

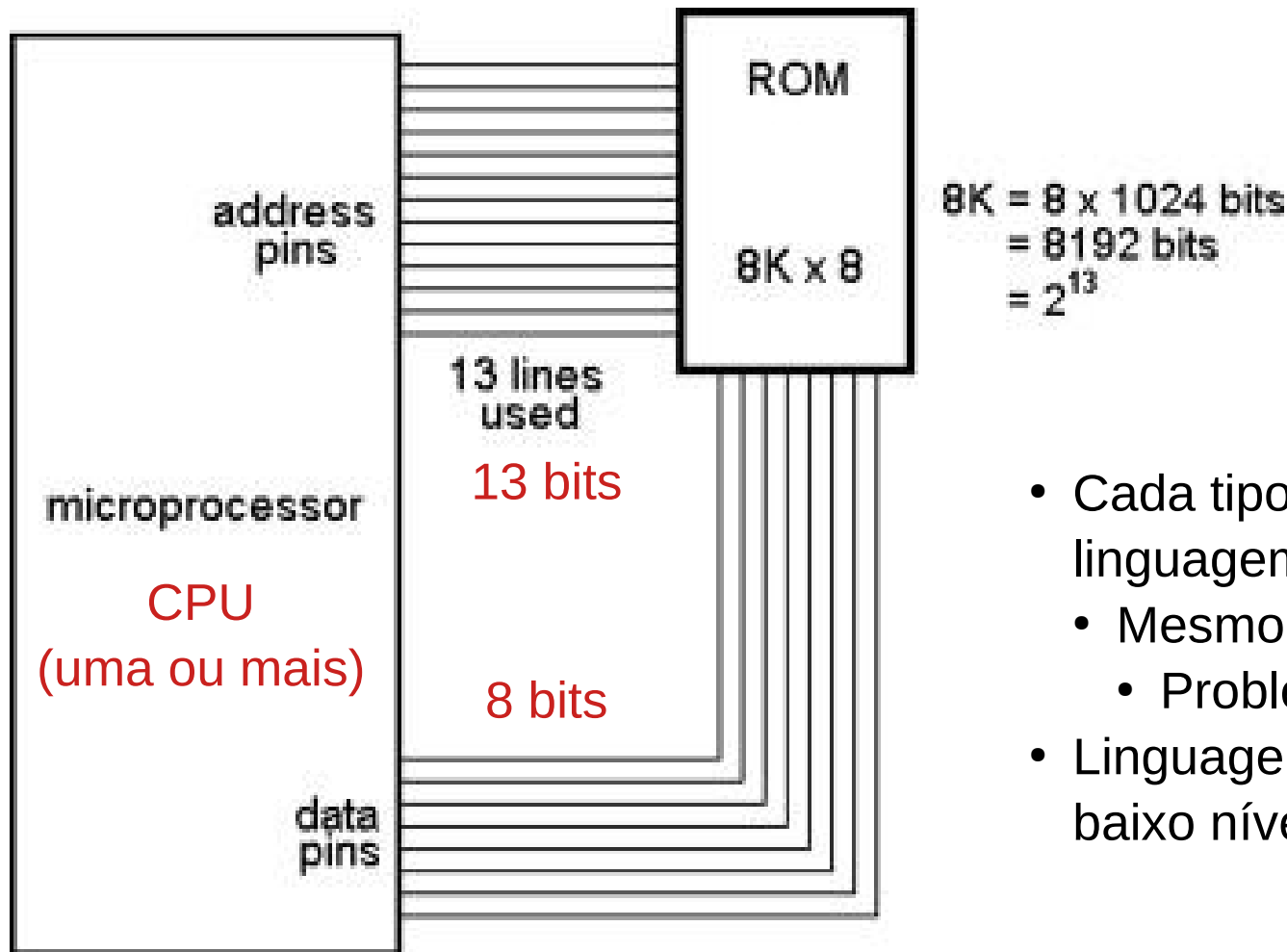
Linguagens de Máquina e de Montagem

Eduardo Furlan Miranda

2024-03-03

Linguagem de máquina (ou código de máquina)

- Lido e executado diretamente pela CPU
- Composto de números binários: 01001000 01100101 01101100



- Cada tipo de CPU tem a sua linguagem específica
 - Mesmo x86 tem suas variações
 - Problema de compatibilidade
- Linguagem de programação de baixo nível

Linguagem de montagem (*Assembly*)

endereço

código de máquina

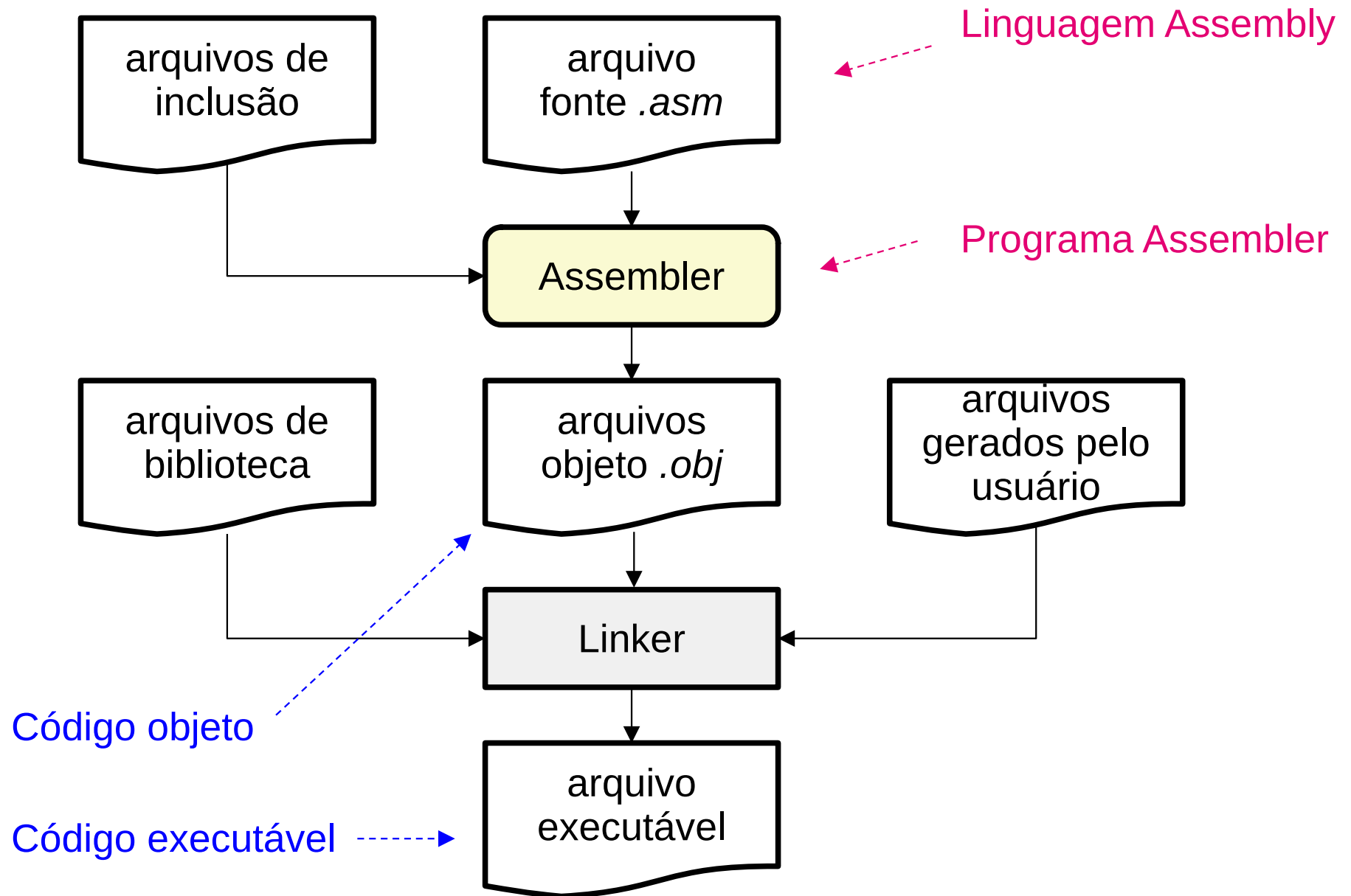
Assembly

1			GLOBAL _start
2			
3			; Code goes in the text section
4			SECTION .text
5	00000000	B801000000	_start: mov rax,1 ; 'write' system call
6	00000005	BF01000000	mov rdi,1 ; file descriptor
7	0000000A	48BE-	mov rsi,hello ; string to write
7	0000000C	[0000000000000000]	
8	00000014	BA0D000000	mov rdx,hLen ; length of string
9	00000019	0F05	syscall ; call the kernel
10			
11			; Terminate program
12	0000001B	B83C000000	mov rax,60 ; 'exit' system call
13	00000020	BF00000000	mov rdi,0 ; exit with error code
14	00000025	0F05	syscall ; call the kernel
15			
16			; Define variables in the data section
17			SECTION .rodata
18	00000000	48656C6C6F20776F72-	hello: db "Hello world!",10
18	00000009	6C64210A	
19			hLen: equ \$-hello

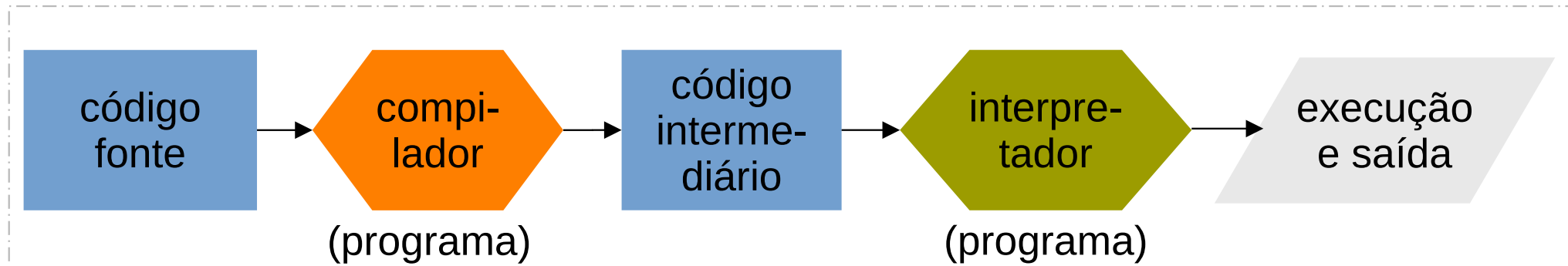
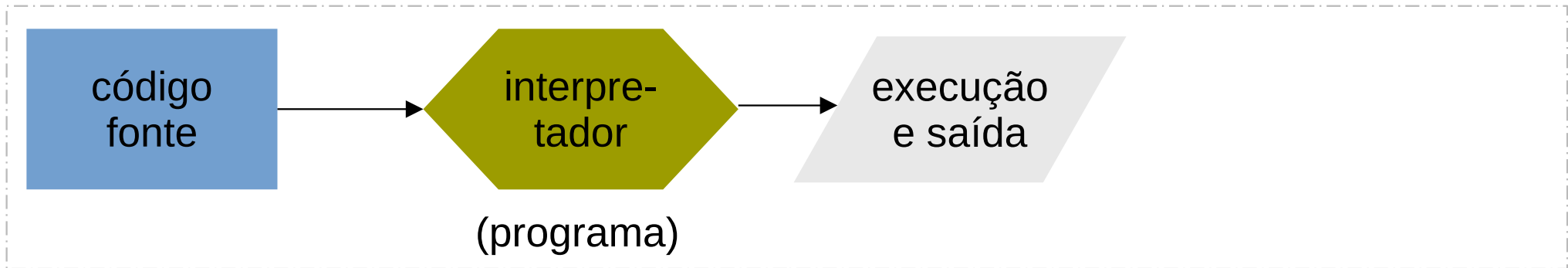
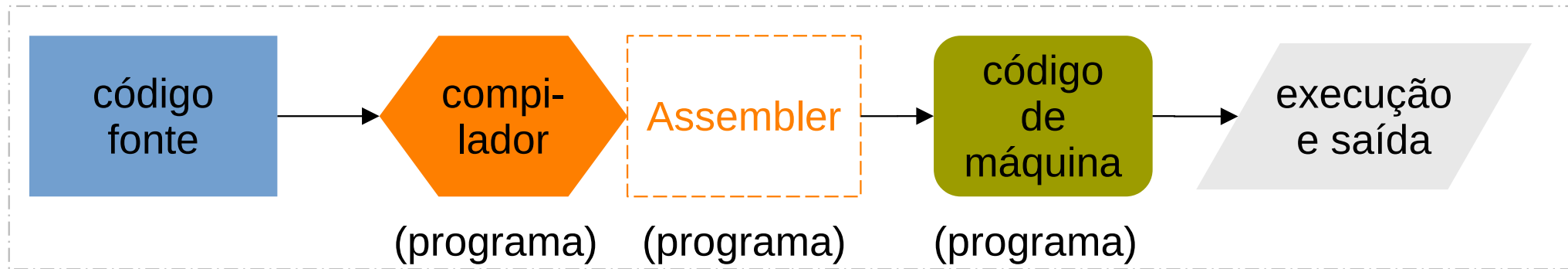
O programa montador (*Assembler*)

- A CPU executa diretamente o código montado
- O programa montador é chamado Assembler (não confundir)
 - Converte/substitui mnemônicos em seus equivalentes numéricos
- Recursos como Diretivas: pseudo-instruções ou pseudo-opcode
 - Macros, variáveis, strings, comentários, endereços simbólicos
- Montadores: GAS, NASM, TASM, WASM, YASM, MASM, ...
 - O Assembly acaba se misturando aos recursos do programa Assembler
 - Um programa feito para um montador pode não rodar no outro
- Compiladores tendem a usar montadores como uma das etapas
 - Modulariza, facilita desenvolvimento e manutenção

Geração de executável



Interpretador, compilador, montador



Compilador

The screenshot displays the Godbolt Compiler Explorer web application. The browser's address bar shows the URL `https://godbolt.org`. The page header includes the "COMPILER EXPLORER" logo, navigation links like "Add...", "More", and "Templates", and a "Share" button. The main interface is divided into two panels. The left panel, titled "C++ source #1", contains a C++ code editor with the following code:

```
1 // Type your code here, or
2 int square(int num) {
3     return num * num;
4 }
```

The right panel, titled "x86-64 gcc 13.2 (Editor #1)", shows the compiled assembly code for the same function:

```
1 square(int):
2     push    rbp
3     mov     rbp, rsp
4     mov     DWORD PTR [rbp-4], edi
5     mov     eax, DWORD PTR [rbp-4]
6     imul    eax, eax
7     pop     rbp
8     ret
```

At the bottom of the right panel, there is an "Output" section showing the compilation time and size: "x86-64 gcc 13.2 - 467ms (2811B) ~170 lines filtered".

Execução (*run-time*)

- Durante a execução do programa, 3 níveis estão em evidência:
 - Nível de microarquitetura (*hardware*)
 - Arquitetura do conjunto de instruções (ISA)
 - Sistema operacional
- Para compiladores que traduzem para código de máquina
 - O executável gerado não precisa do código-fonte, compilador, etc.
- No caso dos interpretadores, dependendo da implementação, durante a execução do programa final é necessário também:
 - A parte de *run-time* do interpretador
 - Código intermediário

Assembly

- Mais fácil de programar do que usando diretamente código de máquina em binário ou hexadecimal
- Cada declaração corresponde a uma instrução de máquina
- A portabilidade é um problema, entre arquiteturas e *assemblers*
- O recurso de nomes e endereços simbólicos faz uma diferença enorme na programação
 - Usado junto com diretivas do montador permite um pequeno nível de abstração do hardware
 - Muito mais fácil de lembrar: ADD, SUB, MUL, DIV
 - Endereços usando nomes ao invés do endereço numérico em si

Assembly

- Permite acesso a 100% do hardware e recursos da arquitetura
- Acesso a instruções que linguagens de alto nível não permitem
- Em linguagens de alto nível para acessar recursos específicos do hardware pode ser necessário o uso de Assembly
- O Assembly é muito útil para debugging e para otimização de código, ao analisar o que um compilador está produzindo
 - Quais instruções o compilador está gerando, e como
 - Para isso é necessário conhecer Assembly e os recursos do processador
- Escrever um programa é demorado e improdutivo
- Difícil de depurar ou manter

Assembly

- Aplicações: *drivers* de dispositivos, rotinas da BIOS, aplicações de tempo real, software embarcado, microcontroladores
- Como é mais produtivo utilizar linguagem de alto nível, geralmente tenta-se primeiro escrever, p. ex. um jogo, em alto nível e as partes que precisam ser otimizadas vão sendo reescritas
 - Pode-se melhorar o algoritmo ou utilizar melhor recursos de hardware
 - É necessário conhecer quais são os recursos de hardware disponíveis
 - Implica conhecer Assembly pois está diretamente relacionado
 - Analisar o Assembly gerado pelo compilador é uma das técnicas
 - Utilização de instruções vetoriais do processador
 - Analisar o Assembly gerado para laços de repetição
 - Depende do compilador, flags de compilação, recursos, técnicas

Linguagem de máquina e Assembly

- Todas as linguagens (Python, C/C++, Java, etc.) usam linguagem de máquina, pois é o que o processador entende e executa
- Um compilador pode traduzir de alto nível para Assembly
- Partes de um interpretador podem ser escritas em Assembly
- Ao compilar em diferentes arquiteturas, é desejável conhecer os recursos do processador e como otimizar para desempenho
 - Significa também analisar o Assembly que o compilador gera
 - Avaliar quais recursos do processador estão sendo usados
 - De que forma está sendo gerado o código objeto
 - Overhead

Formato de um programa em linguagem Assembly

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	DD	3	; reserve 4 bytes initialized to 3
J	DD	4	; reserve 4 bytes initialized to 4
N	DD	0	; reserve 4 bytes initialized to 0

Figure 7-1. Computation of $N = I + J$ on the x86.

- Apesar de existirem variações, geralmente as colunas estão presentes nesta ordem

- Rótulos permitem nomes simbólicos para
 - Endereços de memória
 - Dados
 - Strings
- Alguns montadores requerem o uso de dois pontos junto ao rótulo
 - Para permitir que apenas o rótulo exista em uma linha
- Outros montadores possuem outras sintaxes e limitações
- O GAS (GNU Assembler) que faz parte do GCC é mais voltado para o uso com o compilador
 - Não possui muitas facilidades voltadas para o programador

- A sintaxe da linguagem Assembly pode ter variações nos diferentes montadores. Ex.: MOV e MOVE
- Comentários e documentação são muito importantes no Assembly, já que a linguagem em si é relativamente difícil de ler e entender
 - Outros programadores dependem da documentação
 - Quem programou, depois de um certo tempo, também depende

Pseudo-instruções ou diretivas de montagem

- Fazem parte do montador (*Assembler*)
 - São usadas apenas durante a montagem
 - Variam conforme o montador usado
 - Podem controlar como o código objeto é gerado

Exemplo

Pseudoinstruction	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
DW	Allocate storage for one or more (initialized) 16-bit (word) data items
DD	Allocate storage for one or more (initialized) 32-bit (double) data items
DQ	Allocate storage for one or more (initialized) 64-bit (quad) data items
PROC	Start a procedure

Algumas diretivas

- **SEGMENT** - ex.: `SEGMENT .TEXT`
 - Segmentos são características dos processadores x86
 - A diretiva instrui o montador a iniciar um novo segmento
 - O código objeto criado inclui a informação de segmento
- **EQU** - ex.: `BASE EQU 1000`
 - Atribui 1000 ao símbolo BASE
 - Pode ser usado em expressões, ex.: `LIMIT EQU 4 * BASE + 2000`
- **DB** - ex.: `TABLE DB 11, 23, 49`
 - Aloca espaço e armazena a variável

Macro

- Faz a substituição de texto
- Útil para não ter que repetir trechos
- É descartado pelo Assembler assim que termina a montagem
 - Olhando apenas o código objeto não é possível dizer se foi usado macro
- Possui parâmetros como se fosse uma função, mas na verdade é apenas substituição (expansão) de texto
- Podem ter escopo local e global

```
CHANGE  MACRO P1, P2
        MOV EAX, P1
        MOV EBX, P2
        MOV P2, EAX
        MOV P1, EBX
ENDM
```

```
CHANGE P, Q
CHANGE R, S
```

O processo de montagem

- Montadores de duas passagens (*two-pass assemblers*)
- Algumas referências só podem ser resolvidas se olhar mais à frente no programa
 - Gera a necessidade de avançar na análise (*parsing*) até encontrá-las
 - Depois precisa voltar até o ponto onde tinha parado
- Para evitar o vai-e-volta em um primeiro passo o programa é percorrido e as referências são armazenadas em uma tabela
- Em um segundo passo (segunda leitura do programa) os símbolos são resolvidos usando a tabela

Passo um

- A principal função é construir tabelas, incluindo a de símbolos
 - Um símbolo pode ser um rótulo ou um valor de uma pseudo-instrução
 - Ex.: BUFSIZE EQU 8192
- ILC é um apontador do endereço da instrução sendo montada
 - O valor é zero no início
 - Vai sendo incrementado ao longo da montagem
 - A quantidade de bytes de cada instrução varia
- A maioria dos montadores usa pelo menos 3 tabelas
 - Símbolos
 - Pseudo-instrução
 - Opcode


Exemplo

Tabela de símbolos

Symbol	Value	Other information
MARIA	100	(tamanho dos campos de dados, informação sobre realocação, escopo, etc.)
ROBERTA	111	
MARILYN	125	
STEPHANY	129	

Tabela de opcode

Indica as semelhanças, faz um agrupamento (exemplo)



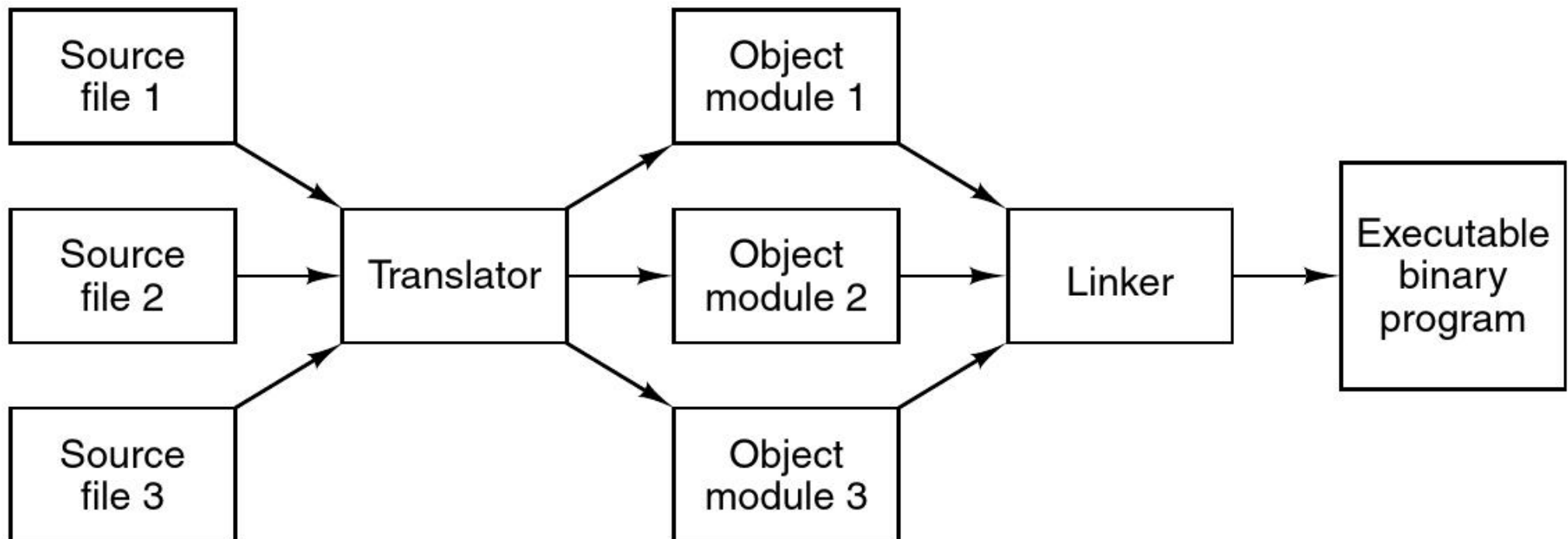
Opcode	First operand	Second operand	Hex opcode	Instruction length	Instruction class
AAA	—	—	37	1	6
ADD	EAX	immed32	05	5	4
ADD	reg	reg	01	2	19
AND	EAX	immed32	25	5	4
AND	reg	reg	21	2	19

Passo dois

- Lê o programa novamente, completando as informações faltantes com o que está nas tabelas criadas no primeiro passo
- O principal objetivo é gerar o programa objeto
 - Como opção pode imprimir uma listagem
 - Inclui no programa objeto informações para o Linker
- O tratamento dos erros é uma parte importante do montador
- Implementação da tabela
 - *Arrays* contendo pares (símbolo, valor)
 - Busca binária para encontrar o símbolo
 - Pode ser usado *hashing* na construção da tabela

Linker

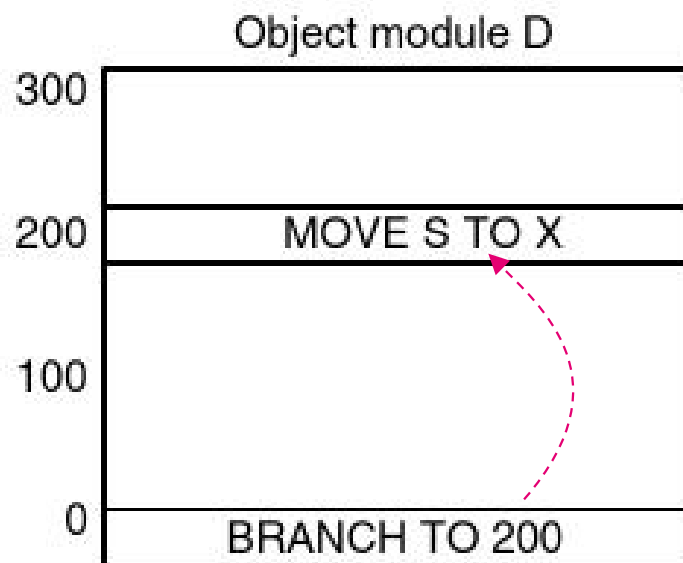
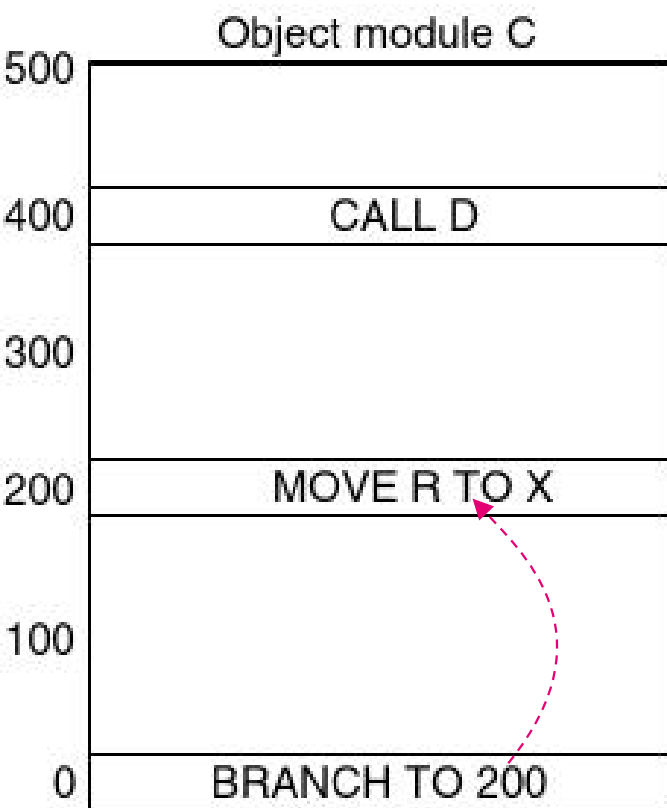
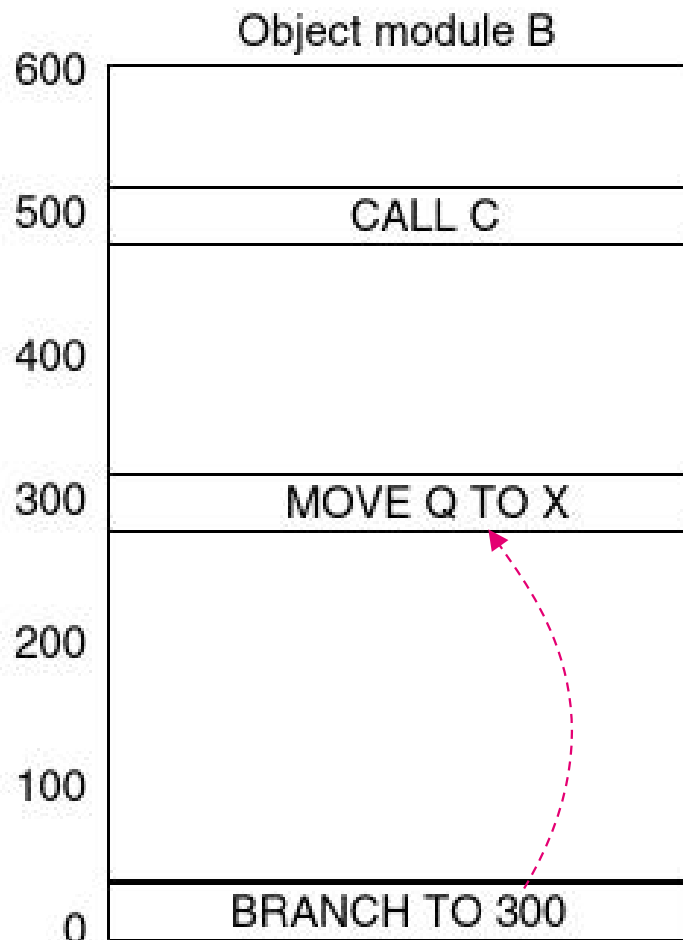
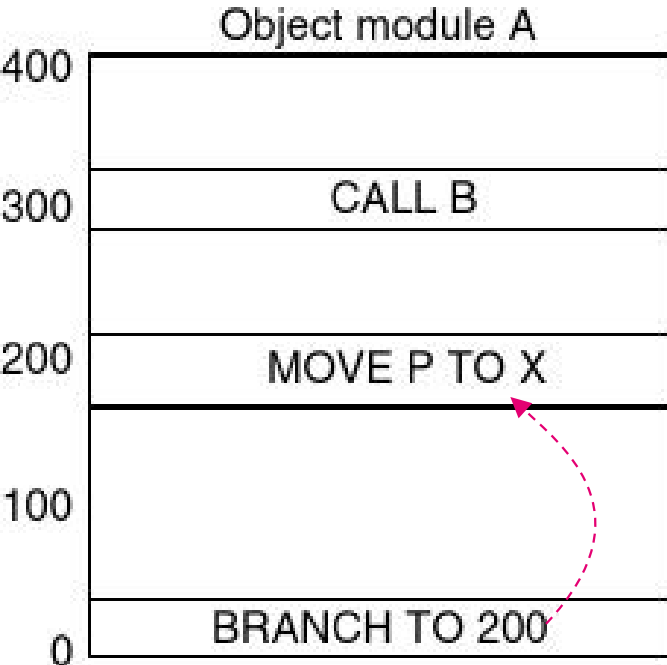
- Compiladores e montadores criam código objeto que podem depender de bibliotecas ou outras rotinas externas
- Quem une tudo e cria o código executável, é o Linker



Linker

- Compilar separadamente permite, p. ex.
 - Distribuir o trabalho entre programadores
 - Realizar alterações em uma parte
 - Usar bibliotecas
- Bibliotecas são uma parte importante
 - Podem ser externas, obtidas em outro projeto
 - Podem ser criadas como parte do programa
 - Permite reaproveitamento de código
- O Linker é “rápido” e o compilador é “Lento”
 - Utilizar o Linker para dividir o trabalho e permitir compilar somente as partes necessárias economiza tempo de desenvolvimento

Tarefas executadas pelo Linker



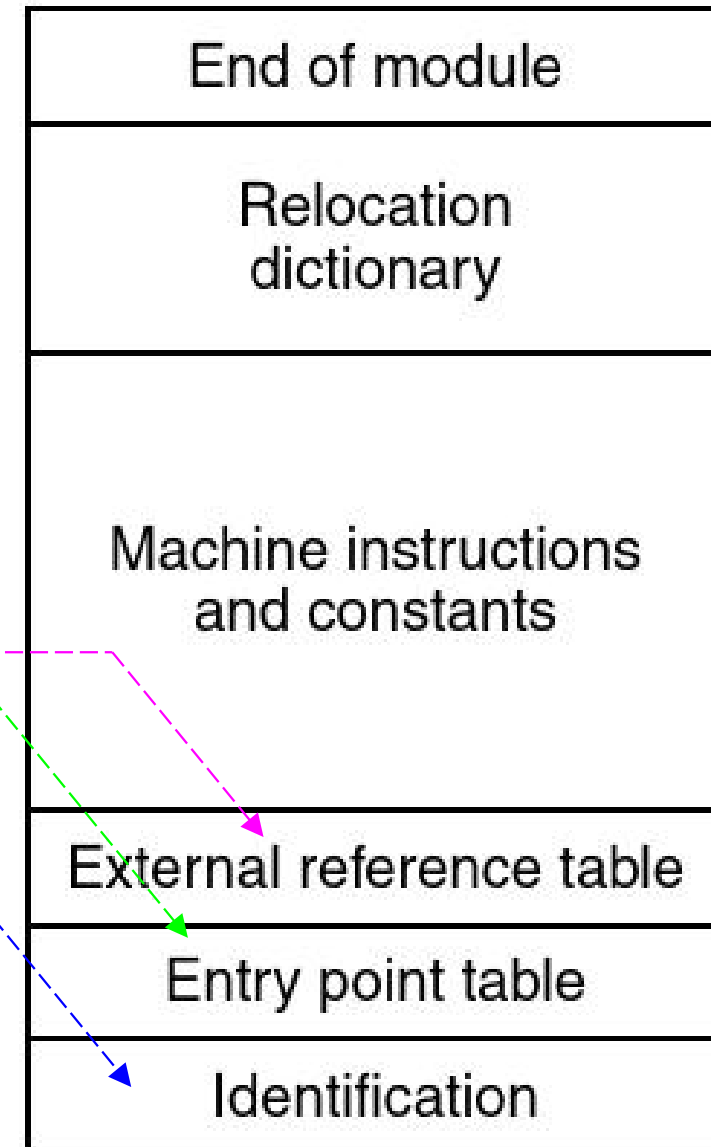
Problema: os endereços contidos nas instruções de máquina são relativos e precisam ser resolvidos

Linker

- O Linker vai carregando os módulos objeto na memória até formar o programa completo
- No programa final o espaço de endereços é linear e único
- Os endereços são resolvidos para refletir as novas posições
- Passos
 - Tabela de módulos objeto incluindo seus comprimentos
 - Baseado na tabela, atribui um endereço base a cada módulo objeto
 - Atualiza instruções que referenciam a memória
 - Atualiza instruções que referenciam outros módulos ou procedimentos

Estrutura de um módulo objeto

- Identificação
 - Nome do módulo e data
 - Comprimentos de partes do módulo
- Tabela de ponto de entrada
 - Lista de símbolos e seus valores relativos
- Tabela de referência externa
 - Símbolos que são usados no módulo porém são definidos em outros módulos

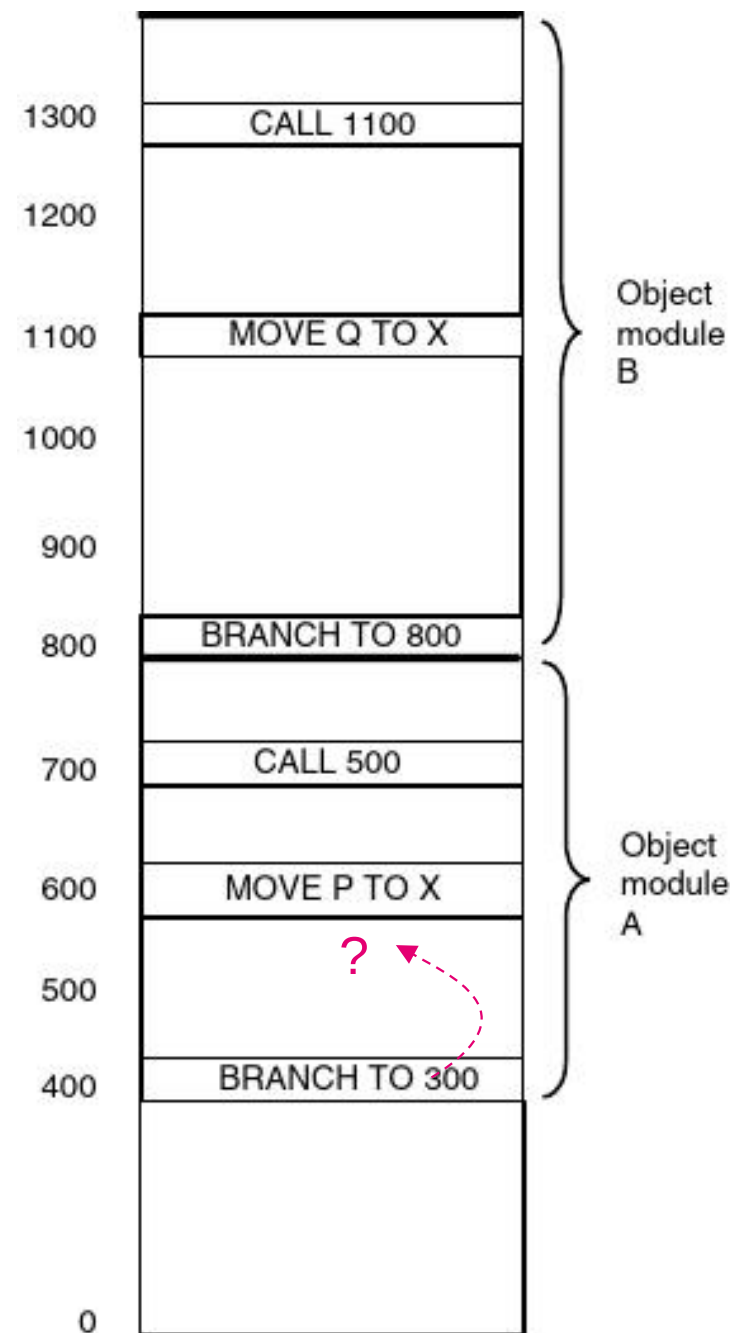


(continua)

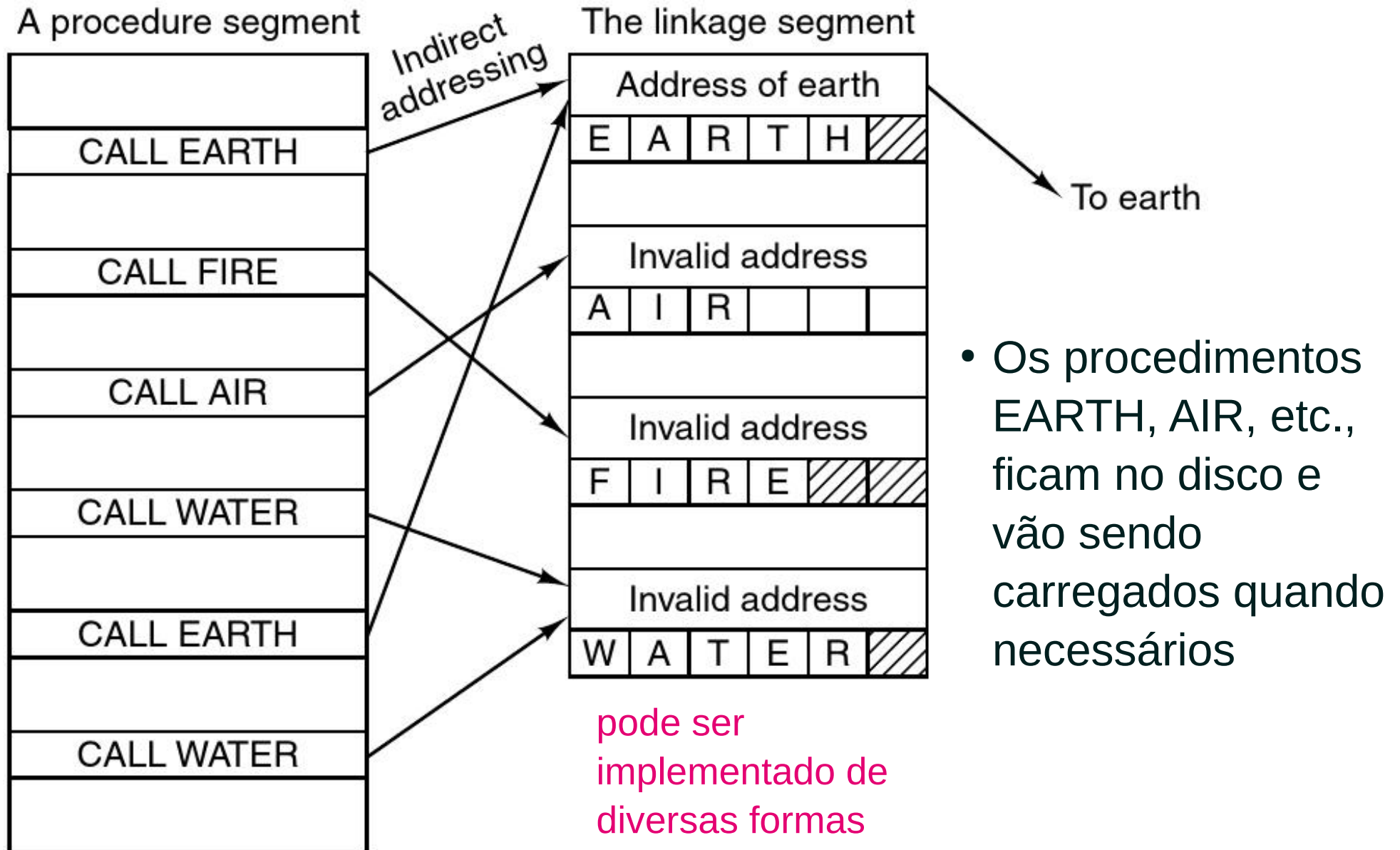
- Instruções de máquina e constantes
 - Parte do módulo com a linguagem de máquina
 - Contém endereços não resolvidos
- Diretório de realocação
 - Informações sobre quais endereços devem ser realocados
 - Quais endereços contêm instruções de máquina e quais contêm constantes
- Fim do módulo
 - Informações adicionais como somas de verificação e endereço de início da execução

Tempo de vinculação e realocação dinâmica

- Dois programas rodando na memória ao mesmo tempo não podem interferir um no outro, eles precisam de espaços de endereços diferentes
- Muitas plataformas, incluindo o x86, possuem no hardware uma Unidade Gerenciadora de Memória (MMU) para configurar espaços de endereços
- Outros sistemas de tradução de memória são: a segmentação, paginação, registrador de deslocamento, instruções relativas ao PC, realocação pelo SO, etc.



Vinculação dinâmica (*dynamic linking*)



Vinculação dinâmica

- MS Windows
 - Biblioteca de vínculo dinâmico (DLL)
 - Vinculação implícita
 - Uma biblioteca de importação é incluída no programa
 - Faz o gerenciamento, carregamento automático
 - Vinculação explícita (sem biblioteca de importação)
 - O sistema operacional carrega a DLL sob demanda durante a execução
 - O executável deve carregar e descarregar explicitamente a DLL
- Unix-like
 - Somente vinculação implícita. Biblioteca compartilhada tem 2 partes
 - Biblioteca hospedeira é incluída no programa
 - Biblioteca de destino é chamada durante a execução

Referências

TANENBAUM, A. S. Cap. 7 - **O nível de linguagem de montagem**. Organização Estruturada de Computadores. [S. l.]: Pearson, 2013.