor carregado não terá o sinal preservado (no caso, 255), sendo a faixa representável entre 0 e 255. Valores numéricos utilizados para definição de variáveis podem ser especificados em base 10 (por exemplo: 123), base 8 (0123) ou base 16 (0x123).

[4]O alinhamento de endereços consiste em alocar o objeto em um endereço que coincide com o início de uma palavra, usando-se um byte de *padding*.

3.1.6 Arranjos

Arranjos são declarados da mesma forma que variáveis, ou seja, por rótulos que definem uma referência para o início de um espaço contíguo em memória contendo itens de um mesmo tipo. Existem quatro tipos de arranjos possíveis (unidimensionais), sendo esses:

1. Inteiro
2. Byte
3. Reserva de espaço
4. String

Arranjos dos tipos 1 e 2 possuem seus itens separados por espaço. Em um arranjo de bytes (tipo 2), os itens são precedidos por um caracter '$'. Nos arranjos do tipo 1 e 2, é alocado um espaço contíguo de palavras ou bytes na memória, ou seja, cada item do arranjo possui 16 bits (2 bytes) para o tipo inteiro ou 8 bits (1 byte) para o tipo byte.

Nos arranjos do tipo 3 (reserva de espaço), é define-se um tamanho em bytes de espaço contíguo a ser reservado em memória. No tipo 4, uma *string* tem seus bytes alocados contiguamente em memória. A alocação de memória nos tipos 2, 3 e 4 é feita por meio de bytes alocados e alinhados em palavras. Por exemplo, no arranjo *byte_array1* temos 5 itens, sendo um item adicional (zero) alocado para o alinhamento, o que gera um total de 3 palavras (6 bytes). O mesmo é feito em arranjos do tipo 3 e 4, caso necessário.

```
ref1
    (instruções)
    ...
int_array
    123 456 -789 12345 -55
byte_array1
    $12 $34 $-56 $3
byte_array2
    [30]
byte_array3
    "hello world!"
```

É importante ressaltar que strings possuem um byte terminador implícito (zero) ao final. Isso é feito para simplificar o processo de impressão e manipulação de strings, o que é feito da mesma forma em linguagens como C, por exemplo. Valores numéricos utilizados para definição de itens ou tamanho de arranjos podem ser especificados em base 8, 10 ou 16.

3.1.7 Código objeto e desmontagem do programa

O código objeto é o resultado do processamento de um programa em linguagem de montagem para um formato binário, que pode então ser executado pela CPU. Na maioria dos casos, o código objeto poderá ser agregado a outras partes de código objeto (como bibliotecas de funções, por exemplo) para a construção de um programa executável.

A união de diversos códigos objeto, juntamente com a informação de relocação, formada por endereços em uma tabela de símbolos são utilizadas por uma ferramenta de ligação, para a construção do executável final. O processo de ligação de diversos

códigos objeto foge o escopo desse documento, que se limita ao processo de montagem e geração de código de máquina que pode ser executado diretamente por um simulador da arquitetura.

Para ilustrar o processo de montagem e geração do código objeto, será utilizado um pequeno programa, que declara duas variáveis (*var_a* e *var_b*), carrega os seus valores em registradores e soma os mesmos.

```
main
    ldw v0,zr,var_a
    ldw v1,zr,var_b
    add v0,v1
    hlt
var_a
    123
var_b
    456
```

O código gerado pelo processo de montagem possui 16 bytes, e este é apresentado abaixo. Na primeira coluna do código, temos os endereços de memória onde cada palavra do código objeto será carregado. As palavras podem ser instruções ou dados. Na segunda coluna, temos o código objeto propriamente dito, que são as palavras a serem carregadas na memória. Em ambas as colunas, os valores são representados em base 16. Na terceira coluna, é apresentado o processo inverso à montagem, conhecido como *desmontagem*. Essa informação é útil para análise do programa após a montagem, podendo ser comparado ao código fonte.

```
0000 4150    (ldw r5,r0,12)
0002 000c
0004 4160    (ldw r6,r0,14)
0006 000e
0008 0856    (add r5,r6)
000a f800    (hlt r0,r0)
000c 007b
000e 01c8
```

É importante observar que para algumas instruções são geradas duas palavras, já para outras outras apenas uma. Isso é o resultado do uso de instruções com ou sem um campo imediado (valor literal), o que depende do modo de endereçamento usado na instrução específica. As variáveis têm o seu espaço alocado e inicializado, não sendo apresentada uma informação de desmontagem, assim como é feito com os valores literais.

3.2 Exemplos

3.2.1 Inicialização de registradores

Constantes

Constantes são necessárias em diversas situações em programas de linguagem de montagem. Por exemplo, a carga de valores numéricos para o processamento, referências dentro de um programa (símbolos), entre outros. Na arquitetura TRM, não há uma instrução especial para a carga de constantes, mas isso pode ser feito em conjunto com

instruções *add* ou *or* em conjunto com o registrador *zr* (zero). Uma instrução *xor* pode ser utilizada para limpar um registrador de forma eficiente. Alguns exemplos:

```
add v0,zr,1234              ; v0 = 1234
or v1,zr,555                ; v1 = 555
add v2,zr,0x222             ; v2 = 0x222
add v3,zr,ref               ; v3 = &ref
xor v4,v4                   ; v4 = 0
```

Transferência valores entre registradores

Para cópia valores entre registradores não há instruções especiais. É possível utilizar instruções *add* ou *or* para isso:

```
add v0,v1,0                 ; v0 = v1
or v1,v2,0                  ; v1 = v2
```

A troca de valores entre dois registradores (operação *swap*) é bastante eficiente:

```
; v0 = 123, v1 = 456
xor v0,v1
xor v1,v0
xor v0,v1
; v1 = 456, v0 = 123
```

Operações com três registradores

Na arquitetura são definidas instruções com apenas dois registradores ou dois registradores e uma constante. O padrão *add r1,r2,r3*, existente em computadores como MIPS, RISC-V ou ARM pode ser emulado, no entanto. Por exemplo, para implementar a instrução *sub v0,v1,v2*, onde *v1* e *v2* são operandos fonte e *v0* o destino, que armazenará o resultado:

```
add v0,v1,0                 ; v0 = v1
sub v0,v2                   ; v0 = v0 - v2
```

3.2.2 Complemento e inversão de sinal

Não são definidas instruções específicas para operações de complemento e inversão de sinal (complemento de 2). Para essas operações, utiliza-se padrões simples. O complemento (operação *not*) pode ser sintetizado por uma operação *xor* com o valor -1. Para o complemento de 2, basta adicionar 1 ao resultado.

```
xor v0,v0,-1                ; v0 = not v0 (one's complement)
add v0,v0,1                 ; v0 = v0 + 1 (two's complement)
```

3.2.3 Controle de fluxo

Tanto operações de seleção quanto laços condicionais podem ser implementados com as mesmas operações. Testes podem ser realizados e seu resultado (verdadeiro ou falso) armazenado em registradores ou por meio de desvios condicionais. Nos exemplos serão utilizados desvios.

Seleção

As operações de seleção '<' (menor que), '>=' (maior ou igual), '==' (igual) e '! =' (diferente) são simples de implementar, pois existem instruções específicas para estes testes. Nos exemplos apresentados, o desvio ocorre caso a condição seja verdadeira. Caso a condição seja falsa, o programa segue (cláusula *else*). Assume-se que os valores *a* e *b* que estão sendo comparados, foram carregados nos registradores *v0* e *v1* respectivamente.

- a < b

```
    blt v0,v1,lessthan
else
    ...
lessthan
    ...
```

- a == b

```
    beq v0,v1,equal
else
    ...
equal
    ...
```

- a >= b

```
    bge v0,v1,greaterequal
else
    ...
greaterequal
    ...
```

- a ! = b

```
    bne v0,v1,notequal
else
    ...
notequal
    ...
```

As operações '>' (maior que) e '<=' (menor ou igual) podem ser sintetizadas pelas instruções *blt* e *bge*, bastando inverter a ordem dos operandos.

- a > b

```
    blt v1,v0,greaterthan
else
    ...
greaterthan
    ...
```

- a <= b

```
    bge v1,v0,lessequal
else
    ...
lessequal
    ...
```

Laços

Operações de repetição (laços) em linguagem e montagem possuem uma estrutura semelhante à operações de seleção, com a diferença de que 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 é necessário utilizar os comandos *break* e *continue*. Em todos os casos, as estruturas utilizadas para essas operações são semelhantes às apresentadas anteriormente.

Repetições incondicionais podem ser implementadas de acordo com o padrão apresentado a seguir. No exemplo, assume-se que uma comparação por igualdade do registrador *zr* (zero) com ele próprio seja sempre verdadeira. Esse exemplo ilustra uma construção semelhante a um laço *while (1)* { ... }.

```
loop
    ...                         ; loop body
```

```
    beq zr,zr,loop
endloop
```

Repetições condicionais podem ser implementadas com um laço incondicional, e internamente realizarem testes e condições de quebra do laço (break) ou adiantamento do teste (continue). A implementação de um laço semelhante a *while (a < b)* { ... } é apresentrada a seguir. O teste realizado para a quebra do laço é o oposto da condição de permanência.

```
loop
    bge v0,v1,endloop           ; if a >= b, break
    ...                         ; loop body
    beq zr,zr,loop
endloop
```

Para um laço do tipo *do* { ... } *while (a < b)*, basta que o teste seja realizado imediatamente antes do desvio incondicional, permitindo que o corpo do laço execute uma vez antes do teste ser feito.

3.2.4 Variáveis e arranjos

A arquitetura realiza o acesso à memória apenas com instruções de transferência de dados (*load / store*). Dessa forma, os operandos precisam ser trazidos da memória antes de serem realizadas operações lógicas e aritméticas, comparações e desvios.

Acesso à variáveis

O acesso à variáveis pode ser feito diretamente por meio de instruções *ldw* e *stw* (inteiros) ou *ldb*, *lbu* e *stb* (bytes). O acesso pode ser realizado por uma referência ou por meio de um registrador com a referência.

```
    ...
    ldw v0,zr,var1              ; v0 = var1
    ...
    add v1,zr,var1              ; v1 = &var1
    ldw v0,v1                   ; v0 = *var1
    ...
    ldb v0,zr,var2              ; v0 = var2
    ...
    add v1,zr,var2              ; v1 = &var2
    ldb v0,v1                   ; v0 = *var2
    ...
var1
    0x999                       ; (integer)
var2
    $0xaa                       ; (byte)
```

No exemplo apresentado abaixo, é feita a soma do valor de duas variáveis e o resultado é armazenado em uma terceira ($var\_c = var\_a + var\_b$). Ao todo, são realizados 3 acessos de dados à memória.

Listing 3.1: vars.s

```
1 main
2     ldw v0,zr,var_a            ; v0 = var_a
3     ldw v1,zr,var_b            ; v1 = var_b
4     add v0,v1                  ; v0 = v0 + v1
5     stw v0,zr,var_c            ; var_c = v0
6     stw v0,zr,0xf000           ; print v0
7     hlt
8
9 var_a
10    123
11 var_b
12    456
13 var_c
14    0
```

Arranjos

O acesso à arranjos pode ser feito diretamente por meio de instruções *ldw* e *stw* (inteiros) ou *ldb*, *lbu* e *stb* (bytes). O acesso pode ser realizado por uma referência em um registrador, somado a um deslocamento. Outra alternativa é utilizar um endereço base fixo (referência ao início do arranjo), somado a um deslocamento armazenado em um registador.

```
    ...
    add v3,zr,array             ; v3 = &array
    ldw v0,v3,0                 ; v0 = array[0]
    ldw v1,v3,2                 ; v1 = array[1]
    ...
    add v3,zr,array             ; v3 = &array
    ldw v0,v3                   ; v0 = *array
    add v3,v3,2                 ; v3 = &array++
    ldw v1,v3                   ; v1 = *array
    ...
    add v3,zr,0                 ; v3 = 0
    ldw v0,v3,array             ; v0 = array[0]
    add v3,v3,2                 ; v3 += 2
    ldw v1,v3,array             ; v1 = array[1]
    ...

array
    123 456 -789 111 222 -333   ; (array of integers)
```

No exemplo abaixo é realizado o acesso a um vetor de inteiros, onde cada elemento do vetor é acessado e modificado. É importante observar que o vetor é indexado por meio do registrador *v0*. Para calcular o deslocamento na memória, é necessário multiplicar o índice pelo tamanho do tipo (inteiro, 2 bytes), o que é feito com uma operação *lsl*. O limite de iterações do laço é dependente do número de elementos no vetor.

Listing 3.2: array.s

```
1 main
2     add v0,zr,0               ; v0 = 0 (i)
3     ldw v1,zr,size            ; v1 = size
4 loop
5     bge v0,v1,endloop         ; if i >= size, break
```

```
 6     lsl v2,v0,1                ; v2 = i * 2
 7     ldw v3,v2,vet              ; v3 = vet[i]
 8     sub v3,v3,5                ; v3 -= 5
 9     stw v3,v2,vet              ; vet[i] = v3
10     stw v0,zr,0xf000           ; print v0 (index)
11     stw v3,zr,0xf000           ; print v4 (value)
12     add v0,v0,1                ; i++
13     beq zr,zr,loop             ; goto loop
14 endloop
15     hlt
16
17 vet
18     0 1 2 3 4 5 6 7 8 9
19 size
20     10
```

O próximo exemplo realiza as mesmas operações, no entanto o acesso é feito por meio de ponteiros. O deslocamento na memória é feito por meio de aritmética de ponteiros, incrementando-se o endereço de memória por duas posições (inteiro). O limite de iterações do laço é dependente do endereço acessado.

Listing 3.3: arrayptr.s

```
 1 main
 2     add v0,zr,vet              ; v0 = &vet
 3     add v1,zr,endvet           ; v1 = &vet + sizeof(vet)
 4 loop
 5     bge v0,v1,endloop          ; if &vet >= &endvet, break
 6     ldw v3,v0                  ; v3 = *vet
 7     sub v3,v3,5                ; v3 -= 5
 8     stw v3,v0                  ; *vet = v3
 9     stw v0,zr,0xf000           ; print v0 (address)
10     stw v3,zr,0xf000           ; print v2 (value)
11     add v0,v0,2                ; &vet++
12     beq zr,zr,loop             ; goto loop
13 endloop
14     hlt
15
16 vet
17     0 1 2 3 4 5 6 7 8 9
18 endvet
```

No exemplo abaixo, é realizado o acesso a um arranjo de bytes, onde as instruções *ldb* e *lbu* são utilizadas. Não é necessário multiplicar o índice pelo tamanho do tipo, pois o acesso é do tipo byte.

Listing 3.4: arraybytes.s

```
 1 main
 2     add v0,zr,0                ; v0 = 0 (i)
 3     ldw v1,zr,size             ; v1 = size
 4 loop
 5     bge v0,v1,endloop          ; if i >= size, break
 6     ldb v2,v0,bytes            ; v1 = bytes[i]
 7     lbu v3,v0,bytes            ; v2 = bytes[i] (unsigned)
 8     stw v2,zr,0xf000           ; print v1
 9     stw v3,zr,0xf000           ; print v2
10     add v0,v0,1                ; i++
11     beq zr,zr,loop             ; goto loop
12 endloop
13     hlt
14
15 bytes
16     $-65 $66 $-67 $68 $-69 $5
17 size
18     6
```

Strings

O acesso a strings ocorre da mesma forma que um arranjo de bytes. No exemplo abaixo, uma string é copiada para um espaço reservado (em branco) de 30 bytes, e posteriormente a string copiada é impressa.

Listing 3.5: string.s

```
1 main
2     add v0,zr,str1              ; v0 = &str1
3     add v1,zr,str2              ; v1 = &str2
4 loop
5     ldb v2,v0                   ; v2 = *str1
6     stb v2,v1                   ; *str2 = v2
7     add v0,v0,1                 ; &str1++
8     add v1,v1,1                 ; &str2++
9     beq v2,zr,endloop           ; if v2 == 0, break
10    beq zr,zr,loop              ; goto loop
11 endloop
12
13    add v1,zr,str2              ; v1 = &str2
14 loop2
15    ldb v2,v1                   ; v2 = *str2
16    stb v2,zr,0xf000            ; print v2
17    add v1,v1,1                 ; &str2++
18    beq v2,zr,endloop2          ; if v2 == 0, break
19    beq zr,zr,loop2             ; goto loop2
20 endloop2
21    hlt
22
23 str1
24    "hello world!"
25 str2
26    [30]
```

3.2.5 A pilha

A pilha tem como objetivo fazer uso da memória livre (não utilizada pelo programa) como uma estrutura de dados gerenciável, sendo essa estrutura essencial para a implementação de aplicações modulares. É por meio do uso da pilha que torna-se possível implementar recursão, passagem de um número maior de argumentos para funções entre outros.

O uso da memória como uma pilha pode ser generalizado em duas operações: PUSH (armazena um item na pilha) e POP (retira um item da pilha). As operações sobre a pilha utilizam um registrador especial para isso (*stack pointer* - *r14* ou *sp*). Antes que a pilha possa ser utilizada, esse registrador deve ser inicializado com uma referência ao *topo da pilha*, que normalmente é o endereço da última palavra da memória. Para implementar as operações PUSH e POP, deve-se proceder da seguinte forma:

- PUSH - reservar um espaço na pilha, gravar o valor de um registrador na pilha (no exemplo, *v0*);

  ```
  sub sp,sp,2
  stw v0,sp
  ```

- POP - ler o valor da pilha e gravar em um registrador, liberar um espaço na pilha (no exemplo, *v0*).

  ```
  ldw v0,sp
  add sp,sp,2
  ```

Diversos itens podem ser colocados na pilha ou retirados de uma só vez. Para isso, deve-se reservar o espaço na pilha e acessar a mesma para as operações de leitura ou escrita por meio de um deslocamento. No exemplo abaixo, 5 registradores são colocados na pilha de uma só vez, usando-se operações de escrita após a alocação do espaço:

```
sub sp,sp,10
stw v0,sp,0
stw v1,sp,2
stw v2,sp,4
stw v3,sp,6
stw v4,sp,8
```

E os mesmos podem ser retirados da mesma forma, lendo-se os valores da pilha e desalocando o espaço.

```
ldw v4,sp,8
ldw v3,sp,6
ldw v2,sp,4
ldw v1,sp,2
ldw v0,sp,0
add sp,sp,10
```

O programador é responsável por realizar esse gerenciamento, assim como lembrar da ordem dos itens armazenados na pilha. A pilha pode ser usada, entre outras coisas, para auxiliar na implementação de programas modulares, usando-se chamadas de função.

3.2.6 Ponteiro de quadro

O ponteiro de quadro possui como objetivo permitir o acesso rápido a variáveis locais, no contexto de uma função e permitir um uso mais flexível do ponteiro de pilha. O ponteiro de quadro (*frame pointer* - *r13*, *fp* ou *v8*) não precisa ser utilizado para essa finalidade, podendo ser apenas mais um registrador disponível ao programador, ou seja, seu uso é opcional.

Se for necessário alocar uma região de memória na pilha para uso no contexto de uma função, pode-se configurar o registrador *fp* com o valor de *sp* antes dessa alocação. Dessa maneira, é possível acessar os dados por meio de uma referência que independe do valor de *sp*, que pode estar sendo utilizado para o processamento de valores intermediários que residem na pilha. Caso for utilizado, esse também é uma forma eficiente de desalocar dados da própria pilha, uma vez que pode-se copiar o valor do registrador *fp* para o *sp* no término de uma função.

3.2.7 Convenções para chamadas de função

Algumas convenções são definidas para chamadas de função. Essas convenções devem ser seguidas para que haja compatibilidade na implementação de aplicações e bibliotecas de funções.

- *a0* - *a3* são registradores de argumento e temporários e são usados para passagem de parâmetros, operações de processamento e retorno de valores.
- *lr* deve ser salvo na pilha caso a função chamar outra função, uma vez que este mantem o endereço de retorno da função chamadora.

- Os registradores *a0* - *a3* não são salvos na função chamada. Se for necessário a sua preservação, é papel da função chamadora fazer isso antes de realizar uma chamada.
- Os registradores *v0* - *v7*, *fp*, *sp* e *lr* devem ser preservados na função chamada, caso forem modificados. Uma função que modifica tais registradores deve salvar seus valores na no início da função e restaurar os mesmos no final, antes do retorno. Apenas registradores modificados precisam ser preservados.
- Funções simples que requerem apenas 4 registradores de trabalho (*a0* - *a3*) e que não chamam outras funções não precisam salvar registradores na pilha, desde que os registradores preservados não forem modificados.

Procedimento para chamadas de função

- Na função chamadora, são realizados os seguintes passos: a) Passar os argumentos nos registradores *a0* - *a3*; b) Configurar *lr* com o endereço de retorno; c) Saltar para a função; d) No retorno, ler os valores retornados de *a0* - *a3*.
- Na função chamada, são realizados os seguintes passos: a) Salvar os registradores *v0* - *v7*, *fp* e *lr* na pilha, conforme necessário; b) (opcional) Copiar *sp* para *fp*; c) (opcional) Alocar uma região da pilha para variáveis locais (acessíveis via *fp*); d) Realizar o processamento; e) Passar os valores de retorno em *a0* - *a3*; f) (opcional) Desalocar a pilha; g) Restaurar os registradores salvos (*lr*, *fp*, *v7* - *v0*) da pilha; h) Retornar para a função chamadora usando *lr*.

O exemplo abaixo ilustra o procedimento. Na função chamada não é feito o salvamento dos registradores *v0* - *v7*, tampouco *fp* ou *lr* pois os mesmos não são modificados. Os argumentos são passados por valor.

Listing 3.6: fcall.s

```
1 main
2     ldw a0,zr,a                ; a0 = a
3     ldw a1,zr,b                ; a1 = b
4     add lr,zr,ret1             ; lr = &ret1
5     beq zr,zr,func1            ; func1()
6 ret1
7     stw a0,zr,res              ; res = a0
8     stw a0,zr,0xf000           ; print a0
9     hlt
10
11 a
12    123
13 b
14    456
15 res
16    0
17
18 func1
19    add a0,a1                  ; a0 = a0 + a1
20    beq zr,lr                  ; return
```

No exemplo a seguir, os parâmetros são passados por referência. Dentro da função chamada, os valores são acessados por ponteiros.

Listing 3.7: fcallbyref.s

```
1 main
2     add a0,zr,a                ; a0 = &a
```

```
3     add a1,zr,b                ; a1 = &b
4     add a2,zr,res              ; a2 = &res
5     add lr,zr,ret1             ; lr = &ret1
6     beq zr,zr,func1            ; func1()
7 ret1
8     ldw a0,zr,res              ; a0 = res (not needed!)
9     stw a0,zr,0xf000           ; print a0
10    hlt
11
12 a
13    123
14 b
15    456
16 res
17    0
18
19 func1
20    ldw a0,a0                  ; a0 = a
21    ldw a1,a1                  ; a1 = b
22    add a0,a1                  ; a0 = a0 + a1
23    stw a0,a2                  ; res = a0
24    beq zr,lr                  ; return
```

É possível realizar a passagem de mais do que quatro parâmetros para uma função. Para isso, deve-se colocar os argumentos na pilha na função chamadora em sua ordem inversa, e passar uma referência para o endereço do primeiro parâmetro como argumento. Os argumentos podem ser acessados na função chamada por valor ou referência. Cabe à função chamadora gerenciar a pilha, desalocando os valores previamente alocados.

No exemplo abaixo, a função principal aloca o espaço de 6 bytes na pilha, para passagem de 3 parâmetros. Os parâmetros são passados em ordem inversa, e o ponteiro de pilha é copiado para o registrador *a0*. Na função chamada, os parâmetros são acessados na pilha por meio de uma referência.

Listing 3.8: fcallstack.s

```
1 main
2     ldw v0,zr,a                ; v0 = a
3     ldw v1,zr,b                ; v1 = b
4     ldw v2,zr,res              ; v2 = res
5     sub sp,sp,6                ; push v2, v1, v0
6     stw v2,sp,4
7     stw v1,sp,2
8     stw v0,sp,0
9     add a0,sp,0                ; a0 = sp
10    add lr,zr,ret1             ; lr = &ret1
11    beq zr,zr,func1            ; func1()
12 ret1
13    ldw v0,sp,0                ; pop v0, v1, v2
14    ldw v1,sp,2
15    ldw v2,sp,4
16    add sp,sp,6
17    stw v2,zr,res              ; res = v2
18    stw v2,zr,0xf000
19    hlt
20
21 a
22    123
23 b
24    456
25 res
26    0
27
28 func1
29    ldw a2,a0,0                ; a2 = a
30    ldw a3,a0,2                ; a3 = b
31    add a2,a3                  ; a2 = a2 + a3
32    stw a2,a0,4                ; res = a2
```

```
33      beq zr,lr                   ; return
```

3.2.8 Funções aritméticas

A arquitetura TRM não possui suporte nativo para operações de multiplicação, divisão ou resto da divisão (módulo). Essas operações precisam ser sintetizadas por funções de biblioteca. Abaixo são apresentadas implementações eficientes para essas operações[5]. É importante observar que as implementações poderiam ser otimizadas, substituindo-se referências a constantes (o número 1, por exemplo) por um registrador com esse valor, entre outras melhorias. Estas são apresentadas dessa forma por motivos de simplicidade e melhor legibilidade do código.

Multiplicação

Para a multiplicação, devem ser passados como parâmetros dois números (multiplicando e multiplicador) nos registradores *a0* e *a1* respectivamente, sendo o produto resultante armazenado no registrador *a0*. Os operandos podem ser ou não sinalizados.

Listing 3.9: mul.s

```
1  ; function: multiply two numbers
2  ; arguments: a0 (multiplicand) and a1 (multiplier)
3  ; result: a0 (product)
4
5  mul
6       xor a2,a2
7  repmul01
8       beq a1,zr,endmul01
9       and r4,a1,1
10      beq r4,zr,skipmul01
11      add a2,a0
12 skipmul01
13      lsl a0,a0,1
14      lsr a1,a1,1
15      beq zr,zr,repmul01
16 endmul01
17      add a0,a2,0
18      beq zr,lr
```

Divisão e módulo

Para a divisão e módulo, devem ser passados como parâmetros dois números (numerador e denominador) nos registradores *a0* e *a1* respectivamente, sendo o quociente resultante armazenado no registrador *a0* e o módulo no registrador *a1*. Os operandos podem ser ou não sinalizados, devendo ser utilizada a versão correta da função de acordo.

Listing 3.10: div.s

```
1  ; function: divide two numbers
2  ; dependencies: udiv
3  ; arguments: a0 (numerator) and a1 (denominator)
4  ; result: a0 (quotient) and a1 (modulus)
5
6  div
7       xor a2,a2
8       bge a0,zr,skipdiv01
9       xor a0,a0,-1
```

[5] A implementação das operações de divisão e módulo ocorre na mesma função, ou seja, a divisão já retorna o quociente e o módulo.

```
10      add a0,a0,1
11      add a2,zr,1
12  skipdiv01
13      bge a1,zr,skipdiv02
14      xor a1,a1,-1
15      add a1,a1,1
16      xor a2,a2,1
17  skipdiv02
18      sub sp,sp,4
19      stw a2,sp,0
20      stw lr,sp,2
21      add lr,zr,retudiv01
22      beq zr,zr,udiv
23  retudiv01
24      ldw lr,sp,2
25      ldw a2,sp,0
26      add sp,sp,4
27      beq a2,zr,skipdiv03
28      xor a0,a0,-1
29      add a0,a0,1
30  skipdiv03
31      beq zr,lr
32
33
34  ; function: unsigned divide two numbers
35  ; arguments: a0 (numerator) and a1 (denominator)
36  ; result: a0 (quotient) and a1 (modulus)
37
38  udiv
39      xor a2,a2
40      add a3,zr,1
41      bne a1,zr,repudiv01
42      xor a0,a0
43      beq zr,lr
44  repudiv01
45      blt a1,zr,repudiv02
46      lsl a1,a1,1
47      lsl a3,a3,1
48      beq zr,zr,repudiv01
49  repudiv02
50      beq a3,zr,rependudiv02
51      bbl a0,a1,skipudiv01
52      sub a0,a1
53      add a2,a3
54  skipudiv01
55      lsr a1,a1,1
56      lsr a3,a3,1
57      beq zr,zr,repudiv02
58  rependudiv02
59      add a1,a0,0
60      add a0,a2,0
61      beq zr,lr
```

Capítulo 4

Ferramenta de montagem e simulação

4.1 Exemplos

Diversos exemplos para uso com a ferramenta de montagem e simulação estão disponíveis no diretório *trm/examples/*. Todos são programas descritos com a sintaxe da linguagem de montagem apresentada no capítulo anterior.

4.2 Compilação da ferramenta

O código fonte da ferramenta de montagem e simulação está disponível no diretório *trm/java/*. Para compilar a ferramenta, é necessário ter o pacote *jdk-11* (ou posterior) instalado em seu sistema, além de ferramentas básicas como *make*. O processo é simples de ser realizado a partir da linha de comando:

```
$ cd trm/java
$ make
$ cd ..
```

Existem duas versões da ferramenta. A primeira possui uma interface gráfica que integra um simples editor, montador e simulador da arquitetura. A segunda possui uma interface que pode ser controlada por linha de comando. Para carregar a ferramenta gráfica, utilize o comando:

```
$ java -jar java/TrmGUI.jar
```

Irá ser apresentada a interface gráfica da ferramenta. As funcionalidades de edição, montagem e simulação podem ser selecionada por diferentes *tabs*. A ferramenta pode de linha de comando pode ser invocada com:

```
$ java -jar java/Trm.jar
```

Será apresentado um conjunto de parâmetros a serem passados à ferramenta para o uso de diferentes modos de operação da mesma. É possível realizar apenas o processo de montagem e geração de um arquivo com o código objeto (modo 'a'), simulação do código objeto gerado anteriormente, a partir de um arquivo (modo 's'), os processos combinados de montagem e simulação, sem a geração de um arquivo intermediário (modo 'r') e por último um sistema de depuração e execução interativa (modo 'd').

```
Syntax: java -jar Trm.jar <mode> <source> <output>
<mode>   - a (assemble), s (simulation), r (run), d (debug)
<source> - assembly (.s) source code for 'a', 'r' and 'd' modes,
           object code (.trm) for 's' mode
<output> - object code (.trm) for 'a' mode
```

4.3 Montagem

Na ferramenta gráfica, o processo de montagem é simples, bastando inicialmente carregar um código fonte no editor de texto (*Code Editor*)[1] e selecionar a tab *Assembler and Simulator*. Para montar o código, seleciona-se a opção *Assemble/Exec*, acessível por meio do menu superior ou botão no lado direito da interface. Após a montagem do programa, será apresentada a listagem do código objeto, tabela de símbolos e opções para simulação.

Para a ferramenta de linha de comando, o processo de montagem de um programa pode ser realizado fornecendo-se à ferramenta o código fonte e um arquivo de saída. No arquivo de saída será armazenado o código objeto gerado e eventuais erros. Exemplo:

```
$ java -jar java/Trm.jar a examples/01-add.s 01-add.txt
```

Caso não seja fornecido um arquivo de saída, a mesma será redirecionada para a saída padrão. Para o código fonte do exemplo, será gerada essa saída:

```
0000 0950    (add r5,r0,123)
0002 007b
0004 0955    (add r5,r5,456)
0006 01c8
0008 4950    (stw r5,r0,61440)
000a f000
000c 4160    (ldw r6,r0,28)
000e 001c
0010 4170    (ldw r7,r0,30)
0012 001e
0014 0867    (add r6,r7)
0016 4960    (stw r6,r0,61440)
0018 f000
001a f800    (hlt r0,r0)
001c 007b
001e 01c8
```

4.4 Simulação

A partir do código objeto gerado no passo anterior, pode-se realizar uma simulação do mesmo. O simulador irá verificar se o código objeto está correto antes de iniciar a simulação.

[1] Esse processo pode ser realizado selecionando-se a opção *File* - *Open* ou colando-se um texto no editor diretamente.

Na ferramenta gráfica, a simulação pode ser realizada de uma só vez (opção *Run*), passo a passo de forma automática (opção *Auto Step*) ou manualmente (opção *Stop/Step*). O processo de simulação na ferramenta gráfica já inclui depuração. A depuração inclui um acompanhamento da execução, onde é feito um apontamento da instrução na listagem do código objeto antes de sua execução, bem como referências a símbolos. Após a execução de uma instrução, é atualizado o banco de registradores, o contador de programa, a memória (caso a última instrução referencie a mesma) e é feito o apontamento da próxima instrução a ser executada.

Na ferramenta por linha de comando, a simulação pode ser realizada passando-se como argumento a opção de simulação e um arquivo com o código objeto.

```
$ java -jar java/Trm.jar s 01-add.txt
[program (code + data): 32 bytes]
[memory size: 61440]
579
579
[halt]
8 cycles
```

4.5 Modo combinado (montagem e simulação, linha de comando)

O modo de execução de programas combinado é composto pela montagem e simulação em um único passo e existe na ferramenta de linha de comando. Esse modo pode ser útil quando se deseja desenvolver uma aplicação, pois evita-se um passo intermediário. Para invocar o modo combinado para execução do mesmo exemplo anterior, pode ser utilizado o seguinte comando:

```
$ java -jar java/Trm.jar r examples/01-add.s
```

O resultado do comando será o seguinte:

```
...
0014 0867    (add r6,r7)
0016 4960    (stw r6,r0,61440)
0018 f000
001a f800    (hlt r0,r0)
001c 007b
001e 01c8
[program (code + data): 32 bytes]
[memory size: 61440]
579
579
[halt]
8 cycles
```

4.6 Modo depuração (linha de comando)

No modo depuração, assim como no modo combinado, são realizados os processos de montagem e simulação do código em um único passo (ferramenta de linha de comando).

A diferença do modo depuração para o modo combinado é que, o segundo executa o programa até o final já o primeiro, permite que o usuário execute o programa de forma interativa, instrução por instrução. Além disso, esse modo possui recursos como endereço para parada (*breakpoint*), reinicio do programa, listagem do programa, listagem da tabela de símbolos e visualização da memória (*memory dump*). Para o programa usado nos exemplos anteriores, o processo é iniciado com o comando:

```
$ java -jar java/Trm.jar d examples/01-add.s
```

O resultado do comando será o seguinte:

```
[program (code + data): 32 bytes]
[memory size: 61440]
pc: 0000, instruction: add r5,r0,123
 r0: [0000]  r1: [0000]  r2: [0000]  r3: [0000]
 r4: [0000]  r5: [0000]  r6: [0000]  r7: [0000]
 r8: [0000]  r9: [0000] r10: [0000] r11: [0000]
r12: [0000] r13: [0000] r14: [f000] r15: [0000]
q: quit; r: run; b: breakpoint; t: reset program; ENTER: next;
l: program list; s: symbol table; d: mem dump
```

4.7 Serviços do simulador

Como parte das funcionalidades do simulador, está implementado um pequeno conjunto de serviços, que pode ser utilizado para entrada e saída de dados em diferentes formatos. Os serviços de saída são acessados por meio de uma operação de escrita na memória (*stw*) e os serviços de entrada por meio de uma leitura (*ldw*). Os serviços podem ser utilizados por meio de instruções no formato regular ou imediato. Exemplos:

```
    add v0,zr,1234              ; v0 = 1234
    add v1,zr,65                ; v0 = 'A'
    stw v0,zr,0xf000            ; write int (with line feed)
    stw v1,zr,0xf006            ; write char
    stw v0,zr,0xf00c            ; write hex (with line feed)
    ...
    ldw v0,zr,0xf010            ; read int
    add v0,zr,buf               ; v0 = &buf
    ldw v0,zr,0xf018            ; read string
    add v0,zr,buf               ; v0 = &buf
    stw v0,zr,0xf008            ; write string
```

Em todas as operações de saída, o primeiro registrador da instrução deve conter o valor a ser apresentado na saída, salvo operações de impressão de *strings*, que devem conter o endereço de memória onde está a mesma. Em operações de leitura, o primeiro registrador da instrução conterá o valor lido da entrada de dados do usuário. Leituras de *strings* devem ter armazenado nesse registrador o endereço de memória onde será armazenada a entrada de dados do usuário. A tabela abaixo apresenta os endereços de cada serviço, bem como a operação que deve ser realizada (escrita (W), instrução *stw* ou leitura (R), instrução *ldw*). A tabela abaixo apresenta os serviços definidos no simulador.

Endereço	**Serviço**	**Operação**
0xf000	write int (with line feed)	W
0xf002	write int	W
0xf004	write char (with line feed)	W
0xf006	write char	W
0xf008	write string (with line feed)	W
0xf00a	write string	W
0xf00c	write hex (with line feed)	W
0xf00e	write hex	W
0xf010	read int	R
0xf014	read char	R
0xf018	read string	R
0xf01c	read hex	R