Laboratório de Pesquisa em Redes e Multimídia

Nível da Linguagem de Montagem

(Aula 15)

Linguagem de Montagem

Universidade Federal do Espírito Santo
Departamento de Informática

Roberta Lima Gomes - LPRM/DI/UFES
Sistemas de Programação I – Eng. Elétrica
2007/2

Introdução

- Tradutores
  - Programas que convertem um programa usuário escrito em uma linguagem para a outra.
  - Linguagem-fonte para Linguagem-alvo.
- **Montador**
  - Linguagem-fonte é uma **Linguagem de Montagem**
    - representação essencialmente simbólica
  - Linguagem-alvo é uma linguagem de máquina numérica
- **Compilador**
  - Linguagem-fonte é uma linguagem de alto nível
  - Linguagem alvo pode ser
    - Uma linguagem de máquina numérica
    - Uma linguagem de montagem (representação simbólica dessa linguagem de máquina)

Linguagem de Montagem (1)

- Uma **Linguagem de Montagem** pura, sem aditivos (macros, pseudo-instruções), é aquela na qual cada comando produz exatamente uma **instrução de máquina**.
- Nesse caso, existe uma correspondência de um-para-um nas instruções da Linguagem de Montagem e a Linguagem das Instruções de Máquina.
  - Por isso a tradução do código de montagem em código de máquina não é chamada compilação, e sim de montagem
- Então, um programa em linguagem de montagem **pura** com *n* linhas seria traduzido (por um montador) em um programa em linguagem de máquina de *n* palavras.

Linguagem de Montagem (2)

- É mais simples a programação em linguagem de montagem (com nomes simbólicos e ferramentas auxiliares) do que fazê-lo em linguagem de máquina pura (hexadecimal/binário).
  - Ex: É muito mais fácil para uma pessoa se lembrar de escrever ADD para adicionar um numero a outro do que lembrar o seu valor equivalente em hexadecimal para programar na linguagem de máquina.
- Programar em linguagem de montagem, em relação a programar em alto nível:
  - é uma tarefa mais difícil.
  - consome muito mais tempo.
  - tem depuração e manutenção também mais difíceis e demoradas
- Por que alguém escolheria programar em linguagem de montagem?

Linguagem de Montagem (3)

- Principal vantagem: performance!
  - Produz-se código de máquina menor e muito mais rápido
    - O programador decide quais instruções de máquina usar diretamente
  - A maioria das aplicações embarcadas (tipo: código de cartões inteligente, rotinas de BIOS, etc..) exigem velocidade de processamento.
- Problemas: portabilidade
  - Utilizado em apenas algumas famílias de máquinas
  - As linguagens de alto nível podem rodar (potencialmente) em diversas máquinas com arquiteturas diferentes

Linguagem de Montagem (4)

- Existem portanto quatro grandes motivos para se aprender a programar em linguagem de montagem:
  - (a) Para os experts no assunto, essa linguagem permite o desenvolvimento de programas de alto desempenho
  - (b) Programas são escritos com rotinas mais otimizadas, o que permite que programas mais complexos possam ser executados em dispositivos com pouca capacidade computacional..
  - (c) São muito utilizadas em compiladores: , a compreensão da linguagem de montagem torna-se inevitavelmente necessária para se entender o funcionamento desse tipo de software
  - (d) É indispensável para quem quer obter uma maior aproximação com a máquina, via programação.
    - Por exemplo, as rotinas do sistema operacional para tratamento das interrupções são diretamente desenvolvidas em linguagem de montagem

Linguagem de Montagem (5)

- Aplicações potenciais a serem implementadas diretamente em linguagem de montagem:
  - Códigos de um cartão inteligente
  - o código em um telefone celular
  - o código dos drivers dos dispositivos
  - rotinas da BIOS
  - os loops internos de aplicações dependentes de performance

Linguagem de Montagem (5)

- Metodologia mista (Sintonização):
  - Desenvolve-se o programa em linguagem de alto nível (tarefa mais rápida e simples)
  - Depois de compilado (código em linguagem de montagem gerado) otimiza-se o código nos trechos de programas que consomem mais tempo de processamento
    - Para aumentar a eficiência do programa

	Número de Programadores-ano para desenvolver o programa	Tempo de Execução do Programa em segundos
Linguagem de Montagem	50	33
Linguagem de alto nível	10	100
Metodologia Mista depois da Sintonização		
Código Crítico: 10%	6	30
Outras Partes do Código: 90%	9	10
	---	----
Total	15	40

Formato de Comando (1)

- Para cada tipo de máquina existe um formado de comando, mas todos são ligeiramente parecidos...

Label	Opcode	Operands	Comments
FORMULA:	MOV	EAX,I	; register EAX = I
	ADD	EAX,J	; register EAX = I + J
	MOV	N,EAX	; N = I + J
I :	DW	3	; reserve 4 bytes initialized to 3
J :	DW	4	; reserve 4 bytes initialized to 4
N :	DW	0	; reserve 4 bytes initialized to 0

$N = I + J.$

Pentium 4

Label	Opcode	Operands	Comments
FORMULA	MOVE.L	I, D0	; register D0 = I
	ADD.L	J, D0	; register D0 = I + J
	MOVE.L	D0, N	; N = I + J
I	DC.L	3	; reserve 4 bytes initialized to 3
J	DC.L	4	; reserve 4 bytes initialized to 4
N	DC.L	0	; reserve 4 bytes initialized to 0

Motorola 680x0

Formato de Comando (2)

- Os comandos da linguagem do montador possuem quatro partes:
  - Um campo para o label (rótulo)
    - Serve para identificação de linha, ou como uma macro
  - Um campo dedicado à operação (código de operação)
    - Onde são colocados os comandos da linguagem para fazer algo
  - Um campo para os operandos
    - Onde são colocados as variáveis e valores a serem utilizados na ação
  - Um campo para os comentários
    - Serve apenas para que o autor lembre-se do porque escreveu
    - Programas feitos nesse nível de linguagem possuem grande dificuldade quanto a sua leitura!

Pseudo-Instruções (1)

- Ou **Diretivas do Montador**
  - Comandos para o próprio Montador.

Algumas pseudo-instruções disponíveis no montador do Pentium 4 (Microsoft MASM)

Pseudoinstr	Meaning
SEGMENT	Start a new segment (text, data, etc.) with certain attributes
ENDS	End the current segment
ALIGN	Control the alignment of the next instruction or data
EQU	Define a new symbol equal to a given expression
DB	Allocate storage for one or more (initialized) bytes
DD	Allocate storage for one or more (initialized) 16-bit halfwords
DW	Allocate storage for one or more (initialized) 32-bit words
DQ	Allocate storage for one or more (initialized) 64-bit double words
PROC	Start a procedure
ENDP	End a procedure
MACRO	Start a macro definition
ENDM	End a macro definition
PUBLIC	Export a name defined in this module
EXTERN	Import a name from another module
INCLUDE	Fetch and include another file
IF	Start conditional assembly based on a given expression
ELSE	Start conditional assembly if the IF condition above was false
ENDIF	End conditional assembly
COMMENT	Define a new start-of-comment character
PAGE	Generate a page break in the listing
END	Terminate the assembly program

Pseudo-Instruções (2)

- São instruções executadas pelo montador
  - Elas não ficam presentes no código traduzido (linguagem de máquina)
- O programador pode, por exemplo, requisitar a reserva de um determinado espaço de memória para o programa
  - DW e DQ: reservam 4 bytes e 8 bytes, respectivamente, para uma determinada variável

```
VAR1: DW 18      ; Aloca 4 bytes para a variável VAR1, e a inicializa com o valor 18.
VAR2: DQ 7       ; Aloca 8 bytes para a variável VAR2, e a inicializa com o valor 7.
```

  - EQU: semelhante ao *define* de C, diz que a expressão deverá ser substituída por determinado valor, no momento em que estiver ocorrendo a tradução

```
INIT EQU 1807          ; Define a palavra INIT como sendo o valor 1807.
PLUS EQU INIT + 2503   ; Define PLUS como a soma entre INIT e 2503.
```

MACROS (1)

- Macros são utilizadas na linguagem de montagem, assim como os métodos ou sub-rotinas são utilizadas em linguagens de mais alto nível
- São uma solução simples e eficiente para o problema de se repetir uma mesma seqüência de instruções no decorrer de um programa
- Para isso, cria-se funções com labels e limitações via pseudo-códigos
- Para usar então essas funções é apenas necessário referenciar o label em uma instrução
- O montador, ao encontrar a definição de uma macro, armazena numa tabela de definição de macros
- Cada vez que a referência aparece num trecho do código, ela é substituída por essa função (conjunto de instruções)

MACROS (2)

- Define-se um Label (nome) e associa esse nome a uma seqüência de instruções
- Dessa forma, diminui-se o tamanho e aumenta-se a legibilidade do código fonte.

```
MOV     EAX,P
MOV     EBX,Q
MOV     Q,EAX
MOV     P,EBX

MOV     EAX,P
MOV     EBX,Q
MOV     Q,EAX
MOV     P,EBX
```

```
SWAP    MACRO
        MOV EAX,P
        MOV EBX,Q
        MOV Q,EAX
        MOV P,EBX
        ENDM

        SWAP

        SWAP
```

Figure 7-4. Assembly language code for interchanging P and Q twice. (a) Without a macro. (b) With a macro.

MACROS (3)

- Apesar dos montadores apresentarem jeitos diversos de declararem macros, todos requerem três segmentos bem definidos:

  (1) um cabeçalho, contendo o nome da macro em definição;

  (2) o corpo da macro, com as instruções em linguagem de montagem pura; e

  (3) uma pseudo-instrução, indicando o fim da definição da macro.

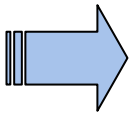
```
MATH   MACRO
       MOV    EAX,18 ; Move o valor 18 para
                       para o registrador EAX
       MOV    EBX,25 ; Move o valor 25 para o
                       registrador EBX
       ADD    EAX,EBX; Efetua EAX=EAX+EBX
       MOV    EBX,EAX; Move o valor de EAX p/EBX
       ENDM
```

MACROS (4)

- A substituição do código da macro fonte chama-se **Expansão de Macros**
  - Ocorre durante o processo de montagem
  - Não há como distinguir, em linguagem de máquina, se o programa utilizou ou não macros
- Qual a diferença do uso de macros para chamadas a procedimentos?

```
MOV     VAR1,99
MATH
MOV     VAR2,32
MATH
MATH
```

Utilização de macros



```
MOV     VAR1,99
MOV     EAX,18
MOV     EBX,25
ADD     EAX,EBX
MOV     EBX,EAX
MOV     VAR2,32
MOV     EAX,18
MOV     EBX,25
ADD     EAX,EBX
MOV     EBX,EAX
MOV     EAX,18
MOV     EBX,25
ADD     EAX,EBX
MOV     EBX,EAX
```

Código expandido da macro

MACROS (5)

- As macros são bastante úteis quando o programador deseja repetir o exato segmento de código diversas vezes ao longo de seu programa.
- Na maioria das vezes, porém, deseja-se realizar blocos de instruções semelhantes, mas não exatamente iguais!
- Solução: os programadores podem utilizar **Macros Parametrizadas**
  - Em suas chamadas, devem possuir variáveis

MACROS (6)

- Exemplo de definição de uma macro com parâmetros

```
MATH2 MACRO V1,V2
      MOV   EAX,V1 ; Move o valor de V1 para
                     para o registrador EAX
      MOV   EBX,V2 ; Move o valor de V2 para o
                     registrador EBX
      ADD   EAX,EBX; Efetua EAX=EAX+EBX
      MOV   EBX,EAX; Move o valor de EAX p/EBX
      ENDM
```

- Chamada de uma macro com parâmetros

```
MOV   VAR1,99
MOV   VAR2,32
MATH2 VAR1,VAR2
```

Montadores (1)

- **Montadores (*Assemblers*)**
  - Montam um programa em linguagem de máquina a partir de sua versão em linguagem de montagem, ou linguagem "assembly".

- Processo geral de Montagem:
  - Acham um endereço inicial para o programa (normalmente 0);
  - Transformam cada parte dos comandos em linguagem *assembly* em *opcode*, número de registrador, constante, etc;
  - Convertem pseudo-instruções para o conjunto de instruções equivalente;
  - Convertem macros no conjunto de instruções e dados equivalentes;
  - Escrevem o programa em linguagem de máquina em um arquivo com as instruções ordenadas e com os endereços indicados por elas (inicialmente especificados como *labels*) já convertidos para números quando possível;

Montadores (2)

- Problema da referência posterior
  - Surge quando uma linha faz referências a estruturas ainda não montadas pelo montador
- Solução: **Montador de Dois Passos**
- Passo 1
  - O montador coleta as definições de símbolos, inclusive os labels de comandos, e armazena em uma **tabela de símbolos**
  - OBS: a maioria dos montadores usa no mínimo três tabelas:
    - a **tabela de símbolos**, a tabela de pseudocódigos e a tabela de códigos de operação
- Passo 2
  - Cada linha do código é substituída pela equivalente da linguagem-alvo
  - Quando o montador se depara com uma referência a uma macro/símbolo, ele usa as tabelas criadas no passo 1 p/ substituí-lo
  - É nesse passo, também, que são diagnosticados os erros de sintaxe e de atribuições inválidas no código fonte.

Montadores (3)

- Passo 1:
  - É usada uma variável conhecida como ILC (Instruction Location Counter - Contador de Posições de Instruções)
  - O passo Um termina com a leitura da pseudo-instrução END

Label	Código de Operação	Operandos	Comentários	Tamanho	ILC
MARIA:	MOV	EAX,I	EAX = I	5	100
	MOV	EbX,J	EBX = J	6	105
ROBERTA:	MOV	ECX,K	ECX = K	6	111
	IMUL	EAX,EAX	EAX = I*I	2	117
	IMUL	EBX,EBX	EAX = J*J	3	119
	IMUL	ECX,ECX	EAX = K*K	3	122
MARILYN:	ADD	EAX,EBX	EAX = I*I+J*J	2	125
	ADD	EAX,ECX	EAX = I*I+J*J+K*K	2	127
STEPHANY:	JMP	DONE	Desvio para lab DONE	5	129

...

Montadores (3)

- Passo 1:
  - Tabela de Símbolos para o programa anterior

Símbolo	Valor	Outra Informacoes
MARIA	100	
ROBERTA	111	
MARYLIN	125	
STEPHANY	129	

...

Montadores (3)

- Montador gera um **Arquivo Objeto**
  - Em geral, não pode ser executado diretamente pela máquina por conter referências a sub-rotinas e dados especificados em outros arquivos.

- Arquivo objeto típico (sistema operacional Unix) contém seis seções distintas:
  - *header* – descreve as posições e os tamanhos das outras seções;
  - *text segment* – contém o código em linguagem de máquina;
  - *data segment* – contém os dados especificados no arquivo fonte;
  - *relocation information* – lista as instruções e palavras de dados que dependem de endereços absolutos;
  - ***symbol table*** – associa endereços com símbolos especificados no arquivo fonte e lista os símbolos não resolvidos (que estão ligados a outros arquivos);
  - *debugging information* – contém uma descrição sucinta de como o arquivo foi compilado, de modo a permitir que um programa *debugger* possa encontrar instruções e endereços indicados no arquivo fonte.

Ligadores (1)

- Muitas vezes, um programa pode ser **decomposto** em **várias rotinas**, cada uma responsável por aspectos específicos da computação a ser executada pelo programa.
- Exemplo: um programa para computar o total da folha de pagamento de uma empresa pode ser divido em:
  - Uma rotina que calcule a salário de cada funcionário, levando em consideração o fundo de garantia, contribuição ao INSS, etc,
  - Uma outra que compute o total a ser despendido para o pagamento de todos os funcionários.
- Normalmente essas rotinas são definidas em módulos separados
- Esses módulos, precisam ser ligados (linkados) para que se gere o programa binário executável.
  - Para gerar um programa executável a partir de um ou mais arquivos objeto temos que usar um *linker*.

Ligadores (2)

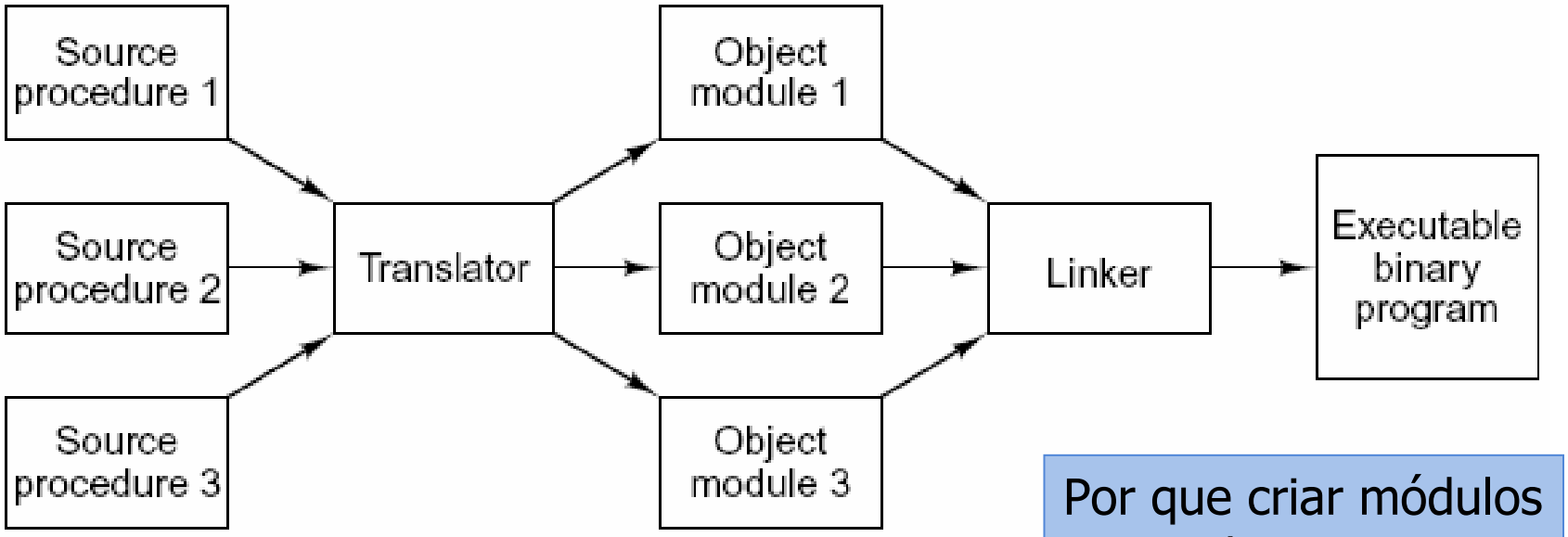
- Exemplo:

```
float calcula_um_salário (float salário_base){
    float salário_total;
    salário_total = salário_base + salário_base * FATOR_FGTS +
                    salário_base * FATOR_INSS;
    return (salário_total);
}

float salário_total (float salário_base[], int numero_funcionários){
    int i;
    float total;
    total = 0.0;
    for (i = 0; i < numero_funcionários; i++)
      total = total + calcula_um_salário (salário_base [i]);
    return (total);
}
```

Ligadores (3)

- As rotinas podem estar em arquivos separados.
- **Ligadores** são programas especiais que recebem como entrada os arquivos objeto correspondentes a estes arquivos e geram como saída o programa final em linguagem de máquina.



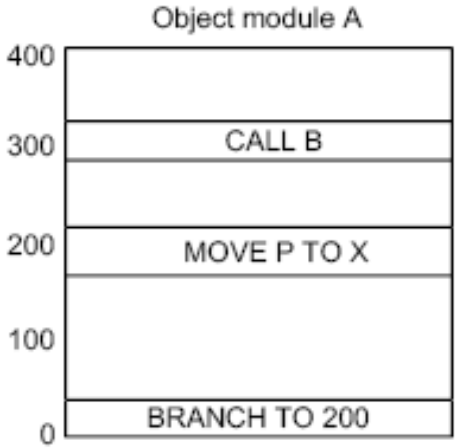
Por que criar módulos separadamente?

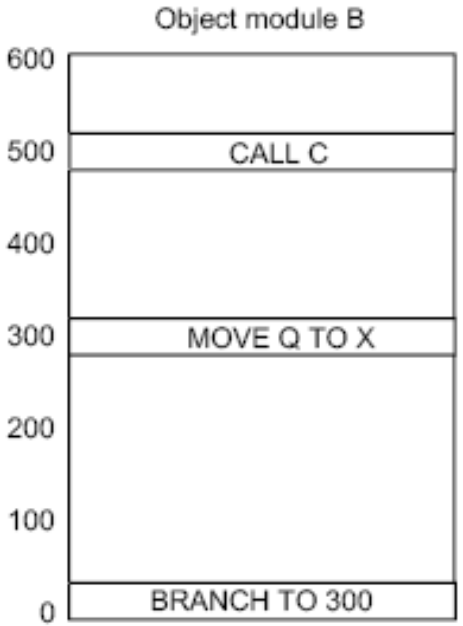
Ligadores (4)

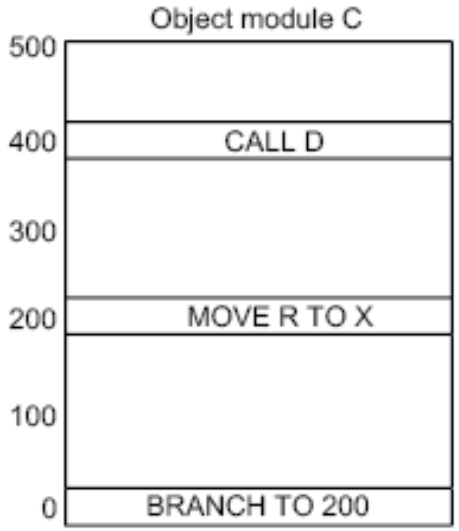
- O motivo de serem criados esses módulos separadamente é que se apenas um comando-fonte for alterado, não é necessário traduzir (montar) novamente todos os procedimentos-fonte
  - Processo de ligação é muito mais rápido
- Um *linker* realiza, então, quatro tarefas básicas:
  - Determina as posições de memória para os trechos de código de cada módulo que compõe o programa sendo "linkado";
  - Resolve as referências entre os arquivos;
  - Procura nas bibliotecas (*libraries*), indicadas pelo programador, as rotinas usadas nos fontes de cada modulo;
  - Indica ao programador quais são os *labels* que não foram resolvidos (não tem correspondente em nenhum módulo ou *library* indicados).

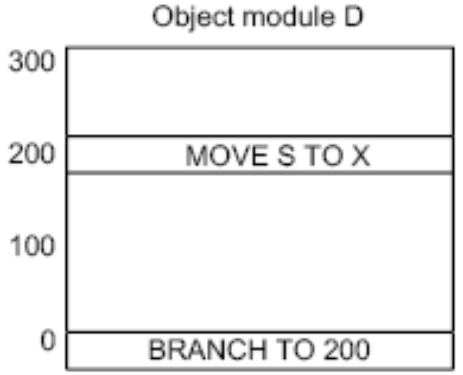
Ligadores (5)

- Problema da relocação
  - Chamadas para endereços de memória relativas ao segmento
- Problema da referência externa
  - Procedimentos externos são invocados



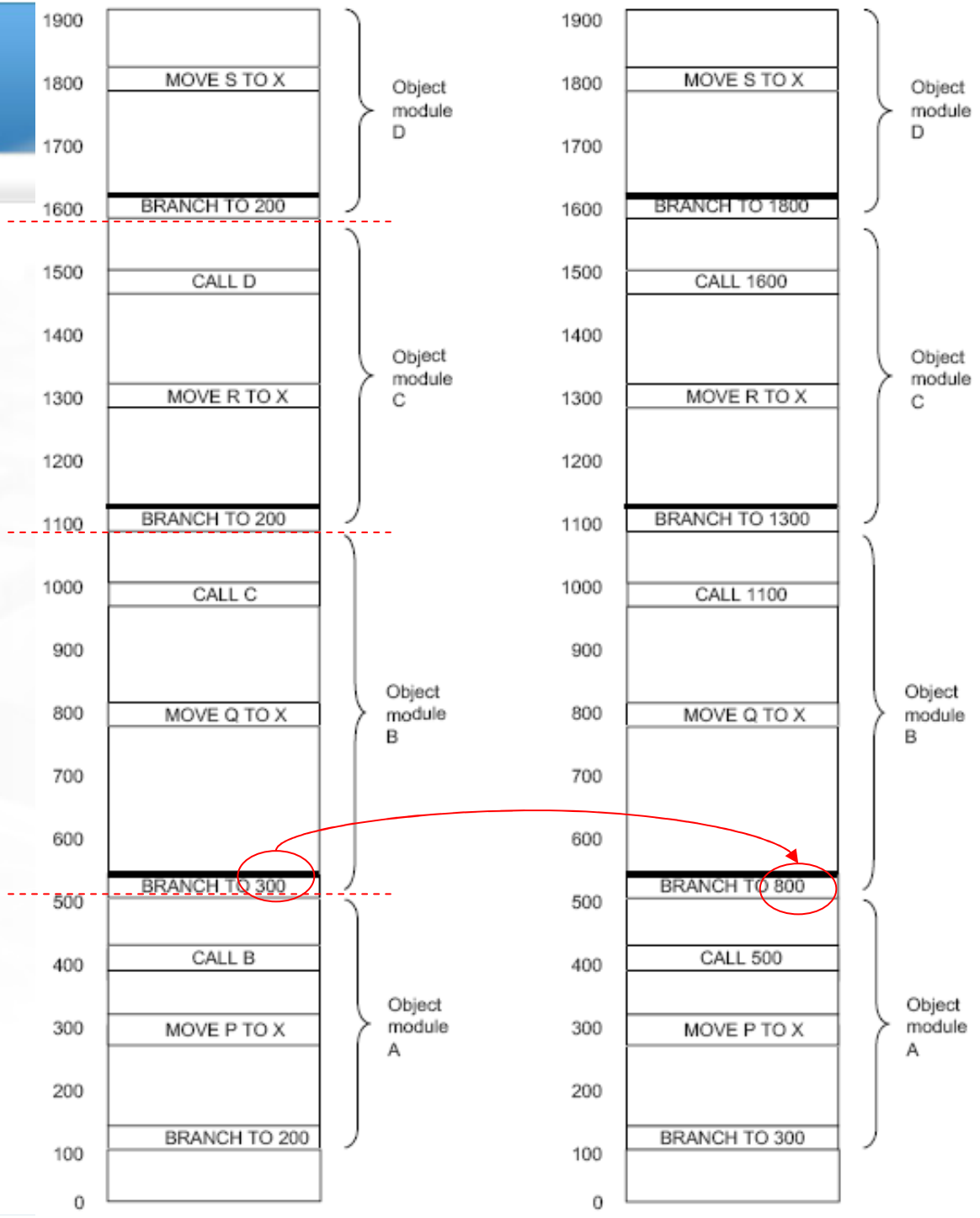






Ligadores (6)

- À esquerda: Os módulos-objetos após terem sido posicionados
- À direita: após a ligação e relocação dos endereços
  - Programa executável pronto para rodar



Ligadores (7)

- Em uma arquitetura Windows, os segmentos compilados recebem a extensão ".obj", e os executáveis a extensão ".exe"
- No UNIX, os segmentos recebem a extensão ".o", e os executáveis não possuem extensão nenhuma
- **Ligação dinâmica**
  - A ligação dinâmica é um recurso que visa minimizar o uso de recursos do sistema operacional, como memória RAM
  - Apesar das arquiteturas computacionais apresentarem abordagens diferentes em relação a este recurso, todas baseiam-se no mesmo conceito: alocar o procedimento apenas quando ele for necessário
  - Dessa forma, o processo ligação acontece assim que ocorrer a chamada ao procedimento

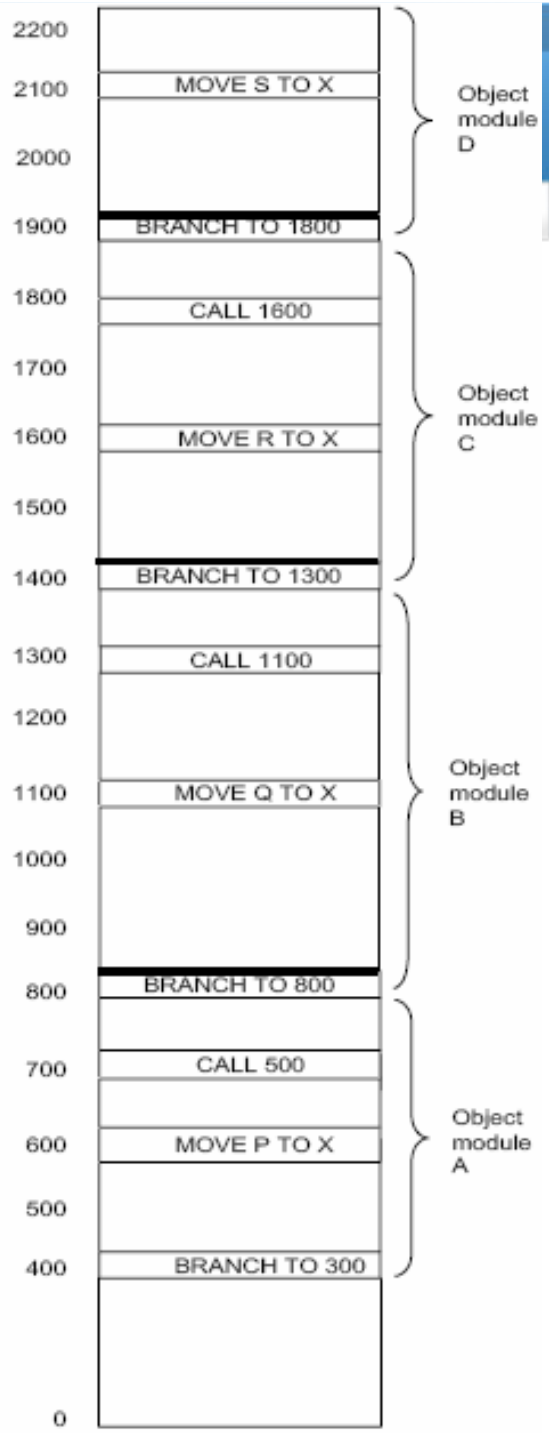
Carregadores (1)

- Programas que tenham sido *linkados* sem erros podem ser executados.
- **Carregador (*Loader*)**
  - Utilizado para executar um programa.
  - Em geral, um programa carregador faz parte do sistema operacional.

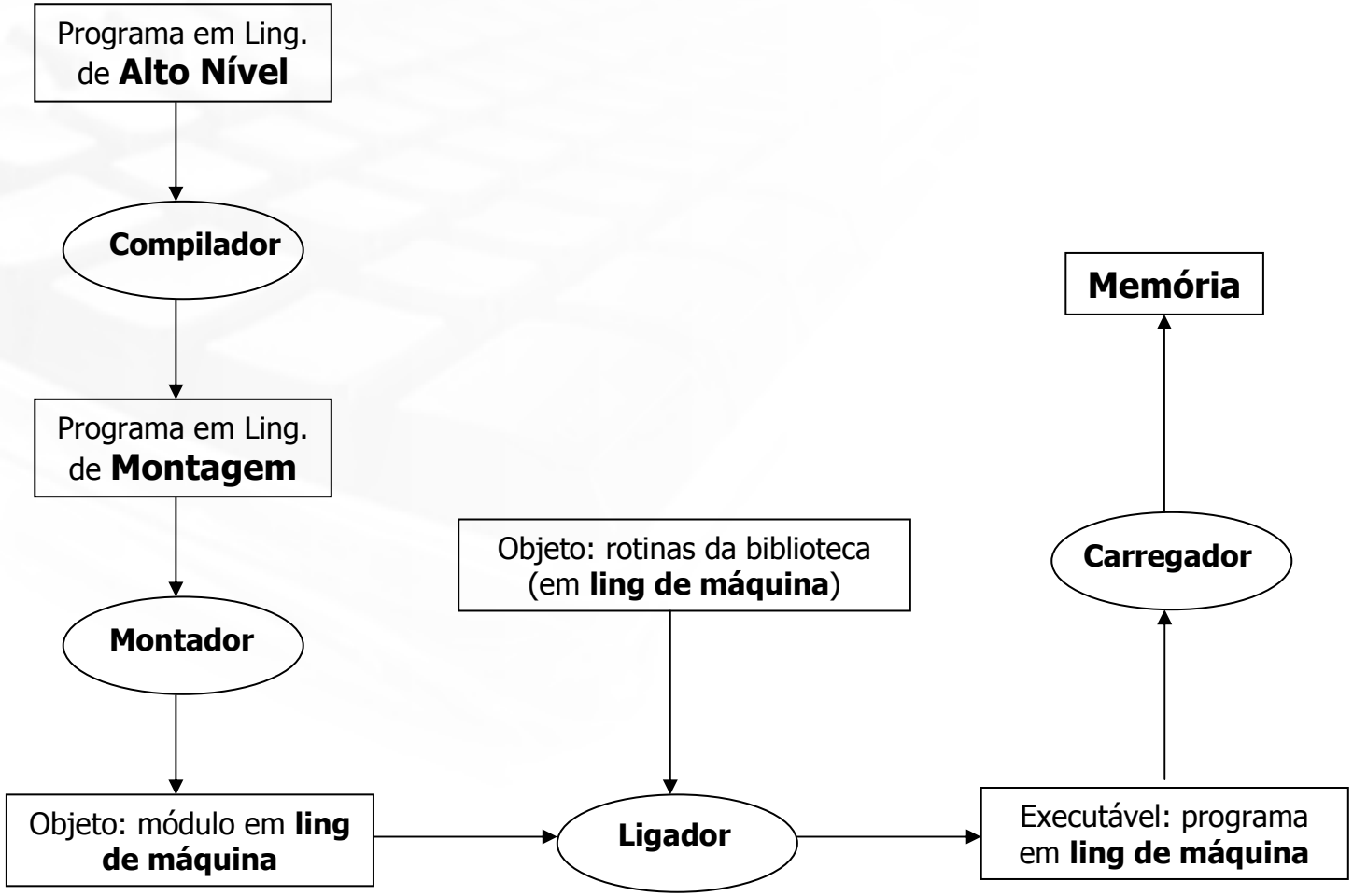
- Para iniciar a execução do programa, o *loader*:
  - Lê o *header* do executável para determinar o tamanho das suas diversas partes;
  - Separa um trecho da memória para receber os segmentos *text*, data, e também para acomodar o *stack*;
  - Copia as instruções e os dados para o trecho de memória separado;
  - Copia os argumentos passados ao programa para a área de memória separada para stack;
  - Inicializa os registradores do processador para valores apropriados (em particular, o que guarda o endereço do topo da *stack*);
  - Salta para a primeira instrução do programa usando a instrução da máquina utilizada para chamada de procedimento.

Carregadores (2)

- O que aconteceria se o programa anterior, já relocado, fosse carregado a partir do endereço 400?
  - Todos os endereços de memória referenciados no programa estariam incorretos...



Execução de um Programa



Compiladores e Interpretadores

- **Compiladores**
  - São programas que recebem como entrada arquivos texto contendo módulos escritos em linguagem de alto nível e geram como saída arquivos objeto correspondentes a cada módulo.
  - Se todas as bibliotecas e módulos são apresentados como entrada, geram um programa executável diretamente.

- **Interpretadores**
  - Recebem como entrada arquivos texto contendo programas em linguagem *assembly* ou linguagem de alto nível, ou arquivos binários com instruções de máquina, e os executam diretamente.
  - Interpretadores percorrem os programas, a partir de seu ponto de entrada, executando cada comando.

Referências

- Andrew S. Tanenbaum, ***Organização Estruturada de Computadores***, 5ª edição, Prentice-Hall do Brasil, 2007.