Capítulo 4

O Conjunto de Instruções do Processador

O conjunto de instruções é um dos pontos centrais na arquitetura de um processador. Vários aspectos na definição e implementação da arquitetura são influenciados pelas características do conjunto de instruções. Por exemplo, as operações realizadas pela unidade lógica e aritmética, o número e função dos registradores e a estrutura de interconexão dos componentes da seção de processamento. Além disso, as operações básicas que acontecem dentro da seção de processamento dependem das instruções que devem ser executadas. O conjunto de instruções afeta o projeto da seção de controle. A sua estrutura e a sua complexidade são determinadas diretamente pelas características do conjunto de instruções.

Este capítulo discute os principais aspectos de um conjunto de instruções, como tipos de operações, operandos e modos de endereçamento.

4.1 Conjunto de Instruções no Contexto de *Software*

A Figura 4.1 situa o conjunto de instruções do processador dentro dos diversos níveis de *software* existentes em um sistema de computação.

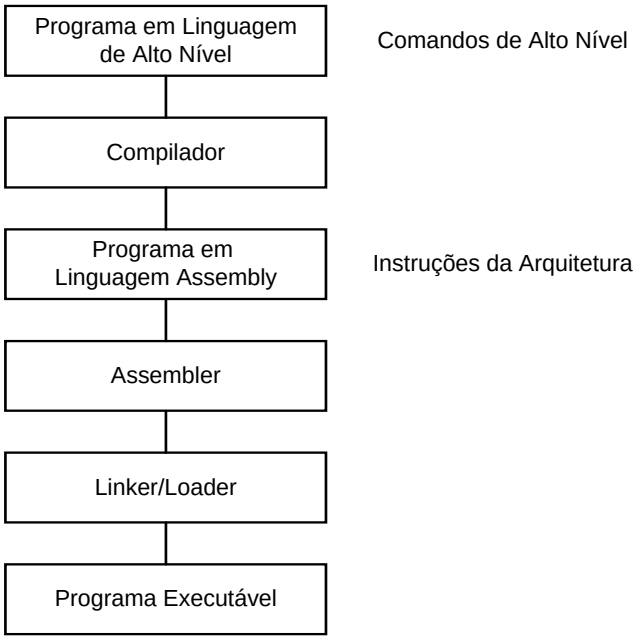


Figura 4.1. *O conjunto de instruções dentro do contexto de software.*

Em geral, os programas são desenvolvidos em uma linguagem de alto nível como FORTRAN, Pascal ou C. O compilador traduz o programa de alto nível em uma seqüência de **instruções de processador**. O resultado desta tradução é o programa em **linguagem de montagem** ou **linguagem de máquina (***assembly language***)**. A linguagem de montagem é uma forma de representar textualmente as instruções oferecidas pela arquitetura. Cada arquitetura possui uma linguagem de montagem particular. No programa em linguagem de montagem, as instruções são representadas através de mnemônicos, que associam o nome da instrução à sua função, por exemplo, `ADD` ou `SUB`, isto é soma e subtração, respectivamente.

O programa em linguagem de montagem é convertido para um programa em **código objeto** pelo **montador** (*assembler*). O montador traduz diretamente uma instrução da forma textual para a forma de código binário. É sob a forma binária que a instrução é carregada na memória e interpretada pelo processador.

Programas complexos são normalmente estruturados em módulos. Cada módulo é compilado separadamente e submetido ao montador, gerando diversos módulos em código objeto. Estes módulos são reunidos pelo **ligador** (*linker*), resultando finalmente no programa executável que é carregado na memória.

O conjunto de instruções de uma arquitetura se distingue através de diversas características. As principais características de um conjunto de instruções são: tipos de instruções e operandos, número e localização dos operandos em instruções aritméticas e lógicas, modos de endereçamento para acesso aos dados na memória, e o formato dos códigos de instrução. Estes aspectos são analisados a seguir.

4.2 Tipos de Instruções e de Operandos

As instruções oferecidas por uma arquitetura podem ser classificadas em categorias, de acordo com o tipo de operação que realizam. Em geral, uma arquitetura fornece pelo menos três categorias de instruções básicas:

- **instruções aritméticas e lógicas**: são as instruções que realizam operações aritméticas sobre números inteiros (adição, subtração) e operações lógicas *bit*-a-*bit* (*AND* , *OR*);
- **instruções de movimentação de dados**: instruções que transferem dados entre os registradores ou entre os registradores e a memória principal;
- **instruções de transferência de controle**: instruções de desvio e de chamada de rotina, que transferem a execução para uma determinada instrução dentro do código do programa.

Várias arquiteturas oferecem outras categorias de instruções, voltadas para operações especializadas. Dentre estas, podemos citar:

- **instruções de ponto flutuante**: instruções que realizam operações aritméticas sobre números com ponto flutuante;
- **instruções decimais**: instruções que realizam operações aritméticas sobre números decimais codificados em binário (BCD – *Binary Coded Decimal*);
- **instruções de manipulação de *bits***: instruções para testar ou atribuir o valor de um *bit*;
- **instruções de manipulação de *strings***: instruções que realizam operações sobre cadeias de caracteres (*strings*), tais como movimentação, comparação ou ainda procura de um caracter dentro de uma string.

Existem muitas diferenças entre as arquiteturas quanto às categorias de instruções oferecidas. Arquiteturas de uso geral oferecem a maioria das categorias relacionadas anteriormente. Arquiteturas destinadas para uma aplicação específica podem oferecer outros tipos de instruções, especializadas para aquela aplicação. Um exemplo seria uma arquitetura voltada para processamento gráfico, que ofereceria instruções para realizar operações sobre *pixels*.

Os tipos de operandos que podem ser diretamente manipulados por uma arquitetura dependem, é claro, dos tipos de instruções oferecidas. A Figura 4.2 mostra como os principais tipos de dados são normalmente representados em uma arquitetura de uso geral.

A Figura 4.2(a) mostra a representação de inteiros, neste exemplo particular, inteiros com 32 *bits*. Números inteiros podem ser representados com ou sem sinal. Em um número inteiro com sinal, o *bit* mais significativo é reservado para indicar o estado do sinal (positivo ou negativo). Números inteiros sem sinal assumem apenas valores positivos. Algumas arquiteturas oferecem instruções específicas para aritmética com ou sem sinal. Estas instruções diferem no modo como são alterados os *bits* do registrador de estado associado a ALU. Algumas linguagens de programação tornam visível para o programador esta distinção entre inteiros com ou sem sinal. Na linguagem C, por exemplo, uma variável declarada do tipo `int` é representada por um inteiro com sinal. Ao contrário, variáveis do tipo `unsigned int` são representadas por inteiros sem sinal, sendo normalmente usadas para indexar elementos de vetores.

A Figura 4.2(b) mostra a representação de números com ponto flutuante, com precisão simples e dupla. A diferença entre precisões está no número de *bits* usados para representar a mantissa e o expoente. Atualmente, a maioria das arquiteturas que operam números com ponto flutuante obedecem a um padrão, denominado IEEE 754, que define a

representação e um conjunto de operações aritméticas e lógicas para números com ponto flutuante.

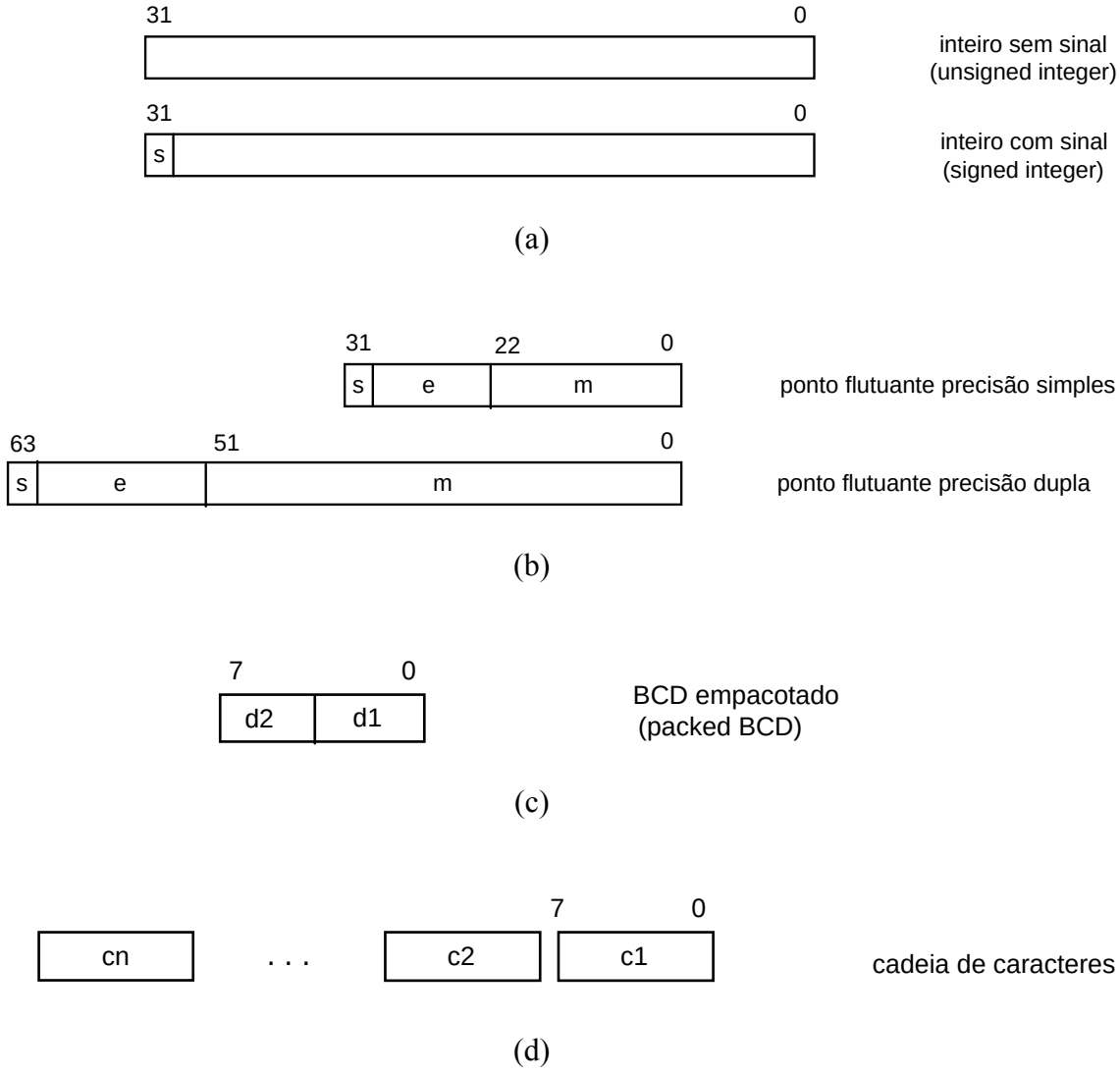


Figura 4.2. *Representação dos tipos de operandos mais comuns.*

A Figura 4.2(c) mostra a representação de números BCD empacotados (*packed Binary Coded Decimal*). Nesta representação, dois dígitos decimais codificados em binário são representados dentro de um *byte*, cada dígito sendo codificado em quatro *bits* do *byte*. Finalmente, a Figura 4.2(d) mostra a representação de cadeias de caracteres, onde cada *byte* dentro de uma seqüência de *bytes* codifica um caracter segundo um certo padrão (por exemplo, o padrão ASCII).

4.3 Número e Localização dos Operandos

Uma outra característica de um conjunto de instruções é o número de operandos explicitamente indicados em uma instrução aritmética ou lógica. Em algumas arquiteturas, estas instruções referenciam explicitamente três operandos, dois **operandos-fonte** e um **operando-destino**, como por exemplo, em

```
ADD     R1, R2, R3
```

onde R1 e R2 são os operandos-fonte e R3 é o operando-destino. Em outras arquiteturas, instruções aritméticas/lógicas especificam apenas dois operandos. Neste caso, um dos operandos-fonte é também o operando-destino. Por exemplo, na instrução

```
ADD     R1, R2
```

R2 contém um dos operandos-fonte e também é usado como operando-destino.

Quanto à localização dos operandos especificados por uma instrução aritmética/lógica, podemos encontrar arquiteturas onde podem ser realizados acessos aos operandos diretamente a partir da memória principal. Por exemplo, nestas arquiteturas podemos ter instruções tais como:

```
ADD     M1,R1,R2
ADD     M1,M2,R1
ADD     M1,M2,M3
```

onde `M1`, `M2` e `M3` são endereços de locações de memória. Em um outro extremo, existem arquiteturas onde todos os operandos encontram-se apenas em registradores. As instruções aritméticas/lógicas são todas do tipo:

```
ADD     R1,R2,R3
ADD     R1,R2
```

A partir do número de operandos explicitamente referenciados e da localização destes operandos, podemos classificar as arquiteturas nos seguintes tipos:

- **arquiteturas memória-memória**: as instruções aritméticas/lógicas usam três operandos e todos os operandos podem estar na memória;
- **arquiteturas registrador-memória**: as instruções aritméticas/lógicas usam dois operandos, sendo que apenas um deles pode residir na memória;
- **arquiteturas registrador-registrador**: as instruções aritméticas/lógicas usam três operandos, todos em registradores. Neste caso, apenas duas instruções acessam diretamente a memória: `LOAD` e `STORE`. A instrução `LOAD` carrega em um registrador um dado armazenado na memória e instrução `STORE` armazena na memória o conteúdo de um registrador.

Arquiteturas memória-memória e registrador-memória apresentam como vantagem um menor número de instruções no código do programa, já que não é necessário carregar previamente em registradores os operandos-fonte de uma instrução aritmética/lógica, como acontece em uma arquitetura registrador-registrador. Por outro lado, a existência de instruções aritméticas/lógicas mais poderosas torna mais complexa a implementação da arquitetura. As arquiteturas Intel 80x86 e Motorola MC680x0 são do tipo registrador-memória. Dentre as arquiteturas memória-memória podemos citar o DEC VAX 11.

4.4 Modos de Endereçamento

Os operandos de uma instrução podem encontrar-se em registradores, na memória principal ou ainda embutidos na própria instrução. O **modo de endereçamento** refere-se à maneira como uma instrução especifica a localização dos seus operandos. Existem três modos de endereçamento básicos:

- **modo registrador**: a instrução indica o número de um registrador de dados onde se encontra um operando (fonte ou destino);
- **modo imediato**: a instrução referencia um operando que se encontra dentro do próprio código da instrução;
- **modo implícito**: a localização do operando não está explicitamente indicada na instrução. Por exemplo, nas chamadas **arquiteturas acumulador**, um dos operandos-fonte e o operando-destino nas instruções aritméticas/lógicas encontra-se sempre em um registrador especial, o **acumulador**. Assim, não é necessário que este registrador seja explicitamente referenciado pela instrução.

A Figura 4.3 mostra exemplos de instruções que usam os modos de endereçamento implícito, registrador e imediato.

Modo	**Exemplo**	**Significado**
Implícito	`ADD R1`	Ac ← Ac + R1
Registrador	`ADD R1,R2`	R2 ← R1 + R2
Imediato	`ADD R1,#4`	R1 ← R1 + 4

Figura 4.3. *Exemplos de uso dos modos de endereçamento implícito, registrador e imediato.*

Os modos de endereçamento citados referenciam apenas operandos que se encontram em registradores ou na instrução. Existem ainda os modos de endereçamento usados para referenciar dados armazenados na memória principal. Entre as diferentes arquiteturas, existe uma enorme variedade de modos de endereçamento referentes à memória principal, e que formam, na realidade, uma classe de modos de endereçamento à parte.

Um modo de endereçamento referente à memória indica como deve ser obtido o endereço da locação de memória onde se encontra o dado que será acessado. Este endereço é chamado **endereço efetivo**. Apesar da variedade mencionada, é possível identificar alguns modos de endereçamento referentes à memória que são oferecidos pela maioria das arquiteturas. Estes modos de endereçamento mais comuns estão relacionados na Figura 4.4.

Modo	Exemplo	Significado	Uso
Direto	`ADD (100),R1`	R1 ← M[100] + R1	acesso a variáveis estáticas
Indireto	`ADD (R1),R2`	R2 ← M[R1] + R2	acesso via ponteiros
Relativo à base	`ADD 100(R1),R2`	R2 ← M[100+R1] + R2	acesso a elementos em estruturas
Indexado	`ADD (R1+R2),R3`	R3 ← M[R1+R2] + R3	acesso a elementos em um vetor

Figura 4.4. *Modos de endereçamento à memória mais comuns.*

No **modo direto**, o endereço efetivo é um valor imediato contido no código da instrução. Por exemplo, na instrução `ADD (100),R1`, um dos operandos encontra-se na locação de memória com endereço 100. O modo de endereçamento direto é usado principalmente no acesso às **variáveis estáticas** de um programa, cujo endereço em memória pode ser determinado durante a compilação do programa.

No **modo indireto**, o endereço efetivo encontra-se em um registrador. Por exemplo, na instrução `ADD (R1),R2`, um dos operandos encontra-se na locação de memória cujo endereço está no registrador R1. Ou seja, o operando na memória é indicado indiretamente, através de um registrador que contém o endereço efetivo. Este modo de endereçamento é usado no acesso a **variáveis dinâmicas**, cujo endereço na memória é conhecido apenas durante a execução do programa. O acesso a uma variável dinâmica é realizado através de um ponteiro, que nada mais é do que o endereço da variável. Para realizar o acesso à variável dinâmica, o ponteiro é carregado em um registrador, e a instrução que acessa a variável usa este registrador com o modo de endereçamento indireto.

No **modo relativo à base**, o endereço efetivo é a soma do conteúdo de um registrador, chamado **endereço-base**, com um valor imediato contido na instrução, chamado **deslocamento**. Por exemplo, na instrução `ADD 100(R1),R2`, R1 contém o endereço-base e 100 é o deslocamento. O endereço efetivo do operando em memória é a soma do conteúdo

de R1 com o valor 100. O modo relativo à base é usado no acesso a componentes de variáveis dinâmicas estruturadas (por exemplo, *record* em Pascal ou *struct* em C). A Figura 4.5 mostra como é calculado o endereço efetivo no modo de endereçamento relativo à base.

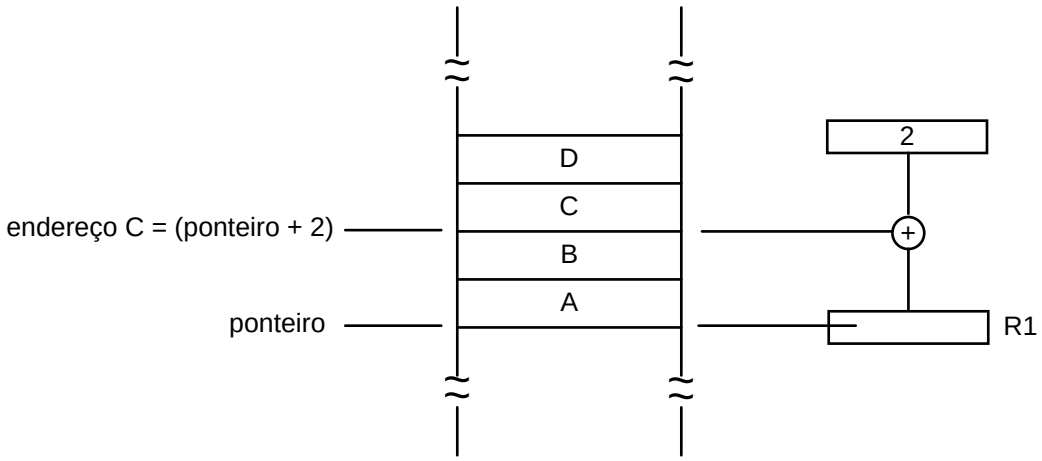


Figura 4.5. *Acesso a estruturas dinâmicas com o modo de endereçamento relativo à base.*

A figura mostra a localização na memória de uma estrutura com quatro campos A, B, C e D. O endereço inicial da estrutura é indicado por um ponteiro, que se torna conhecido apenas durante a execução do programa. No entanto, a posição de cada campo em relação ao início da estrutura é fixo, sendo conhecido durante a compilação. O endereço de um campo é obtido somando-se a posição do campo (o deslocamento) ao ponteiro que indica o início da estrutura (o endereço-base). Por exemplo, na Figura 4.5, para somar um valor ao campo C, o compilador pode usar a instrução `ADD 2(R1),R2`, precedida de uma instrução para carregar em R1 o endereço-base da estrutura.

No **modo indexado**, o endereço efetivo é dado pela soma de um **índice** com um endereço-base, ambos armazenados em registradores. Por exemplo, na instrução `ADD (R1+R2),R3`, R1 contém o endereço-base, e R2 o índice. O modo indexado é normalmente usado no acesso aos elementos de um vetor. A Figura 4.6 mostra como é calculado o endereço efetivo no modo de endereçamento indexado.

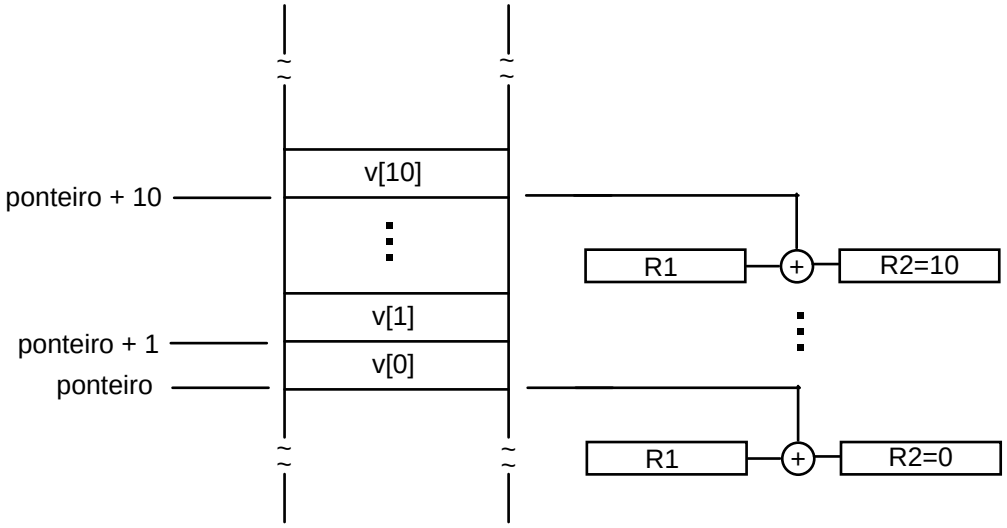


Figura 4.6. *Acesso aos elementos de um vetor com o modo de endereçamento indexado.*

A Figura 4.6 representa a localização na memória de um vetor V. Um ponteiro indica o endereço-base do vetor, onde se encontra o primeiro elemento. O endereço de cada elemento é obtido somando o índice do elemento ao endereço-base. Para realizar o acesso seqüencialmente os elementos do vetor, o índice é inicialmente carregado no registrador com o valor 0. O índice é então incrementado dentro de um *loop* após o acesso a cada elemento. Por exemplo, para somar um valor em registrador aos elementos do vetor, o compilador pode usar as seguintes instruções em um *loop*:

```
ADD    R1,(R2+R3)
ADD    1,R3
```

onde R1 contém o valor a ser somado, R2 contém o ponteiro para o vetor e R3 é o registrador com o índice, com valor inicial 0.

4.5 Formatos de Instrução

Como mencionado no início deste capítulo, as instruções de um programa compilado são armazenadas na memória sob a forma de um código em binário, ou **código de instrução**. Um código de instrução é logicamente formado por **campos de *bits***, que contém as informações necessárias à execução da instrução. Estes campos de *bits* indicam, por exemplo, qual a operação a ser realizada e quais os operandos a serem usados. A Figura 4.7 mostra um exemplo de código de instrução com seus campos de *bits*.

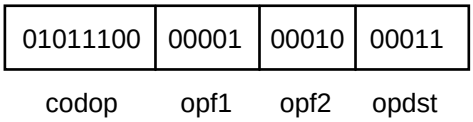


Figura 4.7. *Código de instrução e seus campos de bits.*

Neste código de instrução, o campo codop contém o código da operação a ser realizada, enquanto os campos opf1, opf2 e opdst indicam os operandos fonte e destino, respectivamente. Suponha que o código 01011100 no campo codop indique uma operação de adição, e que os valores 00001, 00010 e 00011 nos campos opf1, opf2 e opdst indiquem os registradores R1, R2 e R3, respectivamente. Assim, este código de instrução é a representação binária da instrução

`ADD  R1,R2,R3`.

O **formato de instrução** refere-se às características do código de instrução tais como tamanho do código, tipos de campos de *bits* e localização dos campos de *bits* dentro do código. Uma arquitetura se caracteriza por apresentar instruções com **formato irregular** ou com **formato regular.** No primeiro caso**,** as instruções podem apresentar códigos com tamanhos diferentes, e um certo campo de *bits* pode ocupar posições diferentes nas instruções onde aparece. Em uma arquitetura com instruções regulares**,** todos os códigos de instrução possuem o mesmo tamanho, e um certo campo de *bits* sempre ocupa a mesma posição nas instruções onde aparece.

As arquiteturas com formatos de instrução irregular possibilitam programas com menor tamanho de código executável. Isto acontece porque aqueles campos de *bits* não necessários a uma instrução são eliminados, economizando espaço de armazenamento na memória.

Por outro lado, arquiteturas com formatos de instrução regular apresentam uma grande vantagem quanto à simplicidade no acesso às instruções. Se todas as instruções possuem um tamanho de *n bits*, basta que o processador realize um único acesso de *n bits* à memória principal para obter uma instrução completa. Considere agora um processador com códigos de instrução com tamanho variável. Neste caso, o processador não sabe, *a priori*, quantos *bits* deve buscar para obter uma instrução completa. Após realizar um acesso, torna-se necessário que o processador interprete parcialmente o código da instrução para determinar se deve realizar um outro acesso à memória para completar a busca da instrução. A decodificação parcial e o acesso adicional podem comprometer o desempenho ao aumentar o tempo de execução da instrução.

A segunda vantagem de instruções regulares é a simplicidade na decodificação das instruções. Em instruções regulares, um certo campo de *bits* sempre ocupa a mesma posição. Isto permite, por exemplo, que os operandos da instrução sejam acessados ao mesmo tempo em que o código de operação é interpretado, já que o processador conhece antecipadamente onde encontrar as informações sobre os operandos. Em instruções irregulares, os campos que indicam os operandos podem aparecer em qualquer posição dentro do código da instrução. Assim, é necessário antes interpretar o código de operação, para determinar as posições dos campos de operando dentro daquele código de instrução em particular. Agora, a

decodificação e o acesso aos operandos são realizados seqüencialmente, o que contribui para aumentar o tempo de execução das instruções.

4.6 Resumo Inicial sobre o Conjunto de Instruções do Processador

Este capítulo, até este ponto, discutiu as principais características de um conjunto de instruções. Praticamente todas as arquiteturas oferecem instruções aritméticas e lógicas, instruções de movimentação de dados e instruções de transferência de controle. A presença de outros tipos de instruções depende da orientação da arquitetura (*Complex Instruction Set Computer - CISC* ou *Reduced Instruction Set Computer -RISC*) e se está voltada para aplicações gerais ou específicas.

As arquiteturas também diferem quanto ao número de operandos e à localização destes operandos nas instruções aritméticas/lógicas. Quanto a estes dois aspectos, as arquiteturas podem ser classificadas em memória-memória, memória-registrador ou registrador-registrador.

O modo de endereçamento é a maneira como uma instrução especifica a localização dos seus operandos. Os modos de endereçamento registrador e imediato são oferecidos por praticamente todas as arquiteturas. Alguns modos de endereçamento, como o relativo à base e o indexado, são úteis no acesso a alguns tipos de variáveis normalmente encontrados em linguagens de alto nível.

O formato de uma instrução diz respeito a características dos códigos de instrução, tais como o seu tamanho e localização dos campos de *bits*. Em arquiteturas com formato de instruções regular, os códigos de instrução possuem tamanho fixo e os campos de *bits* ocupam a mesma posição dentro do código. Instruções regulares facilitam as operações de busca e decodificação das instruções.

4.7 Instruções Básicas: A Linguagem da Máquina

Para exercitar os conceitos apresentados nas seções anteriores, discutimos nesta seção o conjunto de instruções do processador MIPS. O objetivo é ensinar um conjunto de instruções tão simples quanto possível, mostrando tanto a interface do conjunto de instruções com o *hardware* quanto à relação entre as linguagens de alto nível e o conjunto de instruções.

No projeto de qualquer conjunto de instruções é importante lembrar o que os projetistas têm como objetivos comuns. Os dois objetivos principais são:

- encontrar um conjunto de instruções que facilite tanto a construção do *hardware* quanto do compilador e
- maximizar o desempenho e minimizar o custo.

Para atingir estes objetivos existem quatro princípios básicos do projeto do *hardware*:

1. A simplicidade é favorecida pela regularidade – instruções regulares;
2. Quanto menor mais rápido – menor quantidade de componentes possível;
3. Um bom projeto demanda compromisso – mesmo tamanho x diferentes tipos de instrução e
4. Torne o caso comum mais rápido – operações com inteiros e transferência entre registradores.

4.7.1 Operandos do *Hardware* e da Máquina

Os registradores da arquitetura do processador MIPS são de 32 *bits*. São nesses registradores que ficam armazenados os operandos que vão ser manipulados (palavra de 32 *bits*).

As operações aritméticas do processador MIPS utilizam três operandos (dois fontes e um destino). Por convenção a linguagem de montagem do MIPS usa um cifrão **$** seguido de dois caracteres para referenciar um registrador. Por exemplo, **$s0**, **$s1**, para os registradores que correspondem às **variáveis dos programas** escritos em linguagem de alto nível, e **$t0**, **$t1**, para os **registradores temporários**, necessários à tradução do programa para as instruções do MIPS.

Compilação de um Comando de Atribuição em C usando Registradores

O compilador é responsável pela associação entre as variáveis de um programa e os registradores do *hardware*.

Suponha o seguinte comando em linguagem de alto nível $f = (g + h) – (i + j)$. Os mnemônicos para a soma e a subtração do processador MIPS são iguais a *add* e *sub*, respectivamente. Como seria o código em linguagem de montagem do MIPS resultado deste comando?

Supondo que as variáveis *f*, *g*, *h*, *i* e *j* são atribuídas aos registradores *$s0*, *$s1*, *$s2*, *$s3* e *$s4* temos:

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

Na verdade as linguagens de alto nível dão suporte a estruturas de dados mais complexas, como vetores (*arrays*). Esta estrutura, em geral, contém mais elementos que o número de registradores disponíveis na arquitetura. Assim, estas estruturas ficam armazenadas em memória.

O processador MIPS só manipula dados armazenados em registradores. Dessa forma, é necessário ter instruções de acesso à memória. Estas instruções no processador MIPS são denominadas *lw* (*load word*) e *sw* (*store word*). A instrução *lw* transfere dados da memória para o processador. A instrução *sw* transfere dados do processador para a memória.

Compilação de um Comando de Atribuição em C com Operandos em Memória

Suponha que *A* seja um *arrays* com tamanho igual a 100 palavras e que o compilador tenha associado as variáveis *g* e *h* aos registradores *$s1* e *$s2*, respectivamente. Suponha, também, que o endereço inicial do *array* esteja armazenado em *$s3*. Como traduzir o comando de atribuição *g = h + A[8]* para a linguagem de montagem do processador MIPS? Qual o modo de endereçamento utilizado?

lw $t0, 32 ($s3)
add $s1, $s2, $t0

No processador MIPS as palavras precisam sempre começar em **endereços que sejam múltiplos de 4**. Esta é uma **restrição de alinhamento** porque o tamanho da palavra é igual a 4 *bytes*.

As máquinas que endereçam bytes podem ser divididas em duas categorias: *big endian* e *little endian*. Estas categorias se diferenciam em relação à posição do *byte* que representa o endereço da palavra. Na categoria *big endian* o *byte* mais à esquerda é que representa o endereço da palavra, já na categoria *little endian* é o *byte* mais à direita. O processador MIPS pertence à categoria *big endian*.

4.7.2 Representação de Instruções

As instruções são representadas no computador como números binários. Cada parte de uma instrução, é denominada campo e, pode ser considerada como um número separado. Esses números colocados um ao lado do outro formam a instrução.

Como os registradores comumente são utilizados nas instruções é necessário haver uma convenção para mapear o nome dos registradores em números. Na linguagem de montagem do MIPS os nomes ***$s0* a *$s7*** são mapeados nos registradores de **16 a 23** e os nomes ***$t0* a *$t7*** nos registradores de **8 a 15**.

Cada um dos campos de uma instrução possui sua representação binária. No MIPS todas as instruções têm exatamente 32 bits (princípio da simplicidade).

Campos das Instruções de Máquina do Processador MIPS

Para simplificar a compreensão sobre o formato das instruções os campos recebem nomes.

- Instruções Lógicas e Aritméticas

A Figura 4.8 mostra o formato de uma instrução lógica e aritmética do processador MIPS.

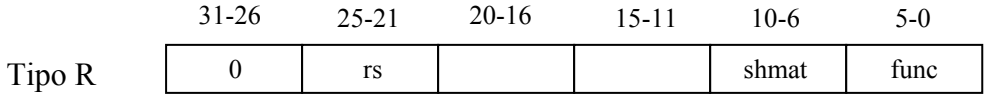


Figura 4.8. *Formato de instrução lógica e aritmética.*

Estas instruções também são denominadas como instruções **tipo R**, onde os campos têm os seguintes significados: **op** – operação básica a ser realizada pela ALU, **opcode**; **rs** – primeiro operando fonte; **rt** – segundo operando fonte; **rd** – operando destino armazena resultado produzido pela ALU; **shmat** – quantidade de bits a serem deslocados (usado em instruções de deslocamento) e **func** – seleciona função da operação apontada no campo op.

As instruções do tipo R possuem opcode igual a 0.

- Instruções de Acesso à Memória

A Figura 4.9 mostra o formato de uma instrução de acesso à memória do processador MIPS.

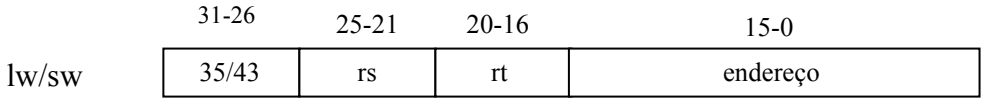


Figura 4.9. *Formato de instrução de acesso à memória.*

O formato de uma instrução de acesso à memória, denominada do **tipo I**, é mostrado na Figura 4.9 e possui os campos op, rs, rt e endereço. O campo de endereço de 16 bits é maior para abranger uma extensa faixa de endereços na memória.

Numa instrução de *lw* pode ser acessado um operando dentro de uma faixa entre o registrador base (dado pelo conteúdo de rs) acrescido ou decrescido de 2^{15} (ou 32.768 palavras). Nesta instrução o campo rt possui outro significado. Ele indica o registrador que armazena o valor fornecido pela memória. O opcode da instrução *lw* é igual a 35.

Numa instrução de *sw* o formato da instrução é idêntico ao da instrução de *lw*. O único campo que possui um significado diferente é o campo rt que indica o registrador que possui o valor que vai ser transferido para a memória. O opcode da instrução de *sw* é igual a 43.

Exemplo:

Suponha o seguinte comando em linguagem de alto nível, *A[300] = h + A[300]*. Uma possível representação em linguagem de montagem do processador MIPS é dada por:

lw $t0, 1200 ($t1)
add $t0, $s2, $t0
sw $t0, 1200 ($t1)

Como podemos saber o código binário correspondente a este comando?

Para facilitar o entendimento, primeiro podemos especificar cada uma dessas instruções com número decimais, de acordo com os seus formatos que são:

	op	rs	rt	endereço		
lw $t0, 1200 ($t1)	*35*	*9*	*8*	*1200*		
	op	rs	rt	rd	shmat	func
add $t0, $s2, $t0	*0*	*18*	*8*	*8*	*0*	*32*
	op	rs	rt	endereço		
sw $t0, 1200 ($t1)	*43*	*9*	*8*	*1200*		

Em seguida, podemos traduzir cada um dos campos das instruções, definidos em números decimais, para os seus respectivos correspondentes em números binários. Esta correspondência está mostrada na Figura 4.10.

100011	01001	01000	0000 0100 1011 0000
000000	10010	01000	01001 00000 100000
101011	01001	01000	0000 0100 1011 0000

Figura 4.10. *Formato das instruções do exemplo em números binários.*

- Instruções de Desvio

Durante a execução de um programa, dependendo dos dados de entrada e dos valores gerados, diferentes instruções podem vir a ser executadas. Em linguagem de alto nível o comando *if* representa fluxos de execução diferentes.

O processador MIPS possui duas instruções de desvio, denominadas *beq* e *bne*, que são similares a um *if* e um *go to* combinados. Elas possuem a seguinte sintaxe.

beq reg1, reg2, L1
bne re1, reg2, L1

O mnemônico *beq* significa *branch equal*. Essa instrução quando executada força um desvio na execução para o *label* L1 (endereço de memória L1), se o valor de registrador 1 for igual ao valor do registrador 2. Já o mnemônico *bne* significa *branch not equal* e a instrução desvia o fluxo de execução do programa para o endereço L1 se o valor dos registradores 1 e 2 forem diferentes. Estas instruções são denominadas instruções de **desvio condicional**. A Figura 4.11 mostra o formato das instruções de desvio condicional.

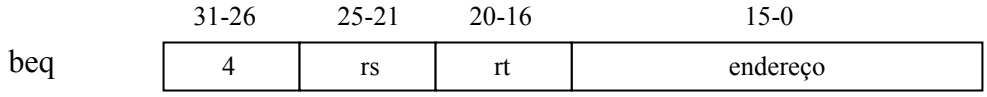


Figura 4.11. *Formato de instruções de desvio condicional.*

O campo op define a operação *beq* ou *bne*. Os campos rs e rt definem os registradores que contém os valores que devem ser comparados. O campo de endereço é manipulado por uma lógica para produzir um valor de 32 *bits*, que combinado com o valor do PC + 4 produz o endereço alvo de desvio, caso o desvio seja realizado de fato. Os opcodes das instruções de *beq* e *bne* são iguais a 4 e 5, respectivamente.

Uma outra instrução de desvio muito utilizada é a instrução de **desvio incondicional** denominada *jump* (*j*). Nesta instrução o desvio é sempre realizado. O formato desta instrução está mostrado na Figura 4.12. O seu opcode é igual a 2.

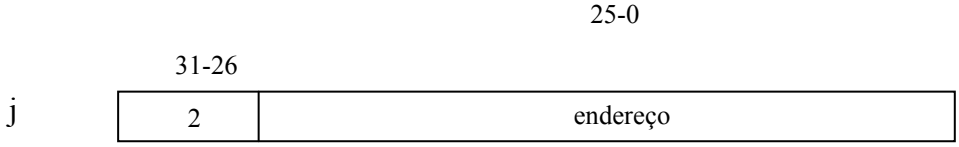


Figura 4.12. *Formato de instrução de desvio incondicional*

Compilação de um Comando *if* em uma Instrução de Desvio Condicional

Considere o seguinte trecho de código em linguagem de alto nível.

```
        if (i == j) go to L1;
        f = g + h;
L1:     f = f – i;
```

Supondo que as cinco variáveis *f*, *g*, *h*, *i* e *j* correspondem aos cinco registradores de *$s0* a *$s4*, qual o código do processador MIPS gerado pelo compilador?

```
        beq $s3, $s4, L1
        add $s0, $s1, $s2
L1:     sub $s0, $s0, $s3
```

onde L1 corresponde ao endereço da instrução *sub* armazenada na memória.

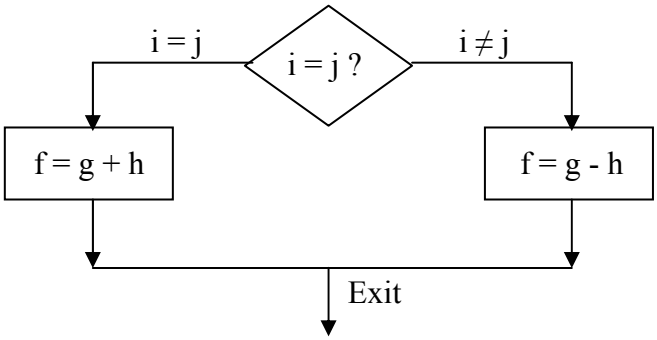
Compilação de um Comando *if-then-else* em Instruções de Desvio Condicional

Considere o seguinte trecho de código em linguagem de alto nível.

```
if (i == j)
    f = g + h;
else
    f = g - h;
```

Supondo que as cinco variáveis *f*, *g*, *h*, *i* e *j* correspondem aos cinco registradores de *$s0* a *$s4*, qual o código do processador MIPS gerado pelo compilador?

Este comando pode ser representado através do seguinte diagrama de fluxo.



A primeira expressão compara dois valores em busca da igualdade. Em geral, realizamos o teste com a condição oposta para que possamos desviar sobre o código que executa a parte subseqüente, correspondente ao *then*. Para controlar os desvios são necessários dois *labels*, um para o *then* (*Else*) e outro para ir para o fim do comando *if* (*Exit*).

```
        bne $s3, $s4, Else
        add $s0, $s1, $s2
        j  Exit
Else:   sub $s0, $s1, $s2
Exit:
```

Loops

Os comandos de desvio são também importantes para o controle de iterações de uma determinada computação, como o caso dos *loops*. As mesmas instruções de linguagem de montagem são usadas nas construções dos *loops*.

Compilação de um *Loop* Contendo um *Array* com Índice Variável

Suponha o seguinte *loop* escrito em linguagem de alto nível.

```
Loop:  g = g + A[i];
       i = i + j;
       if ( i != h) go to Loop;
```

Suponha também que *A* seja um *array* de 100 elementos e que o compilador associe as variáveis *g*, *h*, *i* e *j* aos registradores de *$s1* a *$s4*, respectivamente. Além disso, que o registrador *$s5* contém o endereço base do *array A*. Qual o código em linguagem de montagem do processador MIPS que corresponde a este *loop*?

```
Loop:  add $t1, $s3, $s3        // t1 recebe 2 * i
       add $t1, $t1, $t1        // t1 recebe 4 * i
       add $t1, $t1, $s5        // t1 recebe endereço de A[i]
       lw   $t0, 0 ( $t1)       // t0 recebe conteúdo de A[i]
       add $s1, $s1, $t0
       add $s3, $s3, $s4
       bne $s3, $s2, Loop
```

As seqüências de instruções terminadas em um desvio são tão fundamentais para a compilação que recebem definição própria. Elas são chamadas de **bloco básico**. Uma das principais tarefas do compilador é identificar nos programas os blocos básicos.

While

Considere o seguinte *loop* tradicional escrito em linguagem de alto nível.

```
while ( save[i] == k)
        i = i + j;
```

Suponha que *i*, *j* e *k* correspondam as variáveis *$s3*, *$s4* e *$s5*, e que o endereço inicial do *array save* esteja armazenado em *$s6*. Qual o código em linguagem de montagem correspondente a este trecho de código?

```
Loop:  add $t1, $s3, $s3
       add $t1, $t1, $t1
       add $t1, $t1, $s6
       lw   $t0, 0 ( $t1)
       bne $t0, $s5, Exit
       add $s3, $s3, $s4
       j    Loop
Exit:
```

Para testar se uma variável é menor que outra, a linguagem de montagem do processador MIPS possui uma instrução denominada *set on less than*. Esta instrução compara o conteúdo de dois registradores e atribui o valor 1 a um terceiro registrador se o conteúdo do primeiro registrador for menor do que o conteúdo do segundo registrador. Caso contrário, é atribuído o valor 0 ao terceiro registrador. O mnemônico desta instrução é *slt*. Ela é do tipo R,

seu opcode é igual a 0 e o campo *func* é igual a 42. O formato desta instrução pode ser visto na Figura 4.13

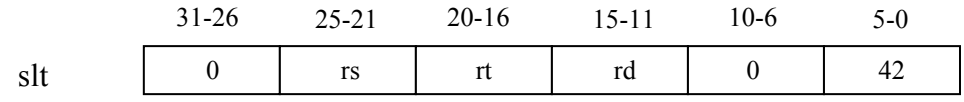


Figura 4.13. *Formato da instrução set on less than.*

O processador MIPS possui um registrador com valor fixo em zero. Ele é o registrador **$zero**, mapeado no registrador 0 e tem acesso somente de leitura. Este registrador é usado para criar todas as demais condições como maior ou igual e menor ou igual.

Exemplo:

Como seria o código do processador MIPS para testar se uma variável *a* (*$s0*) é menor que outra variável *b* (*$s1*) e desviar para *Less* se a condição for verdadeira.

```
slt $t0, $s0, $s1
bne $t0, $zero,Less
```

Switch/Case

O comando *switch/case* permite ao programador de alto nível selecionar uma entre muitas alternativas. Estas alternativas são selecionadas dependendo do valor de uma variável.

As alternativas podem ser codificadas como uma tabela de endereços em seqüências de instruções. Esta tabela é denominada **tabela de endereços de desvio**. O programa precisa indexar esta tabela e desviar para a seqüência desejada. A tabela de desvios nada mais é do que um *array* de palavras contendo endereços que correspondem aos *labels* dentro do código.

O processador MIPS possui uma instrução de desvio que considera o conteúdo de um registrador. Esta instrução desvia incondicionalmente para o endereço armazenado no registrador especificado pela instrução. O formato desta instrução pode ser visto na Figura 4.14.

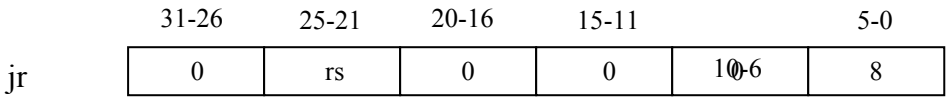


Figura 4.14. *Formato da instrução de desvio incondicional jump register.*

O mnemônico dessa instrução de desvio incondicional é jr. Ela é do tipo R, seu opcode é igual a 0 e o campo *func* é igual a 8.

Compilação de um Comando *Switch* a partir de uma Tabela de Endereços de Desvio

O trecho de código, a seguir, seleciona uma entre quatro alternativas dependendo do valor assumido pela variável *k*, cujo domínio é 0, 1, 2 e 3.

```
switch (k) {
        case 0: f = i + j; break;
        case 1: f = g + h; break;
        case 2: f = g - h; break;
        case 3: f = i - j;
}
```

Suponha que as variáveis de *f* a *k* correspondem às seis variáveis de $s0 a $s5, e que o registrador *$t2* contenha o valor 4 e o registrador *$t4* o valor inicial dos *labels L0*, *L1*, *L2* e *L3* armazenados na memória. Qual o código correspondente em linguagem de montagem?

```
        slt   $t3, $s5, $zero
        bne $t3, $zero, Exit
        slt   $t3, $s5, $t2
        beq $t3, $zero, Exit
        add $t1, $s5, $s5
        add $t1, $t1, $t1
        add $t1, $t1, $t4
        lw   $t0, 0 ($t1)
        jr    $t0
L0:     add $s0, $s3, $s4
        j     Exit
L1:     add $s0, $s1, $s2
        j     Exit
L2:     sub $s0, $s1, $s2
        j     Exit
L3:     sub $s0, $s3, $s4
Exit:
```

4.7.3 Suporte a Chamada de Procedimentos

A utilização de um procedimento, ou sub-rotina, tem a finalidade de estruturar os programas. Dessa forma, os programas podem ser mais facilmente entendidos e permitem a reutilização do código dos procedimentos.

Durante a execução de um procedimento, o programa e o procedimento precisam executar seis passos:

1. armazenar os parâmetros em um local onde eles possam ser acessados pelo procedimento;
2. transferir o controle para o procedimento;
3. garantir os recursos de memória necessários à execução do procedimento;
4. realizar a tarefa desejada;
5. colocar o resultado num lugar acessível ao programa que chamou o procedimento e
6. retornar o programa para o ponto de origem.

A linguagem de montagem do processador MIPS utiliza os seguintes registradores na implementação da chamada de procedimentos:

1. ***$a0 - $a3***: quatro registradores para passagem de parâmetros;
2. ***$v0 - $v1***: dois registradores para retorno de valores para o programa
3. ***$ra***: um registrador que contém o endereço de retorno ao ponto de origem.

Além desses registradores, a linguagem de montagem do MIPS possui uma instrução usada unicamente desviar para um endereço e simultaneamente salvar o endereço da próxima instrução no registrador *$ra*. Esta instrução é chamada de ***jump and link***, cujo mnemônico é dado por ***jal***. O seu opcode é igual a 3 e o seu formato é idêntico ao da instrução de desvio incondicional *jump*.

O valor armazenado no registrador *$ra* é chamado de **endereço de retorno**. A instrução *jal* armazena o valor PC + 4 no registrador *$ra* para estabelecer o *link* de retorno, e a instrução que implementa o retorno é a instrução *jump register*.

Às vezes é necessário passar mais do que quatro parâmetros para o procedimento ou mais de dois valores de retorno para o programa. Nestes casos é necessário usar a memória para passar estas informações. A **pilha** é a estrutura de dados utilizada para esta finalidade. Ela implementa uma **estrutura LIFO** (*Last In First Out*) e possui um ponteiro para o último dado inserido.

O processador MIPS possui um registrador para auxiliar nas operações realizadas sobre a pilha. Este registrador é o ***stack pointer*** (***$sp***). Ele armazena o endereço do topo da pilha. Quando um dado é inserido ou retirado do topo da pilha dizemos que é uma operação de ***push*** ou de ***pop***, respectivamente. Por convenção, as pilhas crescem dos endereços mais altos para os endereços mais baixos. Assim, quando colocamos um dado na pilha o valor corrente do *stack pointer* diminui. No caso inverso ao retirarmos dados da pilha o valor do *stack pointer* aumenta.

A pilha também é utilizada para armazenar variáveis que são locais ao procedimento, se as variáveis forem em maior número do que a quantidade de registradores disponíveis. O segmento da pilha que contém os registradores com os conteúdos salvos do procedimento e suas variáveis locais é chamado de **registro de ativação**. A Figura 4.15 mostra o estado da pilha antes, durante e depois da chamada ao procedimento.

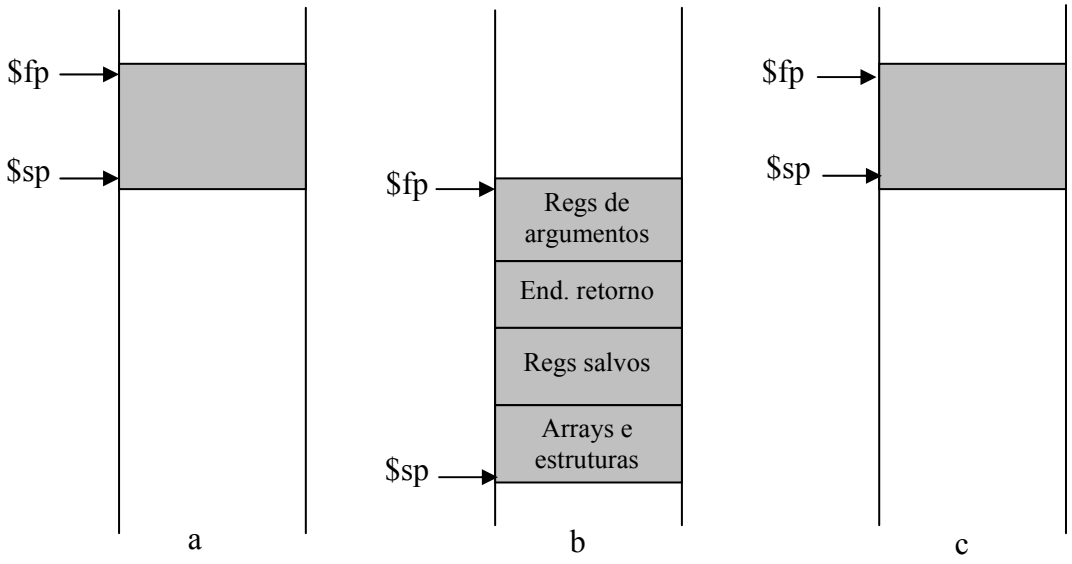


Figura 4.15. *Alocação de espaço na pilha (a) antes (b) durantes e (c) depois da chamada.*

Alguns softwares do processador MIPS usam um registrador chamado ***frame pointer*** (***$fp***) para apontar para o registro de ativação. Eles atuam como um registrador base para as variáveis locais a um procedimento referenciarem a memória.

Alguns registradores devem ser preservados nas chamadas aos procedimentos. A Tabela 4.1 resume as convenções empregadas pela linguagem de montagem do processador MIPS para utilizar os registradores.

Nome	Número do Registrador	Utilização	Preservado?
$zero	0	Valor da constante 0	Não se aplica
$at	1	Reservado para o montador	Não se aplica
$v0-$v1	2-3	Armazenar resultados e avaliar expressões	Não
$a0-$a3	4-7	Argumentos	Sim
$t0-$t7	8-15	Temporários	Não
$s0-$s7	16-23	Salvos	Sim
$t8-$t9	24-25	Temporários	Não
$k0-$k1	26-27	Reservado para o sistema operacional	Não se aplica
$gp	28	*Global pointer*	Sim
$sp	28	*Stack pointer*	Sim
$fp	30	*Frame pointer*	Sim
$ra	31	Endereço de retorno de procedimento	Sim

Tabela 4.1. *Convenções para os registradores da linguagem de montagem do processador MIPS.*

4.7.4 Outras Instruções do Processador MIPS

Os projetistas do processador MIPS desenvolveram duas maneiras diferentes de acesso a operandos. Estes operandos podem ser **operandos imediatos** ou **constantes**. A partir deles é possível tornar mais rápido o acesso a constantes pequenas e mais eficientes os desvios. A seguir abordamos as instruções mais comuns que usam esses dois modos de acesso a operandos.

- Instruções Lógicas e Aritméticas com Operando Imediato

As instruções que utilizam operandos imediatos têm acrescentado em seus mnemônicos a letra *i*. Estas instruções são do tipo I e o campo do valor imediato tem extensão igual a 16 *bits*. Por exemplo, a instrução de soma com um operando imediato, onde desejamos somar a constante 4 ao conteúdo do registrador *$sp* pode ser simplesmente escrita da seguinte forma:

addi $sp, $sp, 4

O mnemônico da instrução de soma com operando imediato é *addi*. Em particular, o opcode da instrução *addi* é igual a 8.

- Instruções de Comparação com Operando Imediato

Os operandos imediatos são muito úteis em comparações. O registrador *$zero* contém a constante 0 proporcionando um meio fácil de realizar comparações com 0. Para proporcionar a mesma funcionalidade de comparação com outros valores, existe uma versão imediata da instrução *set on less than*. Por exemplo, para saber se o conteúdo do registrador *$s2* é menor do que a constante 10 podemos simplesmente escrever:

slti $t0, $s2, 10

O mnemônico da instrução *set on less than* com operando imediato é *slti*.

- Instruções de Carga com Operando Imediato

Ás vezes é necessário obter operandos imediatos que ultrapassem a capacidade de 16 *bits* imposta pelo campo de valor imediato das instruções citadas. Para resolver este problema o conjunto de instruções do processador MIPS possui a instrução *load upper immediate*. O seu mnemônico é igual a *lui*. Esta instrução carrega os 16 *bits* mais significativos de um registrador com o valor imediato fornecido na instrução. Uma instrução subseqüente pode especificar os 16 *bits* menos significativos do registrador, formando assim um valor de 32

bits. A Figura 4.16 mostra a operação da instrução de carga do registrador, cujo exemplo é dado a seguir.

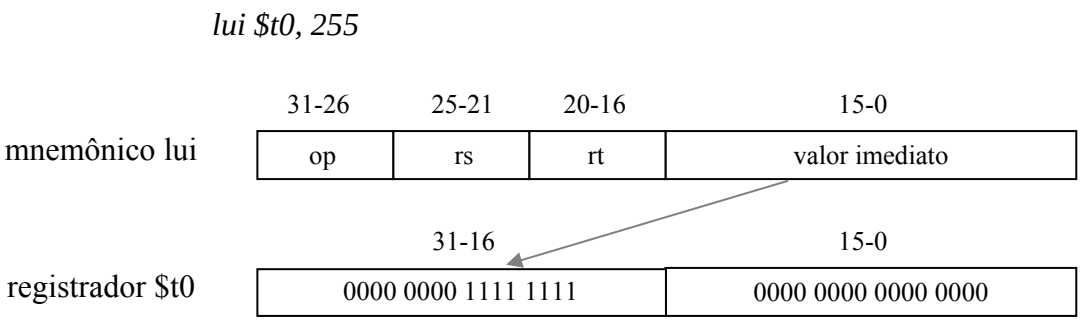


Figura 4.16. *Efeito de instruções de load upper immediate.*

Todas as instruções citadas que possuem operando imediato são do tipo I e o campo do valor imediato tem extensão igual a 16 *bits.* A Figura 4.17 mostra o formato dessas instruções.

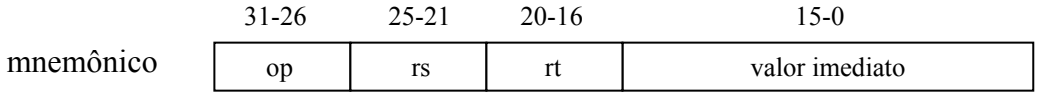


Figura 4.17. *Formato de instruções com operando imediato.*

O conjunto completo das instruções do processador MIPS pode ser encontrado no apêndice A do livro texto.

Exercícios

1) Quais são os tipos de instruções que devem existir em qualquer conjunto de instruções de um processador?

2) Quais são os tipos mais comuns de operandos que são manipulados pelos processadores?

3) O número e a localização dos operandos dão origem a diferentes classes de arquiteturas. Quais são estas classes?

4) Onde podemos encontrar os operandos? Resuma cada um dos modos de endereçamento básicos.

5) O formato de uma instrução pode ser regular ou irregular. Quais as vantagens e desvantagens de cada um destes formatos?

6) Descreva a função de cada um dos níveis do contexto de *software* abordado na primeira seção deste capítulo.

7) Adicione comentários ao código do processador MIPS, a seguir, e expresse em uma frase o que este trecho de código faz. Suponha que os registradores *$a0* e *$v0* são usados para a entrada e a saída, respectivamente. Além disso, suponha que inicialmente o registrador de entrada contém um valor *n* que é um inteiro positivo.

```
begin: addi $t0, $zero,0
       addi $t1, $zero, 1
loop:  slt $t2, $a0, $t1
       bne $t2, $zero ,finish
       add $t0, $t0, $t1
       addi $t1, $t1, 2
       j    loop
finish: add $v0, $t0, $zero
```

8) Mostre a única instrução ou o menor número de instruções do processador MIPS para gerar o seguinte comando em linguagem de alto nível. Suponha que as variáveis *a* e *b* estejam associadas aos registradores *$t0* e *$t1*, respectivamente.

a = b + 100;

9) Mostre a única instrução ou o menor número de instruções do processador MIPS para gerar o seguinte comando em linguagem de alto nível. Suponha que a variável *c* esteja associada ao registrador *$t0* e que o endereço base do *array x* começa em 4096.

x[10] = x[11] + c;

10) Considere o seguinte trecho de código escrito em linguagem de alto nível:

```
for (i = 0; i <= 100; i ++) {
    a[i] = b[i] + c;
}
```

Suponha que *a* e *b* são *arrays* de palavras e que o endereço base de *a* está em *$a0* e o endereço de *b* está em *$a1*. O registrador *$t0* está associado a variável *i* e *$s0* a variável *c*. Escreva o código MIPS correspondente a este fragmento de código. Quantas instruções são executadas durante o processamento deste trecho de código? Quantas referências a palavras de dados na memória serão feitas durante a execução?

11) O fragmento de código mostrado, a seguir, processa um *array* e calcula dois valores armazenados nos registradores *$v0* e *$v*1. Suponha que o *array* possua 500 palavras, indexadas de 0 a 499, e que seu endereço base e tamanho estejam armazenado nos registradores *$a0* e *$a1*. Descreva o que faz este trecho de código. Mais especificamente o que é retornado nos registradores *$v0* e *$v*1?

```
        add $a1, $a1, $a1
        add $a1, $a1, $a1
        add $v0, $zero, $zero
        add $t0, $zero, $zero
outer:  add $t4, $a0, $t0
        lw   $t4, 0 ($t4)
        add $t5, $zero, $zero
        add $t1, $zero, $zero
inner:  add $t3, $a0, $t1
        lw   $t3, 0 ($t3)
        bne $t3, $t4, skip
        addi $t5, $t5, 1
skip:   addi $t1, $t1, 4
        bne $t1, $a1, inner
        slt $t2, $t5, $v0
        bne $t2, $zero, next
        add $v0, $t5, $zero
        add $v1, $t4, $zero
next:   addi $t0, $t0, 4
        bne $t0, $a1, outer
```

12) Suponha que o código do exercício 10 seja executado em uma máquina com *clock* igual a 500Mhz. Além disso, a quantidade de ciclos gastos em cada instrução, para esta máquina, é dada pela tabela a seguir. No pior caso quantos segundos serão necessários para executar todo este código?

Instrução	Ciclos
add, addi, slt	1
lw, bne	2