Capítulo 4

Carregadores e ligadores

O resultado do processo de compilação é um arquivo contendo um programa em *assembly* equivalente ao programa originalmente descrito em linguagem de alto nível. Um programa em linguagem *assembly*, ou linguagem simbólica, contém seqüências de instruções mnemônicas que representam as operações que devem ser realizadas pelo processador. Essas instruções são definidas pelos projetistas do processador; o conjunto de todas as instruções definidas para um processador contitui seu **jogo de instruções**.

O **montador** *(assembler)* é o programa do sistema responsável por traduzir o código *assembly* em linguagem de máquina, traduzindo cada instrução do programa para a seqüência de bits que codifica a instrução de máquina. Como cada processador tem sua própria linguagem, montadores são específicos para processadores. Montadores são objetos de estudo da Seção 4.1.

Neste capítulo, serão ainda descritas as atividades do sistema necessárias para que o programa montado possa efetivamente ser executado — a **ligação**, que resolve as referências que tenham sido feitas a dados e rotinas em outros programas, e o **carregamento**, que transfere o programa montado para a memória principal e dá início à sua execução.

4.1 Montadores

O processo de montagem recebe como entrada um arquivo texto com o código fonte do programa em *assembly* e gera como saída um arquivo binário, o **módulo objeto**, contendo o código de máquina e outras informações relevantes para a execução do código gerado.

Em geral, montadores oferecem facilidades além da simples tradução de código *assembly* para código de máquina. Além das instruções do processador, um programa fonte para o montador pode conter diretivas ou **pseudo-instruções** definidas para o montador (e não para o processador), assim como macro-instruções, uma seqüência de instruções que será inserida no código ao ser referenciada pelo nome. Um montador que suporte a definição e utilização de macro-instruções é usualmente denominado um **macro-montador** (*macro-assembler*). Um **montador multiplataforma** *(cross-assembler)* é um montador que permite gerar código para um processador-alvo diferente daquele no qual o montador está sendo executado.

Na seqüência apresenta-se brevemente as atividades relacionadas ao processo de montagem, partindo da descrição do formato de entrada esperado até a geração do módulo-objeto de saída.

4.1.1 Programas *assembly*

O montador recebe como entrada um arquivo texto cujas linhas são instruções *assembly*. Embora o formato específico de um arquivo-fonte em *assembly* possa sofrer ligeiras variações de acordo com o sistema, a descrição a seguir cobre a maior parte dos aspectos relevantes para a operação do montador.

O seguinte trecho de código apresenta um típico programa *assembly*:

```
POS      DS.W      1
; Busca 0 na sequencia de inteiros
SRCH0    MOVEA.L   #DATUM,A0   ; (DATUM) definido alhures
         MOVE.L    #DATUM,D0   ; guarda inicio
         CLR.W     D1
LOOP     CMP.W     (A0)+,D1
         BNE       LOOP
         SUB.L     A0,D0
         MOVE.W    D0,POS
         RTS
         END
```

Cada linha desse programa pode conter instruções ou comentários; uma linha é de comentário quando contém no início da linha o caráter `;` (ponto-e-vírgula).

As linhas de instrução contêm até quatro campos. A primeira coluna pode apresentar um **rótulo** opcional. A função básica do rótulo é criar uma identificação para poder referenciar simbolicamente a linha de código rotulada. O montador pode reconhecer rótulos pela presença de caracteres distintos de `;` a partir da primeira posição da linha.

A segunda coluna contém o **campo de operação**, que especifica a instrução que será montada. A operação pode ser tanto uma instrução de máquina, a exemplo de `MOVE` e `RTS`, como uma pseudo-instrução, como `DS`. Os sufixos às instruções presentes no exemplo indicam o tamanho do operando — no caso dos processadores da família 68K, `.B` *(byte)*, `.W` *(word)* ou `.L` *(long word)* respectivamente para um, dois ou quatro bytes. Se for omitido, o tamanho *word* é usado como padrão.

Dependendo da instrução presente na segundo coluna, o montador sabe se deve esperar zero, um ou dois operandos na terceira coluna, que corresponde ao **campo de operandos**. Operandos podem fazer referências a registradores do processador (no caso do 68K, registradores de dados `D0` a `D7`, de endereços `A0` a `A7`, contador de programas `PC` e, para algumas instruções, de códigos de condição `CCR`), a símbolos definidos pelos programas através de rótulos e a valores constantes.

A especificação do valor de uma constante que representa um operando imediato, indicado pelo prefixo #, pode se dar sob diversas formas de representação. Por exemplo, nas instruções

```
MOVE.B    #48,D0
MOVE.B    #$30,D0
MOVE.B    #@60,D0
MOVE.B    #%110000,D0
MOVE.B    #'0',D0
```

o operando imediato é sempre o mesmo valor, representado respectivamente como um número decimal (sem prefixo adicional), hexadecimal (prefixo `$`), octal (prefixo `@`), binário (prefixo `%`) e ASCII (entre aspas simples). Qualquer que fosse a forma selecionada, o código de máquina gerado para essa instrução seria o mesmo:

```
00010000 00111100
00000000 00110000
```

A especificação do código de máquina para as principais instruções da família 68K usadas neste texto são apresentadas no Apêndice B.

Seqüências de caracteres *(strings)* são definidas também entre aspas simples. Por exemplo, `'ABC'` define uma seqüência de três bytes com valores $41, $42 e $43.

A quarta e última coluna, também opcional, corresponde ao **campo de comentários**. No exemplo, cada comentário é iniciado pelo caráter `;`, após o qual todo o restante da linha pode ser ignorado pelo montador.

Através do uso de pseudo-instruções e dos rótulos, é possível fazer referências a posições de memória e a variáveis através de identificadores simbólicos. Isso permite que o programador possa usar esses identificadores simbólicos como operandos de suas instruções sem ter que necessariamente saber a qual posição de memória a variável ou instrução está alocada. A regra para a composição de tais identificadores pode apresentar diferenças entre montadores distintos. Em geral, identificadores podem incluir letras minúsculas ou maiúsculas, dígitos e o caráter _, mas não podem ser iniciados por um dígito.

Estrutura de programa *assembly*

Um programa *assembly* é tipicamente composto por pelo menos dois segmentos, um **segmento de dados** que define o espaço associado ao armazenamento das variáveis e constantes usadas pelo programa; e um **segmento de instruções**, onde o código do programa é armazenado. Além dessas duas seções, um programa-fonte *assembly* pode conter uma **seção de definições**, usadas na descrição dos programas e que não produzem nenhum efeito no código gerado.

Por conveniência da leitura do código fonte, a seção de definições é tradicionalmente alocada ao início do código. Assim, quando o código for lido por um ser humano ele terá noção do significado das constantes simbólicas usadas ao longo do programa. Com relação aos segmentos de dados e de instruções, não há um posicionamento fixo. Na prática, um programa pode ter vários segmentos associados.

Para o montador, o posicionamento dos diferentes trechos de programa no código fonte deve ser irrelevante. Na verdade, é necessário que o montador seja capaz de manipular representações simbólicas antes que elas tenham sido definidas. Considere o seguinte exemplo de um trecho de programa:

```
1       START   ADD.L   D0,D1
2               JMP     NEXT
3       LOOP    ADD.L   #1,D1
4       NEXT    CLR.L   D5
5               JMP     LOOP
```

Na linha 2 desse trecho de programa há uma referência a um símbolo, `NEXT`, cujo valor ainda não havia sido determinado — essa definição só acontecerá na linha 4. Há duas possibilidades de lidar com essas referências futuras.

A primeira possibilidade é deixar uma lacuna reservada no código gerado associada ao operando da instrução da linha 2. Posteriormente, quando houvesse uma definição desse valor — provavelmente quando o fim do arquivo com o código fonte fosse alcançado — essa lacuna seria preenchida. Neste caso, seria possível gerar o código de máquina realizando um único passo (uma única leitura) sobre o arquivo. Entretanto, haveria um maior custo na complexidade de implementação do montador, que deveria manter referências a todas as lacunas que devem ser preenchidas ao final da montagem.

A outra possibilidade, conceitualmente mais simples, é realizar o processo montagem em dois passos. O primeiro passo simplesmente lê o arquivo com o objetivo de criar a Tabela de Símbolos, ou seja, obter os valores associados a todas as constantes simbólicas definidas no programa. No segundo passo, uma nova leitura sobre o arquivo é realizada para gerar o código de máquina; nesse passo, a informação da tabela de símbolos criada no primeiro passo é utilizada.

Pseudo-Instruções

Além das instruções do processador, um programa *assembly* preparado para um montador também pode conter pseudo-instruções que estabelecem a conexão entre referências simbólicas e valores a serem efetivamente referenciados. Cada montador pode oferecer um conjunto de pseudo-instruções diferenciado. As pseudo-instruções descritas a seguir representam um subconjunto significativo de facilidades oferecidas por montadores.

A pseudo-instrução de **substituição simbólica**, `EQU`, associa um valor definido pelo programador a um símbolo. Por exemplo, a linha de instrução

```
SIZE    EQU     100
```

associa o valor decimal 100 ao símbolo SIZE, que pode ser posteriormente referenciado em outras instruções, como em

```
        MOVE    #SIZE,D0
```

Para essa pseudo-instrução, o rótulo deve estar sempre presente e o operando pode ser qualquer expressão que, quando avaliada, defina o valor para o símbolo. Essa expressão pode envolver outros símbolos já definidos.

A pseudo-instrução `EQU` define símbolos que serão usados durante o processo de montagem, mas que não farão parte do módulo-objeto. Para definir constantes para a execução do código, ou seja, que ocuparão algum espaço em memória durante a execução do programa gerado, as pseudo-instruções `DC` e `DS` devem ser usadas.

A **definição de variável inicializada**, isto é, com algum valor constante definido no momento da alocação de espaço para a variável, dá-se através da pseudo-instrução `DC`, como nos exemplos

```
CONTADOR  DC.L    100
ARR1      DC.W    0,1,1,2,3,5,8,13
MENSAGEM  DC.B    'Alo, pessoal!'
```

O rótulo deve estar presente nessa instrução para permitir referenciar a posição de memória de cada variável ao longo do código. O sufixo no código de operação indica o tamanho em bytes para cada variável, seguindo o padrão das instruções da família 68K. Assim, `CONTADOR` fará referência a uma posição de memória cujos quatro bytes seguintes terão inicialmente a representação para o valor 100. `ARR1` é uma referência para a posição de memória que dá início a um bloco contíguo de oito palavras de dois bytes, cada uma delas com o valor especificado na posição correspondente no operando. De forma similar, `MENSAGEM` faz referência ao início de um bloco de treze bytes representados em ASCII no operando.

Outra forma de reservar um espaço de memória para armazenar valores é através da pseudo-instrução de **declaração de variáveis**, `DS`, que reserva a quantidade de espaço indicada mas não inicializa seu conteúdo. Por exemplo,

```
VALUE   DS.W    1
```

associa ao símbolo `VALUE` uma referência para um endereço de memória que tem espaço suficiente para armazenar valores de uma variável de tamanho dois bytes (word).

A pseudo-instrução `ORG` determina a **origem do segmento**. Um segmento é um conjunto de palavras de máquina que deve ocupar um espaço contíguo na memória principal. A posição de memória (endereço) associada ao início do segmento é denominada a sua origem. O módulo-objeto gerado pelo montador contém tipicamente pelo menos dois segmentos, um segmento de código de máquina e um segmento de dados.

O efeito da pseudo-instrução `ORG` depende do tipo de montador que irá interpretá-la. Em alguns casos, pode ser uma definição de um endereço absoluto de memória no qual a origem do segmento deve ser posicionada. Em outros, é apenas a definição de um nome para referências futuras ao segmento quando sua posição de origem for definida. A forma genérica aqui adotada para a instrução será

```
        ORG ident
```

onde `ident` é um identificador que pode ser um valor constante já definido (no caso dos montadores absolutos) ou estar sendo definido como o nome de um segmento (nos demais casos).

Por exemplo, no trecho de programa a seguir


```
SEG1    EQU     $1000
        ORG     SEG1
        MOVE.W  DATA,D0
        MOVE.W  D0,DATA+2
        RTS
```

indica que a primeira instrução `MOVE.W` (na terceira linha) estará alocada à posição $1000 da memória, dando início ao segmento de código do módulo-objeto que será gerado.

Uma outra pseudo-instrução importante é `END`, que indica ao montador o **fim do programa** *assembly*. Seu formato geral é

```
        END     ident
```

onde o identificador no operando está associado a um rótulo do início do programa que está sendo encerrado por essa pseudo-instrução. Assim, esse identificador só deve estar presente uma única vez no código, mesmo que o programa fonte esteja distribuído entre diversos arquivos — em geral, está associado a um "módulo principal". Nos demais módulos, a pseudo-instrução `END` aparece sem argumentos, sempre na última linha.

No caso de um programa cujo código-fonte está distribuído entre diversos segmentos, pode ser preciso fazer referências desde um segmento a variáveis definidas em outros arquivos-fontes. Para possibilitar essa conexão de referências, a pseudo-instrução `GLOB` é usada para indicar que cada um dos símbolos indicados pode ser **referenciável externamente**, ou seja, torna o símbolo visível globalmente. Seu formato genérico é

```
        GLOB    idents
```

onde `idents` é a lista de identificadores (separados por vírgulas, se mais de um estiver presente) definidos nesse segmento com a pseudo-instrução que podem ser referenciados a partir de outros módulos. Os demais símbolos definidos no segmento são considerados de escopo local, ou seja, são invisíveis para os módulos externos.

Alguns montadores definem pseudo-instruções tais como `EXTERN` para indicar que o símbolo que está sendo usado no módulo foi definido externamente, em outro módulo. No entanto, essa pseudo-instrução é desnecessária se for assumido que todos os símbolos referenciados mas não definidos localmente devem estar definidos externamente; esse comportamento é adotado pelo montador GNU, o programa `as`.

Macro-instruções

Uma macro-instrução é um sinônimo para um grupo de instruções que pode ser usado como uma instrução ao longo do código-fonte. O uso de macros facilita a especificação de trechos repetitivos de código, que podem ser invocados pelo programador como um única linha no programa. Por esse motivo, diversos montadores apresentam extensões com funcionalidades para a definição e utilização de macros.

Na sua forma mais simples, uma macro é simplesmente uma abreviatura para um grupo de instruções. A forma geral de definição de uma macro é

```
nome    MACRO   [argumentos]
        corpo
        ENDM
```

A pseudo-instrução `MACRO` marca o início da definição da macro-instrução. Toda macro tem um **nome**, especificado como o rótulo da pseudo-instrução e que será utilizado pelo programador para invocar a macro, e um **corpo**, que será usado pelo macro-montador para substituir o nome usado pelo programador pela seqüência de instruções nele especificados. A pseudo-instrução `ENDM` marca o fim da definição.

Uma macro pode opcionalmente receber argumentos, que serão usados para adaptar a expansão do corpo da macro. Assim, a seqüência de instruções especificadas no corpo da macro podem ser parametrizadas pelos argumentos. Os parâmetros formais na definição de uma macro-instrução são precedidos pelo símbolo &.

A definição de `TOLOWER`, exemplo apresentado abaixo, cria uma macro-instrução com dois parâmetros. O primeiro, `&IN`, é interpretado como a referência a um endereço de memória de um byte cujo conteúdo será copiado para o registrador `D0`, onde o sexto bit será setado. O conteúdo resultante será copiado para a posição indicada pelo segundo argumento, `&OUT`:

```
TOLOWER  MACRO     &IN,&OUT
         MOVE.B    &IN,D0
         ORI.B     32,D0
         MOVE.B    D0,&OUT
         ENDM
```

Uma vez que uma macro esteja definida, seu nome pode ser utilizado como se fosse uma operação válida do montador. A associação entre os argumentos da invocação da macro e os parâmetros formais da definição é feita pela posição da variável na declaração e invocação, assim como ocorre em subrotinas nas linguagens de alto nível.

Considerando a definição acima, o uso da macro `TOLOWER` dar-se-ia como em

```
SIZE     EQU       5
CHARS_I  DC.B      'EA876'
CHARS_O  DS.B      SIZE
PROG001  MOVEA.L   #CHARS_I,A0
         MOVEA.L   #CHARS_O,A1
         MOVE.W    #SIZE,D0
LOOP     TOLOWER   (A0),(A1)
         ADDA.W    #1,A0
         ADDA.W    #1,A1
         DBF       D0,LOOP
         RTS
         END       PROG001
```

Após passar pela etapa de processamento de macros, o código acima seria expandido para o seguinte código *assembly*:

```
SIZE     EQU       5
CHARS_I  DC.B      'EA876'
CHARS_O  DS.B      SIZE
PROG001  MOVEA.L   #CHARS_I,A0
         MOVEA.L   #CHARS_O,A1
         MOVE.W    #SIZE,D0
LOOP     MOVE.B    (A0),D0
         ORI.B     32,D0
         MOVE.B    D0,(A1)
         ADDA.W    #1,A0
         ADDA.W    #1,A1
         DBF       D0,LOOP
         RTS
         END       PROG001
```

Outra facilidade geralmente associada a macro-instruções é a possibilidade de definir expansões condicionais de trechos de código. Para tanto, a pseudo-instrução `AIF` é definida. Assim, é possível definir trechos da definição da macro que poderão não estar incluídos na respectiva expansão.

O formato dessa pseudo instrução é

```
        AIF       cond  .mrot
```

onde `cond` é a condição que deve ser avaliada e `.mrot` é o rótulo na macro onde a expansão deverá continuar se a condição for verdade; caso contrário, a expansão continua na linha seguinte. A condição envolve tipicamente operadores relacionais de comparação de *strings*, aqui denotados `EQ` e `NE` para expressar "igual a" e "diferente de", respctivamente. O rótulo de macro é sempre iniciado por um ponto, uma forma de diferenciá-lo dos rótulos que serão incluídos no código expandido.

Para ilustrar esse conceito, considere novamente a definição da macro `TOLOWER`, que usa o registrador `D0` para realizar a operação desejada. Se um dos argumentos para a macro for esse registrador, uma das instruções `MOVE` não deve ser incluída na sua expansão. Usando `AIF`, uma nova definição para essa macro que considera essa possibilidade é

```
TOLOWER  MACRO     &IN,&OUT
         AIF       (&IN EQ 'D0') .PULA
         MOVE.B    &IN,D0
.PULA    ORI.L     32,D0
         AIF       (&OUT EQ 'D0') .FIM
         MOVE.L    D0,&OUT
.FIM     ENDM
```

No Apêndice C.7 apresenta-se o pré-processador C, uma facilidade para definição de macros associada a uma linguagem de alto nível.

4.1.2 Montagem

O processo de montagem de um código *assembly* pode apresentar pequenas diferenças em função das opções adotadas no projeto do montador, mas em linhas gerais as funcionalidades a seguir são suportadas.

Uma etapa inicial que pode ser suportada é o pré-processamento do código, onde informação não relevante pode ser eliminada. Por exemplo, o pré-processador do montador `as` elimina comentários e converte constantes em formato caráter para as correspondentes constantes em valores numéricos. Na seqüência, o montador realiza o pré-processamento de macros, obtendo um código pronto para a criação do módulo objeto.

Funcionalidades básicas

O montador estará recebendo como entrada um arquivo em formato texto, do qual ele deverá ler cada linha para fazer o processamento que for necessário. Assim, um dos primeiros grupos de funcionalidades que se faz necessário é a manipulação de arquivos, descritas na Seção 2.8.

Uma vez obtida a linha do arquivo, a tarefa de extrair de cada linha o campo de interesse estará representada através dos seguintes procedimentos, todos recebendo como argumento uma referência para a linha a ser processada:

GETLABEL(): extrai o rótulo da linha, se presente; caso contrário, returna o valor nulo.

GETOPERATION(): extrai da linha o mnemônico de operação, que pode ser de uma instrução de máquina ou de uma pseudo-instrução.

GETOPERANDS(): obtém uma lista de operandos, que eventualmente pode ser vazia, a partir do conteúdo do campo de operandos da linha.

Processamento de macro-instruções

De maneira geral, um processador de macro deve realizar quatro tarefas básicas:

1. Reconhecer as definições de macros;
2. Salvar as definições de macros de forma possibilitar a posterior expansão;
3. Reconhecer as invocações a macros; e
4. Expandir as invocações, possivelmente substituindo argumentos e verificando condições.

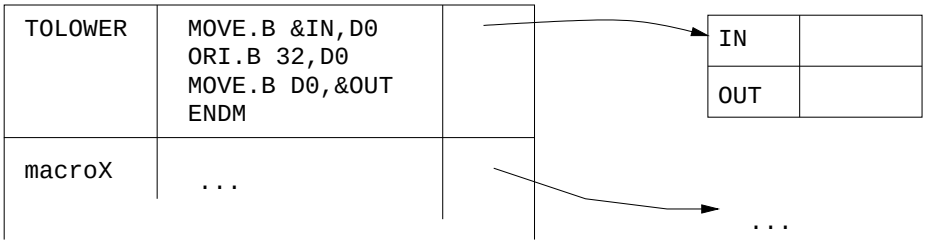
O processador de macros pode ser visto como um programa independente do montador, que é invocado antes do processo de montagem propriamente dito. Sua implementação mais simples pode ser realizada em dois passos.

No primeiro passo, cada linha do arquivo com o programa fonte (já sem comentários) é lida. Caso contenha na coluna do campo de operação a pseudo-instrução `MACRO`, então o que se segue é uma definição de macro, que deve ser armazenada. Uma estrutura de dados, a **Tabela de Definição de Macro**, é usada para guardar essas definições. A chave nessa tabela é o nome da macro, definido no campo de rótulo dessa mesma linha.

Associado a cada nome de macro-instrução, a tabela de definição de macro contém dois valores. Um valor é o corpo da definição da macro e o outro a sua lista de parâmetros formais.

Para obter a lista de parâmetros formais, o processador de macro verifica se essa lista está presente no campo de operandos da linha. Se estiver, cada membro da lista será associado a uma entrada em outra estrutura de dados auxiliar, a **Tabela da Lista de Argumentos**, que será referenciada na tabela de definição de macro (Figura 4.1).

Figura 4.1 Estruturas de dados no processamento de macros.



Para armazenar o corpo da macro, a linha a seguir é lida e copiada literalmente para a tabela. Verifica-se então se o campo de operação da linha copiada era `ENDM`; se sim, então a definição dessa macro é concluída. Caso contrário, o procedimento repete para a linha seguinte.

O primeiro passo do processador de macro é encerrado quando a pseudo-instrução `END` é encontrada, sinalizando o fim do código fonte.

No segundo passo, cada linha de entrada é novamente lida e o campo de operação é obtido. Se a operação especificada for `MACRO`, esta linha e todas as que a seguem, até aquela que contenha a operação `ENDM`, são ignoradas. Caso contrário, verifica-se se a operação está presente na tabela de definição de macro. Se não estiver, a linha é copiada para o arquivo de saída na sua forma original. Caso contrário, a linha contém uma invocação de macro-instrução que deve ser expandida.

Na expansão da macro, verifica-se se a tabela da lista de argumentos contém algum elemento. Caso haja argumentos, as *strings* com os nomes dos argumentos são associadas, como valores na tabela, aos parâmetros, que são as chaves nessa tabela.

A expansão da macro continua pela leitura de linhas de código da definição, a partir da tabela de definição de macros. Caso a linha contenha no campo de operação a pseudo-instrução `ENDM`, o processo de expansão

está concluído e continua a leitura do arquivo de entrada. Caso contrário, caso o campo de operando contenha algum nome iniciado pelo símbolo `&`, então esse nome é buscado na tabela da lista de argumentos para ser substituído pela *string* correspondente nesta expansão. Após essa substituição (se houver), a linha resultante é passada para o arquivo de saída.

O processamento de macros conclui quando a pseudo-instrução `END` é copiada para o arquivo de saída, o qual então conterá apenas linhas com instruções do processador ou pseudo-instruções, já sem comentários ou macro-instruções.

O processamento de macros pode ocorrer em um único passo caso se restrinja que todas as invocações a uma macro só podem ocorrer no código-fonte após sua definição, quando então os dois passos podem ser combinados.

Criação da tabela de símbolos

Criado o código *assembly* contendo apenas instruções de processador e pseudo-instruções, o passo seguinte é analisar as referências simbólicas contidas no código de forma a permitir a criação do módulo objeto. Nessa etapa do processamento, a atividade principal é a criação da **Tabela de Símbolos** do montador.

Na sua forma mais simples, a tabela de símbolos tem como chaves as *strings* com os nomes simbólicos definidos no programa *assembly*. Símbolos podem ser definidos como resultado de duas situações:

1. Como um rótulo em uma pseudo-instrução `EQU`; neste caso, o valor do símbolo está definido no campo do operando.

2. Como um rótulo em uma outra instrução; neste caso, o valor do símbolo está relacionado à posição de memória dentro do segmento onde ocorre a definição do símbolo.

Para poder criar a tabela de símbolos, o montador deve obter a informação sobre o espaço ocupado pelo código de máquina gerado para cada instrução do processador ou pseudo-instrução que tenha impacto na alocação de memória. Para tanto, o montador faz uso de duas estruturas auxiliares, a tabela de instruções de máquina e a tabela de pseudo-instruções.

A **Tabela de Instruções da Máquina** (ou MOT, de *machine operations table)* contém toda a informação necessária para permitir a tradução de um mnemônico para o código de máquina correspondente. A chave dessa tabela é o código de operação da instrução. Os valores incluem as regras para a geração do código de máquina e o espaço de memória que será ocupado pela instrução, em bytes. O conteúdo dessa tabela é determinado pelo processador para o qual o código está sendo gerado.

A **Tabela de Pseudo-Instruções** (ou POT, de *pseudo-operations table*) tem seu conteúdo definido pelos projetistas do montador. Assim como a MOT, é uma tabela com conteúdo exclusivamente para consulta, não sendo modificado durante a execução do programa. Tem como um dos valores as regras para obter o espaço de memória que deve ser alocado em função da pseudo-instrução. Enquanto muitas das pseudo-instruções, tais como `EQU` ou `END`, não têm impacto sobre a ocupação de memória, `DS` e `DC` têm como efeito a necessidade de reservar e/ou modificar o conteúdo de posições de memória associadas ao programa. Assim, tais pseudo-instruções deverão gerar informação que irá fazer parte do módulo de carregamento gerado pelo montador.

A partir da informação derivada das tabelas do montador é possível saber quanto espaço de memória cada linha de instrução do programa fonte irá ocupar no módulo de carregamento gerado. Será a partir dessa informação que o montador poderá definir qual a posição a ser alocada para cada instrução do programa. Esta informação será mantida em uma variável **contador de localização** (LC, de *location counter*). Essa informação é transiente, ou seja, ela apenas existe durante a execução do montador.

4.1.3 Formato do módulo objeto

Com o processo de montagem, os segmentos do programa *assembly* são convertidos em arquivos no formato de módulo objeto, que serão posteriormente carregados para execução na memória. Tipicamente, um

arquivo objeto contém os seguintes itens de informação:

Cabeçalho: contém a identificação do tipo de arquivo e dados sobre o tamanho do código e eventualmente o arquivo que deu origem ao arquivo objeto;

Código gerado: contém as instruções e dados em formato binário, apropriado ao carregamento;

Relocação: contém as posições no código onde deverá ocorrer mudanças de conteúdo quando for definida a posição de carregamento;

Símbolos: contém os símbolos globais definidos no módulo e símbolos cujas definições virão de outros módulos;

Depuração: contém referências para o código fonte, tais como o número de linha, nomes originais dos símbolos locais e estruturas de dados definidas.

Nem sempre todas essas informações precisam estar presentes no módulo objeto. Por exemplo, um arquivo em formato COM no sistema operacional DOS contém apenas o código gerado. Neste caso, algumas restrições são impostas para garantir essa simplicidade. A posição de carregamento é pré-definida no endereço `0x100` de algum segmento livre e o tamanho do código não deve exceder a capacidade de endereçamento interno a um segmento (64 KBytes). Caso o arquivo exceda esse tamanho, o programador será responsável por garantir a operação correta do programa executável.

4.2 Montagem e carregamento combinados

O esquema mais simples que incorpora a montagem e o carregamento como atividades separadas na execução de programas *assembly* é o esquema **absoluto**. Nesse esquema, o montador gera um arquivo (módulo de carregamento) contendo, além do código de máquina, a informação necessária para que o programa carregador possa carregar o código de máquina nas posições corretas de memória e transferir a execução para o programa carregado.

Esse tipo de esquema é na prática bastante limitado. No entanto, apresenta diversas funcionalidades que permitem introduzir detalhes importantes da operação de carregadores e montadores.

4.2.1 Montadores em dois passos

Uma vez que o código fonte *assembly* já tenha suas macro-instruções expandidas, a etapa de montagem propriamente dita pode ser iniciada. Para essa descrição, considera-se também que na etapa de pré-processamento os comentários foram eliminados do código fonte. Assim, todas as linhas devem conter instruções *assembly* ou pseudo-instruções do montador.

Para introduzir os conceitos relacionados à montagem em dois passos, será inicialmente apresentado um exemplo motivador, apresentando um pequeno código *assembly* e o que seria gerado pelo montador a partir dele. Posteriormente, cada um dos passos do montador será detalhado.

Motivação

Considere como exemplo uma subrotina *assembly* que deverá transferir um valor armazenado em uma posição para outra posição de memória, usando o registrador `D0` como armazenador temporário do dado. A subrotina terá o nome `PGM`, a posição de memória que tem o dado original será rotulada `VALUE` e a posição de memória destino será rotulada `RESULT`.

A listagem a seguir apresenta o código fonte, que é simplesmente uma seqüência de caracteres armazenada em um arquivo texto. Como descrito na Seção 4.1.1, *strings* na primeira coluna denotam rótulos para as posições de memória associadas; a segunda coluna contém *strings* que representam as instruções (mnemônicos), e a terceira coluna contém *strings* representando os argumentos das instruções.

```
1    DATA     EQU      $6000
2    PROGRAM  EQU      $4000
3             ORG      DATA
4    VALUE    DS.W     1
5    RESULT   DS.W     1
6             ORG      PROGRAM
7    PGM      MOVE.W   VALUE,D0
8             MOVE.W   D0,RESULT
9             RTS
10            END      PGM
```

No primeiro passo de execução, o montador deve gerar a Tabela de Símbolos. Para o exemplo acima, a Tabela de Símbolos associada deve conter a seguinte informação:

Símbolo	**Valor**
DATA	$6000
PGM	$4000
PROGRAM	$4000
RESULT	$6002
VALUE	$6000

Há duas situações que devem ser consideradas para possibilitar a geração dessa tabela. A primeira situação, a mais simples, é quando a definição do símbolo é derivada de uma pseudo-instrução `EQU`. Esse caso é o mais simples porque toda a informação necessária para compor a entrada da tabela de símbolos está explícita na instrução. Assim foram definidos os símbolos `DATA` e `PROGRAM` na Tabela de Símbolos.

A segunda situação envolve a definição de símbolos usados como rótulos em outras instruções — no exemplo, o caso de `VALUE`, `RESULT` e `PGM`. Essa situação é mais complexa por necessitar conhecimento da posição no segmento ou na memória que a instrução estará ocupando para poder efetivamente definir seu valor. Para tanto, o contador de localização (LC) deve ser atualizado durante o primeiro passo de acordo com o espaço reservado para cada instrução, informação que deve ser derivada a partir das tabelas de instruções de máquina (MOT) e de pseudo-instruções (POT).

No segundo passo do montador, o código deve ser efetivamente gerado. No caso desse exemplo, dois segmentos serão produzidos.

O primeiro segmento corresponde a uma área de dados, gerada a partir da interpretação das pseudo-instruções `DS` — eventualmente, a interpretação de pseudo-instruções `DC` também poderiam gerar informação para esse segmento, se estivessem presente. O segmento gerado contém a seguinte informação:

Posição	$6000	$6002
Conteúdo	$0000	$0000

O segundo segmento corresponde à área de instruções, contendo o código de máquina associado a cada instrução. Para gerar esse código, o montador teve que *(i)* obter a codificação de máquina para cada instrução e *(ii)* resolver as referências simbólicas presentes nos operandos das instruções.

O conteúdo do segundo segmento, assumindo que endereços absolutos são representados como *long word*, contém:

Posição	$4000	$4002	$4004	$4006	$4008	$400A	$400C
Conteúdo	$3038	$0000	$6000	$31C0	$0000	$6002	$4E75

Na seqüência, serão analisados os procedimentos que o montador deve realizar para possibilitar essa geração de código.

Primeiro passo

Considerando a montagem em dois passos, o primeiro passo pode ser descrito pelo Algoritmo 4.1. Neste primeiro passo, a principal atividade é a manipulação de rótulos de forma a descobrir o símbolo que está sendo criado em cada linha, se for o caso, e manter o controle sobre qual a posição (valor) de definição do símbolo. Esta última informação é obtida a partir da atualização do contador de localização LC a partir da avaliação do tamanho das instruções anteriores.

Algoritmo 4.1 Passo 1 do montador.

```
ASSEMBLER1(source)
 1  LC ← 0
 2  file ← OPENFILE(source)
 3  while ¬ENDOFFILE(file)
 4  do instruction ← READLINE(file)
 5     label ← GETLABEL(instruction)
 6     opcode ← GETOPERATION(instruction)
 7     operand ← GETOPERAND(instruction)
 8     entry ← FINDTABLE(POT, opcode)
 9     if entry ≠ NIL
10       then switch opcode
11               case ORG :
12                    LC ← GETOPERANDVALUE(operand)
13               case END :
14                    ASSEMBLER2(file)
15                    return
16               case default :
17                    if opcode = EQU
18                      then value ← GETOPERANDVALUE(operand)
19                           length ← 0
20                      else value ← LC
21                           length ← GETINSTRUCTIONSIZE(entry, operand)
22                    if label ≠ NIL
23                      then INSERTTABLE(ST, label, value)
24                    LC ← LC + length
25       else entry ← FINDTABLE(MOT, opcode)
26            if entry ≠ NIL
27              then length ← GETINSTRUCTIONSIZE(entry, operand)
28                   if label ≠ NIL
29                     then INSERTTABLE(ST, label, LC)
30                   LC ← LC + length
31              else ERRORMESSAGE( "Invalid opcode" , instruction)
32                   CLOSEFILE(file)
33                   return
```

Nesse procedimento do primeiro passo do montador, o arquivo fonte é manipulado linha a linha. Para

cada linha, o código de operação é analisado para descobrir se a instrução é uma pseudo-instrução (código de operação encontrado na POT) ou uma instrução *assembly* (código de operação encontrado na MOT). Caso o código não esteja em nenhuma das duas tabelas, uma mensagem de erro deve ser apresentada indicando que a instrução não foi reconhecida e o processo deve ser abortado.

Observe que duas buscas devem ser realizadas para cada linha de instrução obtida do arquivo fonte. Como a Tabela de Pseudo-Instruções é bem menor que a Tabela de Instruções de Máquina, a busca é inicialmente realizada na primeira tabela. Apenas se o código buscado não for encontrado na POT a busca será realizada na tabela maior. A eficiência de implementação destas buscas irá se refletir diretamente na eficiência do montador. Por este motivo, é importante que bons algoritmos de busca e de manutenção da informação em tabelas sejam adotados na implementação do montador.

A atualização da Tabela de Símbolos (ST) é efetivada durante o processamento das pseudo-instruções com código de operação `EQU`, `DC` ou `DS` (estas duas tratadas na condição *else* no caso *default*) ou durante o processamento de instruções *assembly* com campo de rótulo não-nulo. Para a pseudo-instrução `EQU`, o rótulo é o nome do símbolo cujo valor deve ser obtido do operando. No caso das pseudo-instruções `DC` e `DS`, o rótulo é o nome do símbolo cujo valor é a posição corrente da instrução.

No processamento das demais pseudo-instruções, nenhum símbolo é criado. Se a pseudo-instrução é `ORG`, apenas o contador de localização deve ser atualizado. A pseudo-instrução `END` deve encerrar o primeiro passo do montador, invocando o segundo passo.

O processamento das pseudo-instruções `ORG` e `EQU` requerem, já nesse passo, uma avaliação do valor do operando. Esse operando pode ser um literal ou um símbolo previamente definido. Caso seja um literal, o valor do operando é obtido a partir da conversão da *string* que representa o valor em alguma base — binária, octal, decimal ou hexadecimal. Caso seja um símbolo previamente definido, o valor é obtido a partir da busca da Tabela de Símbolos. Esse processamento auxiliar é realizado pela rotina GETOPERANDVALUE.

O procedimento do montador deve ainda avaliar o espaço ocupado pela instrução sendo processada, com o fim de atualizar corretamente o contador de localização. Para tanto, utiliza-se informação contida na tabela correspondente *(entry)* e o campo do operando, conforme indicado nas invocações à rotina GETINSTRUCTIONSIZE.

Usando a rotina do exemplo motivador (Seção 4.2.1), é possível acompanhar o processo de atribuição de valores a símbolos que ocorre durante o primeiro passo da montagem. À medida que as linhas de códigos forem lidos, o contador de posição LC vai assumir os valores apresentados na primeira coluna, em hexadecimal:

LC	**Instrução**		
0000	DATA	EQU	$6000
0000	PROGRAM	EQU	$4000
0000		ORG	DATA
6000	VALUE	DS.W	1
6002	RESULT	DS.W	1
6004		ORG	PROGRAM
4000	PGM	MOVE.W	VALUE,D0
4006		MOVE.W	D0,RESULT
400C		RTS	
400E		END	PGM

O valor inicial do LC é 0. A pseudo-instrução `EQU` não ocupa espaço no código gerado e portanto não altera o valor do LC. O efeito das duas primeiras instruções é registrar na Tabela de Símbolos a informação que os símbolos `DATA` e `PROGRAM` têm os valores $6000 e $4000, respectivamente.

A pseudo-instrução `ORG` da terceira linha também não ocupa espaço de código, mas tem efeito sobre o LC — ele passará a registrar a posição de memória para a próxima linha do programa. Assim, LC passará a $6000, que é o valor de `DATA` obtido da Tabela de Símbolos.

A linha seguinte reserva espaço para a variável VALUE. Na Tabela de Símbolos será registrado para o rótulo (o símbolo VALUE) o valor de LC ($6000). Como o sufixo do tamanho para a pseudo-instrução DS é .W, será reservado espaço para uma *word* (dois bytes). Portanto, o LC é incrementado para o valor $6002.

Da mesma forma, na linha seguinte, ao símbolo RESULT será associado o valor $6002 na Tabela de Símbolos e o LC será incrementado para $6004.

Com a pseudo-instrução ORG da sexta linha o LC será alterado para $4000. Deste modo, ao rótulo definido na instrução seguinte, PGM, é associado na Tabela de símbolos o valor $4000.

Para incrementar corretamente o LC, é preciso saber quantos bytes serão ocupados pela instrução da sétima linha. A partir do tratamento dos operandos e assumindo que endereços absolutos são representados em quatro bytes *(long word)*, encontra-se a informação que seis bytes serão usados para esta instrução, de forma que o LC será incrementado para $4006. Da mesma forma, encontra-se que a instrução seguinte também ocupa seis bytes, e o LC é incrementado para $400C.

Finalmente, no processamento da instrução RTS encontra-se que ela ocupa dois bytes, sendo que o valor do LC passa a $400E. A pseudo-instrução END simplesmente indica ao montador o fim do programa fonte, tendo como argumento um endereço da primeira instrução executável.

Segundo passo

Uma vez que todos os símbolos que podem vir a ser usados como operandos das instruções no código-fonte já foram avaliados no primeiro passo do montador, é possível concluir a montagem. A atividade do montador no segundo passo é a geração do código de máquina.

A estrutura básica do segundo passo do montador é similar àquela do primeiro passo. O procedimento deve ler todas as instruções do programa fonte e, para cada linha, gerar o código de máquina correspondente. No processamento de operandos de cada instrução pode ser necessário realizar consultas à Tabela de Símbolos para obter os valores dos operandos simbólicos.

O procedimento preciso para o segundo passo do montador depende do tipo de carregador associado. Nesta seção será apresentada uma estrutura genérica desse algoritmo (Algoritmo 4.2) para descrever as tarefas realizadas nesse passo; detalhes adicionais serão apresentados na seqüência, quando forem apresentadas algumas opções de carregamento.

Nesse procedimento são introduzidas novas rotinas auxiliares de processamento das instruções. A rotina GENERATEMACHINECODE produz o código de máquina associado ao código de operação da instrução e aos seus operandos. Tipicamente, na entrada da tabela correspondente à instrução sendo processada há uma referência para uma função que é capaz de realizar esse processamento. Por exemplo, o "código" gerado para a pseudo-instrução DS pode ser simplesmente uma seqüência de zeros no tamanho reservado pela instrução. Para a pseudo-instrução DC, o código gerado deve corresponder ao processamento dos literais e símbolos do operando. Para qualquer instrução *assembly*, essa rotina utiliza informação da MOT e o processamento dos literais e símbolos do operando para gerar o código de máquina correspondente.

Outra rotina auxiliar é ASSEMBLEMACHINECODE, que posiciona esse código gerado no módulo de carregamento. ASSEMBLECLOSINGCODE tem a função de fechar o módulo de carregamento, podendo adicionalmente realizar outras tarefas associadas ao encerramento do processo de montagem.

4.2.2 Montagem e carregamento *assemble and go*

A forma mais elementar para executar o código de máquina gerado pelo montador é através do esquema *assemble and go*, no qual um único programa de sistema combina a realização das tarefas associadas a um montador e a um carregador.

O esquema *assemble and go*, como o nome sugere ("monta e executa"), combina as etapas de montagem e carregamento em um único programa. Neste caso, não há a criação de um arquivo com o módulo objeto. Quando o código de máquina é gerado pelo montador ele é colocado diretamente na posição de memória

Algoritmo 4.2 Passo 2 do montador.

```
ASSEMBLER2(file)
 1  LC ← 0
 2  REWINDFILE(file)
 3  while ¬ENDOFFILE(file)
 4  do instruction ← READLINE(file)
 5     opcode ← GETOPERATION(instruction)
 6     operand ← GETOPERAND(instruction)
 7     switch opcode
 8        case EQU :
 9             /* nothing */
10        case ORG :
11             LC ← GETOPERANDVALUE(operand)
12        case END :
13             startAddress ← GETOPERANDVALUE(operand)
14             ASSEMBLECLOSINGCODE(startAddress)
15             CLOSEFILE(file)
16             return
17        case default :
18             if opcode = DC ∨ opcode = DS
19               then entry ← FINDTABLE(POT, opcode)
20               else entry ← FINDTABLE(MOT, opcode)
21             length ← GETINSTRUCTIONSIZE(entry, operand)
22             code ← GENERATEMACHINECODE(entry, operand)
23             ASSEMBLEMACHINECODE(LC, code)
24             LC ← LC + length
```

indicada pelo contador de localização. Assim, ao final da montagem o programa executável já está em memória e o montador simplesmente transfere o controle de execução para a primeira instrução executável do código de máquina gerado.

Nesse tipo de esquema, a rotina ASSEMBLEMACHINECODE no passo 2 do montador simplesmente copia o código gerado para a posição de memória indicado pelo contador de localização. A rotina ASSEMBLECLOSINGCODE simplesmente transfere a execução (através de uma instrução de desvio incondicional) para o início do programa montado.

A grande desvantagem do esquema *assemble and go* está no fato de que cada execução do programa requer uma nova montagem, mesmo que o programa não tenha sido alterado. Outra desvantagem está no fato de que dois programas devem obrigatoriamente ocupar a memória principal, o montador e o programa montado. Assim, a utilização desse esquema está restrita a sistemas muito simples, não sendo de utilidade na prática.

4.3 Carregamento absoluto

Uma outra forma simples para contornar as desvantagens do esquema *assemble and go* consiste em separar o processo de montagem do processo de execução do código montado. Nesse caso, o montador gera um módulo de carregamento que não precisa ser regenerado a cada execução. Adicionalmente, as funcionalidades do montador não são necessárias para a execução — assim, o espaço de memória ocupado pelo programa montador pode ser liberado durante a execução do programa montado.

Para o carregamento absoluto o módulo de carregamento contém, além do código objeto, a informação sobre as posições de memória para as quais as linhas de código devem ser carregadas. Uma possível estratégia é associar um registro do arquivo objeto a cada segmento, sendo que o início do registro indica a posição de carregamento e o tamanho do segmento em bytes.

O módulo de carregamento para o carregador absoluto é composto por dois tipos de registro. Todos os registros, exceto um, contêm informação que deve ser transferida para a memória na posição indicada (registro tipo 0). O outro tipo de registro deve ter apenas uma ocorrência no fim do módulo de carregamento, correspondendo à informação do endereço para início da execução do programa (registro tipo 1). Registros do tipo 0 são gerados no segundo passo do montador pela rotina ASSEMBLEMACHINECODE, enquanto o registro do tipo 1 é gerado pela rotina ASSEMBLECLOSINGCODE.

No exemplo do código gerado na Seção 4.2.1, a organização do módulo de carregamento segundo esse esquema seria composto por três registros:

```
0  00006000  4  00000000
0  00004000  E  30380000600031C0000060024E75
1  00004000
```

O primeiro campo de cada registro indica o tipo do registro. O valor `0` neste campo indica que o conteúdo a seguir (quarto campo), de dimensão quatro bytes (informação no terceiro campo) deverá ser transferido à memória a partir da posição $6000 (informação do segundo campo). Similarmente, a informação do segundo registro indica a transferência dos 14 bytes do quarto campo a partir da posição $4000. Finalmente, o último registro tem no primeiro campo o valor `1`, indicando que a execução deverá ser transferida para a posição indicada no segundo campo ($4000).

Para esse esquema de carregamento, o algoritmo do carregador absoluto é apresentado no Algoritmo 4.3.

Algoritmo 4.3 Carregador absoluto.

```
ABSOLUTELOADER(module)
 1  file ← OPENFILE(module)
 2  while ¬ENDOFFILE(file)
 3  do register ← READLINE(file)
 4     type ← GETREGISTERTYPE(register)
 5     address ← GETREGISTERADDRESS(register)
 6     if type = 0
 7       then size ← GETREGISTERLENGTH(register)
 8            code ← GETREGISTERCONTENT(register, size)
 9            MOVETOMEMORY(address, code, size)
10       else GOTO(address)
```

As rotinas auxiliares aqui utilizadas são de funcionalidade simples, servindo para extrair os diversos campos do registro do módulo de carregamento (GETREGISTERTYPE, GETREGISTERADDRESS, GETREGISTERLENGTH e GETREGISTERCONTENT), para transferir código do programa para a posição especificada de memória (MOVETOMEMORY) ou para transferir a execução para o endereço especificado (GOTO).

4.4 Relocação e Ligação

Os esquemas de montagem e carregamento absolutos, por sua simplicidade, não apresentam a flexibilidade necessária ao uso em sistemas operacionais modernos. Uma forte limitação está no fato de que o programador

deve ter acesso direto a posições de memória, especificando exatamente em que região da memória o programa e seus dados serão carregados através da pseudo-instrução `ORG`.

Em sistemas operacionais modernos, tal limitação inviabiliza o uso daqueles esquemas. A memória é um recurso controlado pelo sistema, sendo que o programador não deve estar amarrado a conhecer posições da memória física para que o seu programa funcione corretamente. Por outro lado, desenvolver um programa completamente independente de sua localização é uma atividade complexa, embora possível. A solução é deixar que o software de sistema resolva problemas relacionados com posicionamento do código através da *relocação*.

Outro recurso que também requer a colaboração do montador e do carregador para seu funcionamento é a combinação, ou *ligação*, de módulos interdependentes mas montados independentemente. Neste caso, deve ser possível a partir de um módulo fazer uma referência a um símbolo definido em outro módulo. No esquema de montador absoluto apresentado, tal situação geraria uma condição de erro pelo símbolo não estar definido, ou seja, não ter um endereço associado. Qualquer referência a símbolos externos deveria ser resolvida manualmente pelo programador. Com esquema de montagem e carregamento ajustáveis, o montador recebe a informação de que um símbolo está definido em outro módulo ou de que um símbolo estará sendo referenciado por outro módulo. Esta informação é registrada junto ao módulo objeto para uso pelo carregador, que realiza a resolução destes símbolos entre os módulos envolvidos.

4.4.1 Estruturas de dados adicionais

Os dois tipos de ajustes que podem ocorrer no conteúdo do módulo objeto são:

relocação: ajuste interno ao segmento;

ligação: ajuste entre segmentos distintos.

A atividade de relocação é realizada conjuntamente por montadores e carregadores. Montadores são encarregados de marcar as posições no código objeto passíveis de alteração devido à relocação do código. Carregadores devem reservar um espaço na memória de tamanho suficiente para receber o código de máquina e atualizar suas posições alteráveis a partir da informação sobre sua localização na memória.

No exemplo da Seção 4.2.1, as palavras que começam nas posições $4002 e $4008 do código objeto contêm endereços relocáveis. Verificando o código gerado, observa-se que a posição $4002–$4003 contém uma referência ao endereço $6000, e a posição $4008–$4009 contém uma referência ao endereço $6002. Se o início do segmento de dados for alocado a outro endereço de memória que não $6000, o conteúdo destas posições de memória deverá ser ajustado de acordo com esta mudança. O programa carregador é o encarregado de realizar estes ajustes. Para tanto, o módulo objeto deverá conter informação adicional que permita a realização dos ajustes.

Outro tipo de informação que deverá ser mantida no módulo objeto refere-se a referências aos símbolos externos. Neste caso, há duas situações que podem ser tratadas:

1. o símbolo é referenciado neste segmento, mas é definido em outro segmento; e
2. o símbolo é definido neste segmento e poderá ser referenciado em outro segmento.

A primeira situação é usualmente descrita como uma *referência externa* (ER), enquanto que a segunda situação será descrita como uma *definição local* (LD) de um símbolo externamente referenciável. A informação sobre estes dois tipos de símbolos deverá estar presente no módulo objeto.

Na seqüência, analisaremos o esquema usual de resolução de relocação e referências externas — através de carregadores de *ligação direta*. Para este tipo de carregadores, o montador deverá incluir no módulo objeto estruturas de dados adicionais que incluam a informação necessária. São elas:

Dicionário de Símbolos Externos (ESD): contém todos os símbolos que podem estar envolvidos no processo de resolução de referências entre segmentos: símbolos associados a referências externas (ER), as definições locais (LD) ou a definições de segmentos (SD);

Diretório de Relocação e Ligação (RLD): para cada segmento indica que posições deverão ter seus conteúdos atualizados de acordo com o posicionamento deste e de outros segmentos na memória.

Estas duas estruturas de informação deverão estar presentes no módulo objeto. A partir deles, o carregador de ligação direta deve ser capaz de definir os valores para todos os símbolos com referências entre segmentos e reajustar o conteúdo das posições afetadas pela relocação.

O montador absoluto oferecia como resultado um módulo objeto com dois tipos de registros, registro com código de máquina (tipo 0) e um registro de fim (tipo 1). Um montador trabalhando no esquema de ligação direta deve fornecer dois tipos adicionais de registros além destes, um tipo para ESD e outro para RLD. Uma estrutura simplificada destes tipos de registros é indicada a seguir.

Registros do tipo ESD contêm todos os símbolos definidos no segmento que podem ser referenciados por outros segmentos, além de símbolos referenciados mas não definidos no segmento. Os símbolos locais que podem ser referenciados externamente podem ainda ser de dois tipos, definição do segmento ou definição local. Nos exemplos a seguir, um registro deste tipo apresentará a seguinte estrutura:

1. Tipo do registro (0)
2. Símbolo
3. Tipo de definição (SD — segmento, ou LD — local)
4. Endereço relativo no segmento
5. Comprimento em bytes

Neste modelo simplificado de montagem e carregamento por ligação direta apresentado aqui, definições do tipo ER não receberão tratamento diferenciado.

Registros do tipo TXT contêm o código de máquina, com a informação do endereço relativo incorporada. O formato deste registro é:

1. Tipo do registro (1)
2. Endereço relativo
3. Comprimento em bytes
4. Código de máquina

Registros do tipo RLD indicam quais posições no segmento deverão ter conteúdo alterado de acordo com os endereços alocados aos segmentos, indicando também a partir de que símbolo o conteúdo deverá ser corrigido. O formato deste registro adotado neste texto é:

1. Tipo de registro (2)
2. Posição relativa
3. Comprimento em bytes
4. Símbolo (base de ajuste)

Finalmente, um registro do tipo END especifica o endereço de início de execução para o segmento que contém a “rotina principal”, sendo vazio para os demais segmentos:

1. Tipo de registro (3)
2. Endereço de execução

4.5 Carregamento e ligação combinados

Assim como esquemas primitivos de montagem já incorporavam o processo de carregamento (Seção 4.2), estratégias iniciais de ligação eram combinadas ao processo de carregamento.

Uma das estratégias primitivas adotada em alguns sistemas era o esquema de ligação por **vetor de transferência**, que reservava ao início da área de carregamento um espaço onde os endereços efetivos de rotinas seriam definidos. No código montado, as referências às rotinas eram "desviadas" para posições no vetor de transferência; na posição correspondente, encontrava-se uma instrução de desvio para a posição efetiva de memória onde a rotina havia sido carregada. A limitação desse tipo de programa é que apenas referências externas a rotinas podiam ser resolvidas automaticamente.

Nesta seção apresentaremos as atividades desempenhadas por um **carregador de ligação direta**, outro esquema simples que combina carregamento e ligação em um único programa. Carregadores de ligação direta permitem a resolução a rotinas e dados externos, representando um avanço em relação ao esquema de vetor de transferência.

A operação do carregador de ligação direta será apresentada a partir de um exemplo simples. Considere o seguinte programa, que faz referência a um símbolo externo `DIGIT`:

```
1     MAIN    MOVE.B    DIGIT,D0
2             CMPI.B    #10,D0
3             BLT       ADD_0
4             ADDQ.B    #('A'-'0'-10),D0
5     ADD_0   ADDI.B    #'0',D0
6             MOVE.B    D0,CHAR
7             RTS
8     CHAR    DS.W      1
9             END       MAIN
```

Este programa obtém um valor inteiro entre 0 e 15 de `DIGIT` e irá colocar na variável `CHAR` sua representação ASCII, entre '0' e 'F'.

No segmento onde `DIGIT` é definido, é preciso indicar que este símbolo poderá ser referenciado externamente. Para tanto, a pseudo-instrução `GLOB` é utilizada.

O trecho a seguir ilustra a definição de `DIGIT` em outro segmento:

```
1             GLOB      DIGIT
2     PGM     MOVE.W    VALUE,D0
3             MOVE.W    D0,DIGIT
4             RTS
5     VALUE   DS.W      1
6     DIGIT   DS.W      1
7             END
```

O efeito da pseudo-instrução `GLOB` será a criação de um registro do tipo `ESD` com tipo de definição `LD` — quando a posição relativa do símbolo for definida na tabela de símbolos locais, a informação do registro deverá ser complementada.

O montador deverá gerar o seguinte módulo objeto (com campos separados por pontos) para o segmento `MAIN`:

```
0.'MAIN'.'SD'.00.1C
1.00.6.103900000000
1.06.4.0C00000A
1.0A.2.6D02
```

```
1.0C.2.5E00
1.0E.4.06000030
1.12.6.13C00000001A
1.18.2.4E75
1.1A.2.0000
2.02.4.'DIGIT'
2.14.4.'MAIN'
3.00
```

Neste exemplo, valores numéricos são apresentados em hexadecimal e símbolos na forma de seqüências ASCII — na realidade, o módulo objeto teria apenas a seqüência de bits associada a cada uma destas representações.

O início do módulo objeto contém o diretório de símbolos externos (ESD, registros com primeiro campo com valor 0), o código de máquina gerado (TXT, registros com primeiro campo 1), o diretório de relocação e ligação (RLD, registros com primeiro campo 2) e o registro de fim de segmento (END, com primeiro campo 3). Para o registro de fim de segmento, a posição relativa de execução (posição 00) é especificada.

Similarmente, para o segmento `PGM` o seguinte módulo é gerado:

```
0.'PGM'.'SD'.00.12
0.'DIGIT'.'LD'.10.2
1.00.6.30390000000E
1.06.6.33C000000010
1.0C.2.4E75
1.0E.2.0000
1.10.2.0000
2.02.4.'PGM'
2.08.4.'PGM'
3.
```

4.5.1 Algoritmos do carregador de ligação direta

O carregador de ligação direta recebe como argumentos a lista de módulos a carregar, trabalhando usualmente em vários passos — tipicamente dois.

O carregador apresentado aqui irá realizar três passos. No primeiro passo, ele deve alocar espaço contíguo de memória suficiente para os segmentos. Para saber quanto espaço é necessário, a informação sobre o comprimento de cada segmento — presente em registros tipo ESD, com tipo de definição SD — é obtida. O Algoritmo 4.5.1 apresenta um algoritmo para realizar esta etapa do primeiro passo.

Deve-se observar que, caso a restrição de que o espaço alocado aos módulos não precise ser contíguo, esse primeiro passo é dispensável.

Uma vez determinado qual o endereço inicial de carregamento (IPLA — *Initial Program Load Address*), o carregador inicia a criação de uma **Tabela de Símbolos Externos Globais** (GEST). Para tanto, apenas a informação presente em registros do tipo ESD, com tipos de definição SD e LD, é utilizada. Este segundo passo é apresentado no Algoritmo 4.5.

Nestas apresentações simplificadas do algoritmo de carregamento, o tratamento de erros não é indicado. Na fase de definição da GEST, um possível erro que poderia ser detectado e indicado ao usuário é a duplicação na definição de símbolos na tabela, ou seja, um mesmo símbolo sendo redefinido em segmentos distintos.

No último passo sobre os arquivos de entrada, o carregador irá realizar a transferência do código de máquina para a memória e transferir o controle da execução do programa para o endereço inicial do programa recém-carregado. Este passo é apresentado no Algoritmo 4.6.

Neste passo, o carregador volta a tomar como endereço inicial de carregamento o valor *ipla*, lendo novamente cada módulo objeto na seqüência original. A variável *execPoint* irá registrar a posição de início de execução para o segmento que definir um registro do tipo END com argumento.

Algoritmo 4.4 Primeiro passo do carregador de ligação direta: alocação de memória.

```
DLLOADER1(moduleList)
 1  total ← 0
 2  for each module in moduleList
 3  do file ← OPENFILE(module)
 4     found ← false
 5     repeat
 6           record ← READLINE(file)
 7           type ← GETRECORDTYPE(record)
 8           if type = 'ESD'
 9             then deftype ← GETDEFINITIONFIELD(record)
10                  if deftype = 'SD'
11                    then length ← GETLENGTHFIELD(record)
12                         total ← total + length
13                         found ← true
14       until found
15     CLOSEFILE(file)
16  ipla ← ALLOCMEMORY(total)
```

Algoritmo 4.5 Segundo passo do carregador de ligação direta: definição da GEST.

```
LDLOADER2(moduleList, ipla)
 1  GEST ← CREATETABLE()
 2  segLength ← 0
 3  segStart ← ipla
 4  for each module in moduleList
 5  do file ← OPENFILE(module)
 6     while ¬ENDOFFILE(file)
 7     do record ← READLINE(file)
 8        type ← GETRECORDTYPE(record)
 9        if type = 'ESD'
10          then deftype ← GETDEFINITIONFIELD(record)
11               if deftype = 'S'
12                 then value ← segStart
13                      segLength ← GETLENGTHFIELD(record)
14                 else value ← segStart + GETPOSITIONFIELD(record)
15               symbol ← GETSYMBOLFIELD(record)
16               INSERTTABLE(GEST, symbol, value)
17          else if type = 'END'
18                 then segStart ← segStart + segLength
19                      CLOSEFILE(file)
```

Algoritmo 4.6 Terceiro passo do carregador de ligação direta: transferência de código e início de execução.

```
LDLOADER3(moduleList, ipla, GEST)
 1  execPoint ← ipla
 2  segStart ← ipla
 3  segLength ← 0
 4  for each module in moduleList
 5  do file ← OPENFILE(module)
 6     while ¬ENDOFFILE(file)
 7     do record ← READLINE(file)
 8        type ← GETRECORDTYPE(record)
 9        switch type
10          case 'ESD' :
11               deftype ← GETDEFINITIONFIELD(record)
12               if deftype = 'S'
13                 then segLength ← GETLENGTHFIELD(record)
14                      segSymbol ← GETSYMBOLFIELD(record)
15          case 'TXT' :
16               position ← segStart + GETPOSITIONFIELD(record)
17               length ← GETLENGTHFIELD(record)
18               code ← GETCODEFIELD(record)
19               MOVETOMEMORY(position, code, length)
20          case 'RLD' :
21               position ← segStart + GETPOSITIONFIELD(record)
22               length ← GETLENGTHFIELD(record)
23               defType ← GETDEFINITIONFIELD(record)
24               if deftype = 'X'
25                 then symbol ← GETSYMBOLFIELD(record)
26                      newValue ← FINDTABLE(GEST, symbol)
27                 else base ← FINDTABLE(GEST, segSymbol)
28                      MOVEFROMMEMORY(position, oldValue, length)
29                      newValue ← oldValue + base
30               MOVETOMEMORY(position, newValue, length)
31          case 'END' :
32               position ← GETPOSITIONFIELD(record)
33               if position ≠ NIL
34                 then execPoint ← segStart + position
35               segStart ← segStart + segLength
36               CLOSEFILE(file)
37  GOTO(execPoint)
```

Quando o registro lido é do tipo ESD, o único processamento envolvido é obter o comprimento do segmento de forma a permitir a atualização correta da variável que indica a posição inicial de carga de cada segmento, *segStart*. Esta informação está contida no registro ESD cujo tipo de definição é SD (*Segment Definition*).

Os registros do tipo TXT têm seu conteúdo transferido para a memória principal. Cada campo do registro — posição relativa ao início do segmento, tamanho e conteúdo — é obtido, sendo que o endereço de destino é resolvido tomando por base o valor de *segStart*.

Ao final da transferência, as posições indicadas em registros do tipo RLD têm seu conteúdo ajustado a partir da informação registrada na GEST. Os registros do tipo RLD têm a indicação da posição relativa que deve ser corrigida, sendo que a posição de memória cujo conteúdo será alterado, *position*, é obtida a partir da combinação desta informação com o endereço de início do segmento, *segStart*. O valor pelo qual o conteúdo deverá ser alterado é especificado pelo campo de símbolo presente neste registro — o símbolo é lido do registro e seu valor é obtido a partir de uma busca na GEST. Neste ponto, pode se detectar um erro caso algum símbolo tenha sido referenciado e não definido em nenhum módulo — é onde se processará a informação para referências do tipo ER, não tratadas explicitamente pelo algoritmo.

4.5.2 Exemplo de aplicação

Considere a aplicação desses algoritmos de carregamento com ligação direta ao exemplo apresentado acima. Dois arquivos de módulo objeto são passados como argumentos ao carregador, um para o segmento `MAIN` e outro para o segmento `PGM`.

Na fase de alocação de memória, o carregador obtém a informação que o segmento `MAIN` tem 28 bytes (1C em hexadecimal), enquanto que o segmento `PGM` tem 18 bytes. Assim, o carregador deve requisitar a alocação de uma área de 46 bytes ao sistema operacional.

Considere que o sistema operacional alocou esta área a partir da posição de memória $1000, que será o valor de *ipla*. Na fase de criação da `GEST`, o primeiro registro lido é a definição de `MAIN`, que receberá o valor de *segStart*, inicializado com $1000. Assim, o par `(MAIN,$1000)` é inserido na tabela. A variável *segLength* receberá o valor $1C. Os demais registros deste módulo, de tipo 1 e 2, são ignorados nesta fase. Quando o último registro é lido, de tipo 3 (`END`), o valor de *segLength* é atualizado para $101C.

Quando o segundo módulo é processado neste segundo passo, o primeiro registro é também a definição de um segmento, `PGM`. O par `(PGM,$101C)` é então inserido na `GEST`, sendo *segLength* atualizado para $12. O segundo registro é a definição de um símbolo local. Neste caso, a variável *position* recebe o valor $10 e a variável *newValue* recebe $102C. Assim, a `GEST` receberá a definição do par `(DIGIT,$102C)`. Como para o primeiro módulo, os demais registros são ignorados, até a leitura do registro do tipo 3. Como não há outros módulos a varrer, o segundo passo é encerrado, com o valor de *segStart* sendo atualizado para $102E.

Ao final deste passo, a `GEST` apresentará o seguinte conteúdo:

Símbolo	**Valor**
MAIN	$1000
PGM	$101C
DIGIT	$102C

No terceiro passo, o valor das variáveis *segStart* e *execPoint* são reinicializados para $1000. No processamento do primeiro módulo, o efeito do registro tipo 0 (`ESD`) é simplesmente atribuir a *segLength* o valor $1C. Os registros do tipo 1 são então processados, sendo que para o primeiro deles o conteúdo `103900000000`, de dimensão 6 bytes, é transferido para a posição $1000+$00 de memória. O segundo registro indica a transferência de `0C00000A` (4 bytes) para a posição $1000+$06 de memória, e assim consecutivamente. Ao final destas transferências, a memória apresenta o seguinte conteúdo (em hexadecimal):

Posição	00	02	04	06	08	0A	0C	0E
$1000	1039	0000	0000	0C00	000A	6D02	5E00	0600
$1010	0030	13C0	0000	001A	4E75	0000		

Os registros do tipo 2 (`RLD`) são então lidos. O primeiro deles indica que os 4 bytes a partir da posição relativa $02, ou seja, a *long word* na posição de memória *segStart*+$02 ou $1002, deve ser atualizada pelo valor do símbolo `DIGIT`. A pesquisa na `GEST` indica que este símbolo tem o valor $102C, que somado ao conteúdo anterior da posição ($00000000) resulta em

Posição	*Conteúdo*
$1002	0000
$1004	102C

Observe que este é um endereço externo ao segmento sendo processado, ou seja, este procedimento corresponde a uma tarefa de ligação.

O segundo registro deste tipo no primeiro segmento indica que 4 bytes na posição relativa $14, ou seja, a *long word* na posição de memória $1014, deve ser atualizada pelo valor do símbolo `MAIN`. O conteúdo inicial desta posição é $0000001A, e o valor de `MAIN`, $1000, é obtido da `GEST`. Assim, estas posições de memória são atualizadas para

Posição	*Conteúdo*
$1014	0000
$1016	101A

Observe que este é o endereço de uma variável definida neste próprio segmento, que teve de ser reajustada por relocação.

O processamento do primeiro módulo se encerra com a definição da variável `EXE` para o valor $1000 após a leitura do registro de tipo 3.

Para o segundo módulo, o procedimento se repete, agora com *segStart* com valor $101C. Após a transferência de código a partir desta posição de memória, o processamento de registros de tipo 2 realizarão ajustes de relocação apenas no código deste segmento — a *long word* na posição $101E (*segStart*+2) receberá o valor $0000102A ($0000000E+`PGM`) e a *long word* na posição $1024 (*segStart*+8) receberá o valor $0000102C ($00000010+*segStart*).

Deste modo, o conteúdo completo da memória após o processamento do segundo módulo será

Posição	00	02	04	06	08	0A	0C	0E
$1000	1039	0000	102C	0C00	000A	6D02	5E00	0600
$1010	0030	13C0	0000	101A	4E75	0000	3039	0000
$1020	102A	33C0	0000	102C	4E75	0000	0000	

O carregador encerra seu processamento transferindo o controle (através de instrução JUMP) para a posição *execPoint* ($1000) de memória.

4.6 Ligadores

A estratégia de ligação direta apresentada na Seção 4.5 ilustra bem o princípio de resolução de endereços entre módulos, mas ainda apresenta limitações. Uma das limitações é que o programa carregador é mais complexo que o carregador absoluto, ocupando mais espaço em memória. Como o carregador compartilha memória com o programa sendo executado, menos memória é deixada para a aplicação.

Uma estratégia alternativa é isolar os procedimentos de ligação e de carregamento em programas separados. O programa *ligador* recebe como entrada os diversos módulos a conectar, gerando como saída um único *módulo de carga*. O programa carregador recebe o módulo de carga como entrada, transfere seu código para a memória e realiza apenas os ajustes de relocação de acordo com o endereço base de memória.

Separando as funções entre dois programas, para o exemplo acima um ligador poderia criar o seguinte módulo de carga:

```
0.2E
1.00.6.10390000002C
1.06.4.0C00000A
1.0A.2.6D02
1.0C.2.5E00
1.0E.4.06000030
1.12.6.13C00000001A
1.18.2.4E75
1.1A.2.0000
1.1C.6.30390000002A
1.22.6.33C00000002C
1.28.2.4E75
1.2A.2.0000
1.2C.2.0000
2.02.4
2.14.4
2.1E.4
2.24.4
3.00
```

Este resultado foi obtido essencialmente através das seguintes modificações com relação ao algoritmo do carregador com ligação direta:

1. O registro de início de segmento indica o tamanho total do código de máquina.
2. O endereço inicial de carga, *ipla*, é considerado como sendo 0. Assim, todos os endereços passam a ser relativos ao início do módulo de carga.
3. A saída é enviada a um arquivo (o módulo de carga) ao invés de colocada na memória. A informação de relocação (quais posições de memória deverão ser atualizadas após alocação) deve ser preservada, mas o símbolo de referência não é necessário — será o início do segmento para todos.

O carregador obtém a informação de quanto espaço deve ser alocado do registro inicial (tipo 0), transfere o código (registros tipo 1) para a área de memória alocada e usa a informação do diretório de relocação (registros tipo 2) para ajustar o endereço nas posições indicadas.

Um ligador que produz módulos de carga relocáveis é usualmente denominado um *link-editor*, sendo que os mais elaborados permitem definir diversas seções e a inclusão de comandos específicos para a ligação. Um exemplo é o comando **ld** do Unix. A implementação deste comando (GNU) em Linux incorpora a seguinte informação na sua documentação:

> **ld** combina um número de arquivos objeto e de bibliotecas, reloca seus dados e amarra referências simbólicas. Usualmente o último passo na compilação de um programa é rodar **ld**.
>
> **ld** aceita arquivos *Linker Command Language* escritos em um superconjunto da sintaxe da *AT&T's Link Editor Command Language*, para fornecer controle total e explícito sobre o processo de ligação.

A linguagem de comandos suportada por **ld** controla os seguintes aspectos:

- arquivos de entrada,
- formatos de arquivos,
- *layout* do arquivo de saída,

- endereços de seções,
- posicionamento de blocos comuns.

4.6.1 Bibliotecas

Quando um programador usa em seus programas funções oferecidas pelo sistema, estas funções estão usualmente já montadas, disponibilizadas em formato objeto. Entretanto, ao invés de ter um arquivo objeto para cada função (o que tornaria o número de objetos excessivo) estas funções estão usualmente organizadas na forma de arquivos do tipo *biblioteca*.

Bibliotecas são arquivos que contêm um conjunto de módulos objetos, normalmente agrupados de acordo com sua funcionalidade. A origem do termo "biblioteca" vêm da época dos computadores de grande porte, para os quais as rotinas auxiliares eram mantidas em fitas ou cartões armazenados em salas com prateleiras.

Nesta seção, **bibliotecas estáticas** serão descritas — bibliotecas dinâmicas serão vistas na Seção 4.7. Uma biblioteca estática fornece código objeto que deve ser integrado ao módulo executável antes do momento de execução, durante o processo de ligação.

Em geral, o sistema operacional apresenta utilitários para manipular arquivos tipo biblioteca. Em Unix (Linux), o utilitário `ar` é utilizado para criar, manter e extrair módulos de arquivos de bibliotecas estáticas. Por exemplo, se um módulo objeto `arqmat.o` tiver sido criado com a linha de comando

```
> gcc -c arqmat.c
```

esse módulo pode ser incluído em uma biblioteca `libmy.a`, criada pelo usuário, com a linha de comando

```
> ar -r libmy.a arqmat.o
```

A chave `-r` indica que ocorrerá uma troca *(replacement)* do módulo, caso houvesse uma versão anterior já armazenada na biblioteca; caso contrário, o módulo é acrescentado à biblioteca.

Se essa for a primeira operação com essa biblioteca, ela será criada pelo programa `ar`. Nesse caso, uma solicitação de listar o conteúdo (com a opção `-t`) mostrará que apenas esse módulo está presente:

```
> ar -t libmy.a
arqmat.o
```

Novos módulos podem ser similarmente incluídos:

```
> ar -r libmy.a convexp.o
> ar -t libmy.a
arqmat.o
convexp.o
```

A estrutura típica de um arquivo do tipo biblioteca nesse tipo de sistema operacional é composta por uma identificação do tipo de arquivo (em geral, a *string* `!<arch>\n`), um diretório de membros do arquivo e por uma seção com o conteúdo de cada membro do arquivo.

O diretório de membros do arquivo é composto por uma série de cabeçalhos de membro, sendo que cada cabeçalho de membro contém:

- o nome do membro;
- a data de modificação;
- a identificação do usuário e grupo criador do módulo;
- as permissões de acesso para o módulo;

- o tamanho do módulo em bytes; e
- um terminador de cabeçalho, usualmente a seqüência com os dois caracteres `\` e `n` para indicar um "fim de linha".

A estrutura de um arquivo do tipo biblioteca pode ser usado para agregar qualquer tipo de conteúdo, mas usualmente apenas módulos objetos são agrupados em bibliotecas.

No processo de ligação, além dos módulos objetos gerados a partir dos arquivos fontes originais, o programador pode especificar arquivos do tipo biblioteca. Inicialmente, o ligador irá resolver as referências que puderem ser estabelecidas a partir dos módulos objetos fornecidos. Se, ao final dessa etapa, ainda houver referências não-resolvidas, o ligador procura pela definição dos símbolos dentro das bibliotecas. Ao encontrar o cabeçalho do módulo especificado, o ligador obtém dali toda a informação necessária para extrair apenas o módulo desejado e assim integrá-lo ao módulo de carga executável.

Arquivos de biblioteca são extensamente utilizados, embora nem sempre de forma explícita. Por exemplo, na implementação GNU para o compilador C a biblioteca `libc.a`, armazenada no diretório `/usr/lib`, contém as funções da biblioteca padrão da linguagem. Como essas funções são amplamente utilizadas (tal como a rotina `printf`), o programador não precisa explicitar para o ligador que essa biblioteca deverá ser utilizada para a resolução de símbolos — o próprio compilador `gcc` irá integrar essa biblioteca ao processo de ligação.

Quando uma outra biblioteca tiver de ser utilizada, contendo por exemplo rotinas matemáticas (em Unix, na biblioteca `libm.a`) ou rotinas associadas a outros pacotes ou aplicativos (bancos de dados, interfaces gráficas), é preciso passar essa informação ao ligador. No caso do ligador `ld`, há duas chaves relacionadas ao fornecimento dessa informação — essas chaves podem ser especificadas para o compilador, que as repassa ao programa ligador. A chave `-l`*xxx* indica que a biblioteca cujo nome é `lib`*xxx*`.a` deve ser incorporada ao processo de resolução de referências.

Por exemplo, considere o seguinte programa que incorpora uma rotina matemática — no caso, `cos` para o cálculo do cosseno de um valor real especificado na linha de comando:

```
1   // calccos.c: calcula o cosseno do valor especificado
2   #include <math.h>
3   #include <stdlib.h>
4   #include <stdio.h>
5
6   int main(int argc, char *argv[]) {
7     double valor, result;
8
9     if (argc != 2) {
10      fprintf(stderr, "%s requer um argumento numérico\n", argv[0]);
11      return 1;
12    }
13    valor = atof(argv[1]);
14    result = cos(valor);
15    printf("Cosseno de %lf é %lf\n", valor, result);
16    return 0;
17  }
```

Se o correspondente arquivo de biblioteca com rotinas matemáticas, que contém o código objeto para a rotina `cos`, não for especificado, um erro de ligação será gerado:

```
> gcc calccos.c
/tmp/ccKOiW4b.o: In function ‘main’:
```

```
/tmp/ccKOiW4b.o(.text+0x54): undefined reference to ‘cos’
collect2: ld returned 1 exit status
```

Para incluir a ligação das rotinas matemáticas, o código deve ser compilado e ligado com a inclusão da chave `-lm`, como em

```
> gcc calccos.c -lm
```

A outra chave associada à especificação de bibliotecas é `-L`, que especifica um diretório onde arquivos do tipo biblioteca estarão armazenados, caso seja necessário fazer essa busca em um diretório distinto dos usados por padrão pelo sistema operacional — tipicamente, os diretórios `/lib`, `/usr/lib` e `/usr/local/lib`.

4.7 Carregamento e Ligação Dinâmicos

Os esquemas de ligação e carregamento apresentados até o momento assumem que o módulo executável, uma vez carregado a uma área da memória principal, será o "proprietário" desta área até o fim de sua execução. Em sistemas multiusuários mais recentes, não é isto o que ocorre. Programas em execução podem ser retirados da memória *(swaped-out)* e depois retornar à memória *(swap-in)* em outra posição diferente daquela na qual estava executando inicialmente.

Para atender a este tipo de necessidade, utilizam-se esquemas de ligação e carregamento dinâmico. O princípio básico destes esquemas é que referências a endereços (de dados ou de instruções) são mantidos na forma relativa até o momento em que eles são realmente necessários, ou seja, até o momento de execução da instrução que contém esta referência. Usualmente, esta funcionalidade deve ter parte suportada em hardware de modo a não haver degradações sensíveis de desempenho.

Há dois esquemas básicos de ligação dinâmica, em tempo de carregamento *(load-time)* ou em tempo de execução *(run-time)*. Na ligação dinâmica em tempo de carregamento, o módulo de carga primário (módulo da aplicação) é inicialmente transferido para a memória. Qualquer referência neste módulo para módulos externos (módulos alvos) faz com que o carregador procure cada módulo alvo, carregue-o para a memória e altere as referências para um endereço relativo em memória a partir do início do módulo da aplicação.

Entre as vantagens neste esquema de carregamento pode-se destacar:

- facilidade de atualização de versões de módulo alvo sem alterar a aplicação;
- facilidade no suporte ao compartilhamento de módulos alvo entre aplicações distintas — se o sistema operacional detectar que um módulo alvo já está em memória, uma única cópia pode ser mantida em memória (contanto que o conteúdo do módulo alvo não seja alterado pela aplicação).

A diferença para a ligação dinâmica em tempo de execução está no fato de que o carregamento e a resolução de referências são retardados até o momento em que a instrução com a referência ao módulo externo é executada. As referências a módulos externos continuam presentes na aplicação, mas se em alguma execução a lógica de fluxo de controle fizer com que aquela referência não seja executada, então o módulo alvo não será carregado à memória por aquela aplicação. As vantagens descritas para o esquema de ligação dinâmica em tempo de carregamento continuam válidas também neste caso.

Em versões mais recentes do sistema operacional Linux os módulos objeto e de carga estão principalmente em formato ELF *(Executable and Linking Format)*. Bibliotecas para este tipo de arquivos são denominadas bibliotecas dinâmicas ainda ou arquivos de objetos compartilhados, que são diferenciadas das bibliotecas estáticas por sua extensão — estáticas têm extensão `.a` (archive) e dinâmicas têm extensão `.so` *(shared objects)*.

Sob o ponto de vista do programador usuário, não há diferença no procedimento para uso de rotinas em bibliotecas estáticas ou dinâmicas — da mesma forma, rotinas em bibliotecas padrões são automaticamente buscadas e outras bibliotecas deverão ser especificadas através da chave `-l`. A diferença está na forma de

operação interna do ligador e carregador, que utiliza a interface de programação (API) associado ao formato ELF.

ELF apresenta uma API que pode ser utilizada em programas do sistema desenvolvidos em C para manipular objetos em biblioteca compartilhadas durante a execução de um programa. Essas rotinas, disponibilizadas através da biblioteca `libdl.so`, são:

```
#include <dlfcn.h>
void       * dlopen  (const char *filename, int flag);
const void * dlsym   (void *handle, const char *symbol);
int          dlclose (void *handle);
const char * dlerror (void);
```

A rotina `dlopen` disponibiliza uma biblioteca dinâmica para o programa em execução — em outros termos, as rotinas no arquivo especificados são mapeadas para o espaço de endereçamento do processo em execução. O seu valor de retorno é um ponteiro (*handle*), utilizado nas chamadas posteriores de manipulação da biblioteca. O argumento `flag` indica quando deverá se dar o carregamento. Se tiver o valor `RTLD_NOW` (uma constante definida no arquivo `dlfcn.h`), o carregamento deverá ser imediato, ou seja, ao retornar dessa rotina a biblioteca já estará carregada na memória. Caso o valor especificado seja `RTLD_LAZY`, o carregamento será postergado até o momento em que houver (e se houver) a um símbolo dessa biblioteca. Em caso de erro, o apontador nulo será retornado.

A rotina `dlsym` retorna o endereço do símbolo especificado (variável ou função) que está disponível na biblioteca compartilhada que foi aberta. Em caso de erro, o apontador nulo será retornado.

Quando a biblioteca compartilhada não é mais necessária, ela é liberada através da invocação da rotina `dlclose`, que retorna 0 em caso de sucesso.

Em qualquer situação de erro, a rotina `dlerror` pode ser invocada para obter uma *string* com o diagnóstico do erro.

O exemplo a seguir ilustra como a função `cos` pode ser dinamicamente carregada da biblioteca `libm.so` usando ELF em linux:

```
1  #include <dlfcn.h>
2  #include <stdio.h>
3  int main(int argc, char *argv[]) {
4    void *handle;
5    double (*cosine)(double);
6    char *error;
7    handle = dlopen("/lib/libm.so.5", RTLD_LAZY);
8    if (!handle) {
9      fputs (dlerror(),stderr);
10     return 1;
11   }
12   cosine = dlsym(handle, "cos");
13   if ((error = dlerror()) != 0) {
14     fputs(error, stderr);
15     return 1;
16   }
17   printf("%f\n", (*cosine)(2.0));
18   dlclose(handle);
19 }
20
```

Para criar uma biblioteca compartilhada, inicialmente é preciso gerar um módulo objeto que possa ser carregado dinamicamente. Para tanto, o código gerado deve ser independente de posição. O compilador `gcc`

permite a criação desse tipo de código de forma automática, através do uso da chave `-fPIC` (de *position-independent code*):

```
> gcc -fPIC -c arqmat.c
> gcc -fPIC -c convexp.c
```

Para criar a biblioteca dinâmica contendo os objetos compartilhados, o próprio compilador é utilizado:

```
> gcc -shared -o libmyd.so arqmat.o convexp.o
```

A chave `-shared` indica para o programa `gcc` que o programa que está sendo gerado (indicado pela opção `-o`) é uma biblioteca compartilhada cujos módulos podem ser carregados e ligados dinamicamente. Caso algum desses módulos faça referências a arquivos em outras bibliotecas dinâmicas, é possível tornar transparente para o usuário a necessidade de se carregar essa outra biblioteca especificando-a no momento da criação. Por exemplo, se `arqmat.o` faz uso de rotinas em `libm.so`, a linha de comando

```
> gcc -shared -o libmyd.so arqmat.o convexp.o -lm
```

fará com que `libm.so` seja automaticamente carregada quando `libmyd.so` for especificada.

O padrão em sistemas operacionais modernos é utilizar arquivos compartilhados com carregamento e ligação dinâmica. O compilador `gcc` inclui a opção `-static` caso seja necessário criar um executável ligado estaticamente, mudando assim o comportamento padrão.

4.8 Exercícios

4.1 Das instruções em Assembly 68K abaixo, indique quais são válidas e quais não são, justificando suas resposta. Para aquelas que forem válidas, apresente o código de máquina em hexadecimal. A descrição da instrução `MOVE` encontra-se no Apêndice B.

(a) `MOVE.L D1,#10`

(b) `MOVE.W 16(A4),D2`

(c) `MOVE.B #2056,D3`

4.2 Explique qual a diferença entre

(a) o resultado gerado por um montador absoluto e um montador de ligação direta

(b) ajuste de relocação e ajuste de ligação

(c) ligação dinâmica em tempo de carga e em tempo de execução

4.3 Justifique se as afirmações abaixo são verdadeiras ou falsas:

(a) A Tabela de Símbolos gerada por um montador é uma estrutura de dados usada internamente pelo montador, não sendo nunca incorporada ao módulo objeto gerado.

(b) Uma das vantagens do esquema de ligação dinâmica é a redução de tamanho de módulos objetos, uma vez que nestes a incorporação de códigos de outras rotinas é substituída por referências a estas rotinas.

4.4 Um montador de ligação direta aplicado a dois arquivos em linguagem simbólica do 68K gerou os seguintes módulos objetos:

Módulo 1	**Módulo 2**
0.’MAIN’.’S’.0000.001A	0.’CALC’.’S’.0000.0006
0.’RESULT’.’L’.0018.0002	1.0000.02.2200
1.0000.06.203900000014	1.0002.02.9081
1.0006.06.4EB900000000	1.0004.02.4E75
1.000C.06.33C000000018	3.00
1.0012.02.4E75	
1.0014.04.00004E75	
2.0002.04.’MAIN’	
2.0008.04.’CALC’	
2.000E.04.’MAIN’	
3.02.0000	

Passados como argumentos nessa ordem (módulo 1 seguido de módulo 2) para um carregador de ligação direta, obteve-se o endereço inicial de carga (IPLA) $0200.

(a) Qual o conteúdo da Tabela de Símbolos Externos Globais (GEST) gerada pelo carregador?

(b) O diagrama abaixo é um mapa de conteúdo da memória após o carregamento *sem* os ajustes de ligação e relocação. Indique neste mapa quais posições são ajustadas pelo carregador e qual o novo conteúdo destas posições.

	0	2	4	6	8	A	C	E
020-	2039	0000	0014	4EB9	0000	0000	33C0	0000
021-	0018	4E75	0000	4E75	0000	2200	9081	4E75

