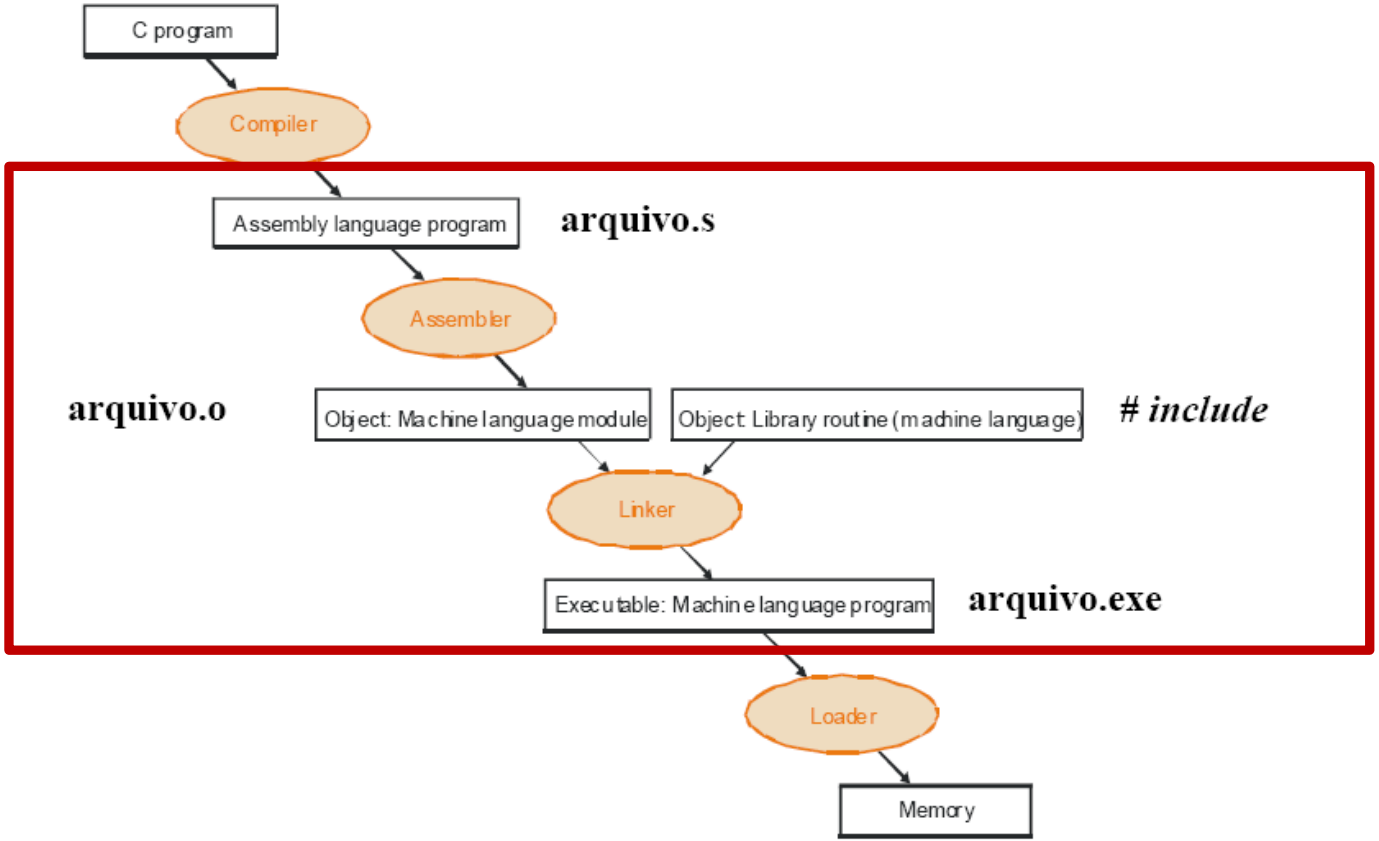
Arquitetura de Computadores

Linguagem de máquina

Prof. Tiago Gonçalves Botelho

Execução de um Programa

- Programa em linguagem de alto nível deve ser traduzido para um programa executável
- São necessários quatro passos básicos:



Compilador C

- **`gcc –S`:** Pára depois do estágio de compilação, não faz a montagem

- **`Exemplo: gcc –S arquitetura.c`**
  `Saída: arquitetura.s`

- Programa em linguagem Assembly:
  - Exemplo em linha de comando

Compilação: Montador

- gcc – c → compila ou monta os arquivos fontes mas não os agrega às bibliotecas.

- A saída é um arquivo Objeto com o mesmo nome do arquivo fonte.

- Esta compilação gera um arquivo objeto .o
- **Exemplo: gcc –c arquitetura.c**
  Arquivo de saída: arquitetura.o

Compilação: Ligador

- gcc – E → Pára depois do estágio de pré-processamento, mas não executa toda a compilação

- O arquivo de saída está na forma do código fonte pré-processado e agrega as bibliotecas.
- **Exemplo: gcc -E arquitetura.c |more**

Compilação: Ligador

- A saída do ligador une o arquivo objeto (.o) às bibliotecas e gera um arquivo executável
- A compilação de um arquivo .c gera um arquivo executável
- **Exemplos:**

1)gcc arquitetura.c -o arquitetura.e

Arquivo de saída: arquitetura.e

2) gcc -v

Saída:version 4.4.1 (TDM-2 mingw32)

Otimizando a compilação

- Existem algumas opções de otimizar a compilação:
  - -O → tenta reduzir o tamanho do código e o tempo de execução sem efetuar otimizações que aumentariam muito o tempo de compilação
  - -O1 → compilação com otimização. Toma mais tempo e memória
  - -O2 → mais otimização ainda. Efetua todas as otimizações possíveis que não envolvem uma troca espaço/velocidade. O compilador não “desenrola” loops ou “function in-line”. Essa opção aumenta o tempo de compilação assim como o desempenho do código gerado. Exemplo: gcc -o2 -o arquitetura-o2 arquitetura.c

Otimizando a compilação

- -O3 → Otimiza ainda mais. -O3 utiliza todas as otimizações especificadas por '-O' e também as otimizações "-finline-functions" e "-frename-registers"

- -O0 → Não há otimização

- -Os → Otimização de tamanho. -Os habilita todas as otimizações que não aumentam o tamanho do código. São feitas ainda otimizações para reduzir o tamanho do código.

Comparando as otimizações

Arquivo fonte (189 bytes)	**Compilação**	**Saída**	**Tamanho (bytes)**
arquitetura.c	gcc –c	arquitetura.o	688
arquitetura.c	gcc –S	arquitetura.s	681
------	gcc –v	version	---
arquiteturaopt1.c	gcc –O1 -S	arquiteturaopt1.s	551
arquiteturaopt2.c	gcc –O2 -S	arquiteturaopt2.s	647
arquiteturaopt3.c	gcc –O3 -S	arquiteturaopt3.s	685
arquiteturaopt0.c	gcc –O0 -S	arquiteturaopt0.s	685
arquiteturaopt0s.c	**Arquitetura –Os -S**	**arquiteturaopt0s.s**	**516**
arquitetura-O.c	gcc –O -S	arquitetura-O.s	549

Comparação entre programação em linguagem assembly e de alto nível

Comparação entre programação em linguagem de montagem e linguagem de alto nível, com ajuste e sem ajuste.

	Programadores-anos para produzir o programa	Tempo de execução do programa em segundos
Linguagem de montagem	50	33
Linguagem de alto nível	10	100
Abordagem mista antes do ajuste		
10% críticos	1	90
Outros 90%	9	10
	–	–
Total	10	100
Abordagem mista após o ajuste		
10% críticos	6	30
Outros 90%	9	10
	–	–
Total	15	40

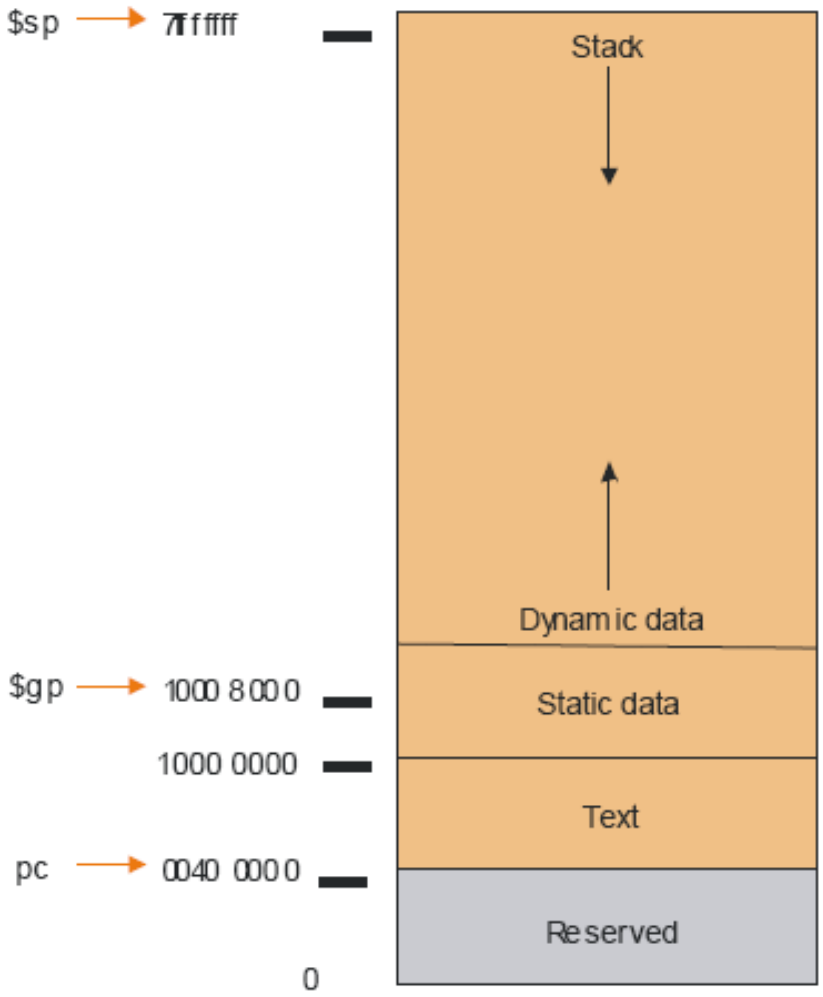
Fonte: Tanenbaum, 2007

Características do Montador

- A função do montador é transformar a linguagem de montagem em código de máquina
- O arquivo-objeto é composto de seis partes distintas:
  1. Cabeçalho: descreve tamanho e posição do restante do arquivo
  2. Segmento de texto: código em linguagem de máquina
  3. Segmento de dados: dados para a execução do programa
  4. Informações sobre realocação: identifica as palavras de instrução e de dados que dependem os endereços absolutos por ocasião da carga do programa na memória
  5. Tabela de símbolos: contém os rótulos não definidos como as referências externas
  6. Informações para análise de erros: associa instruções da máquina com os arquivos fonte em C para o depurador (debbuger)

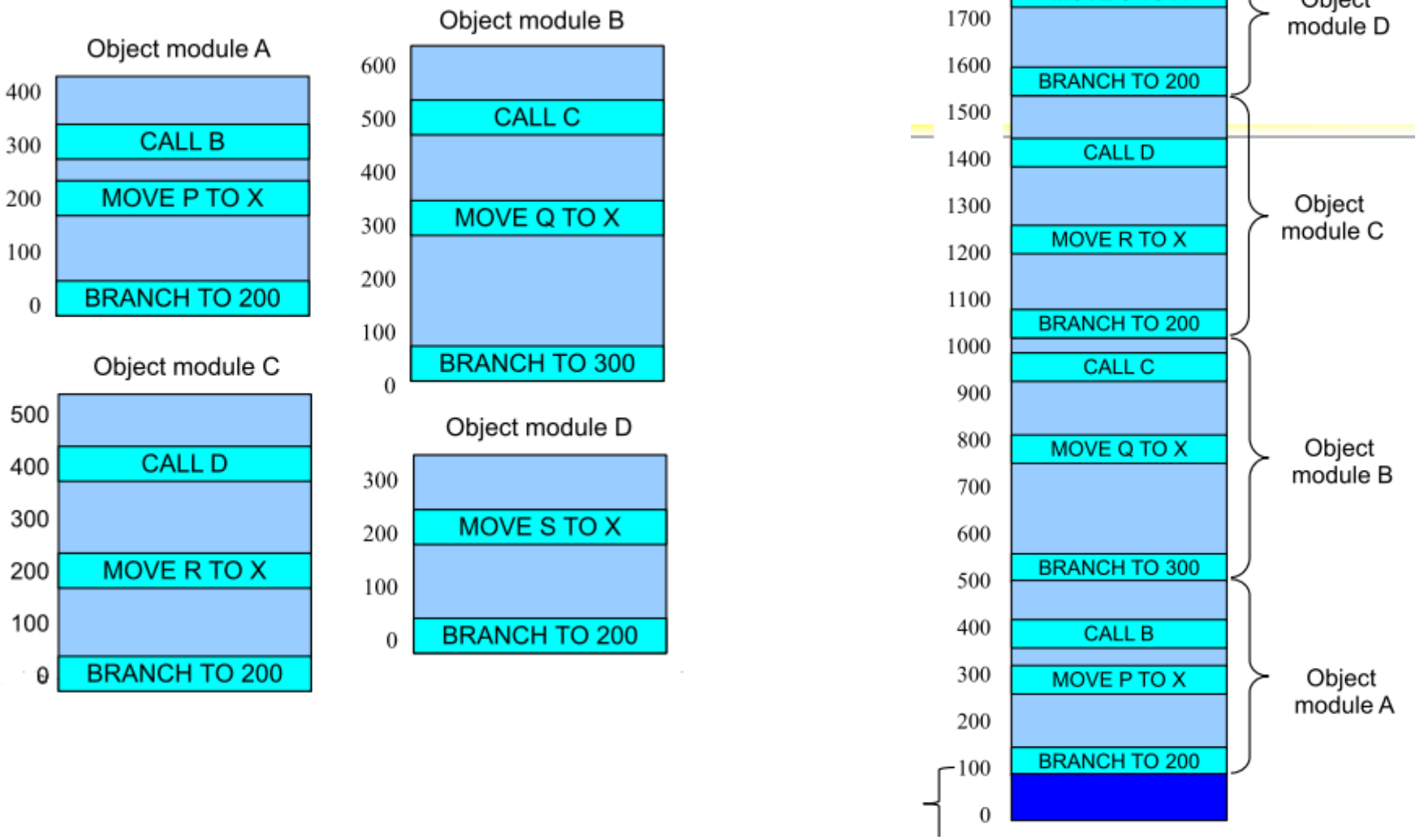
Características do Ligador

- Combina os diversos módulos com as rotinas e bibliotecas resolvendo todas as referências:
  - Coloca os módulos de código e dados simbolicamente na memória
  - Determina os endereços dos rótulos de dados e instruções
  - Resolve as referências internas e externas



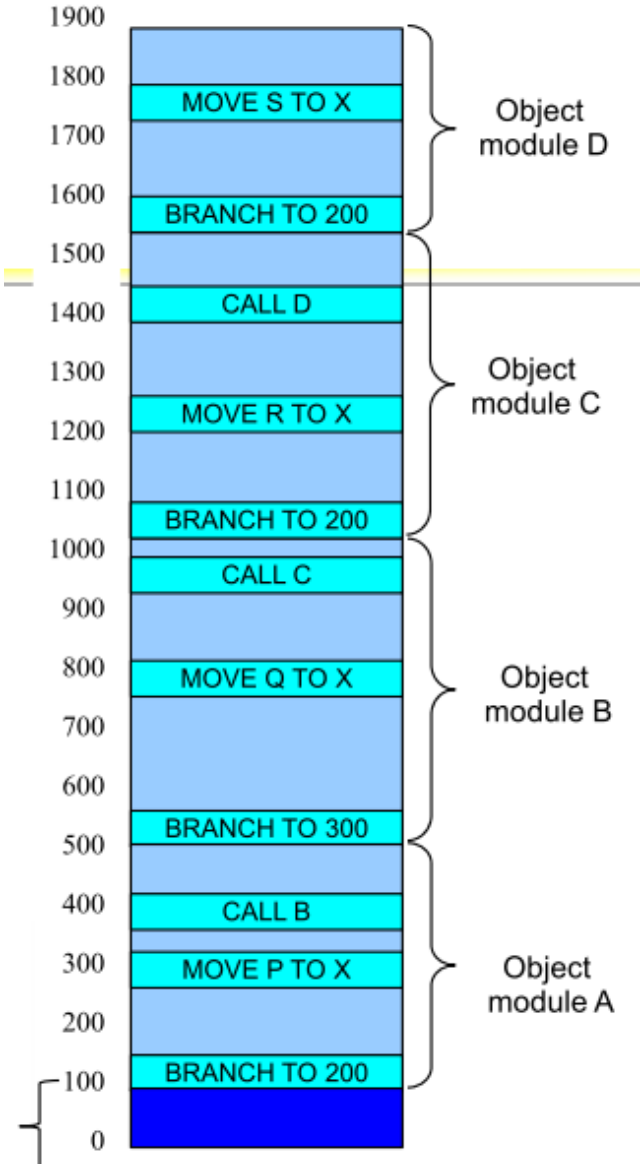
Exemplo de Ligação

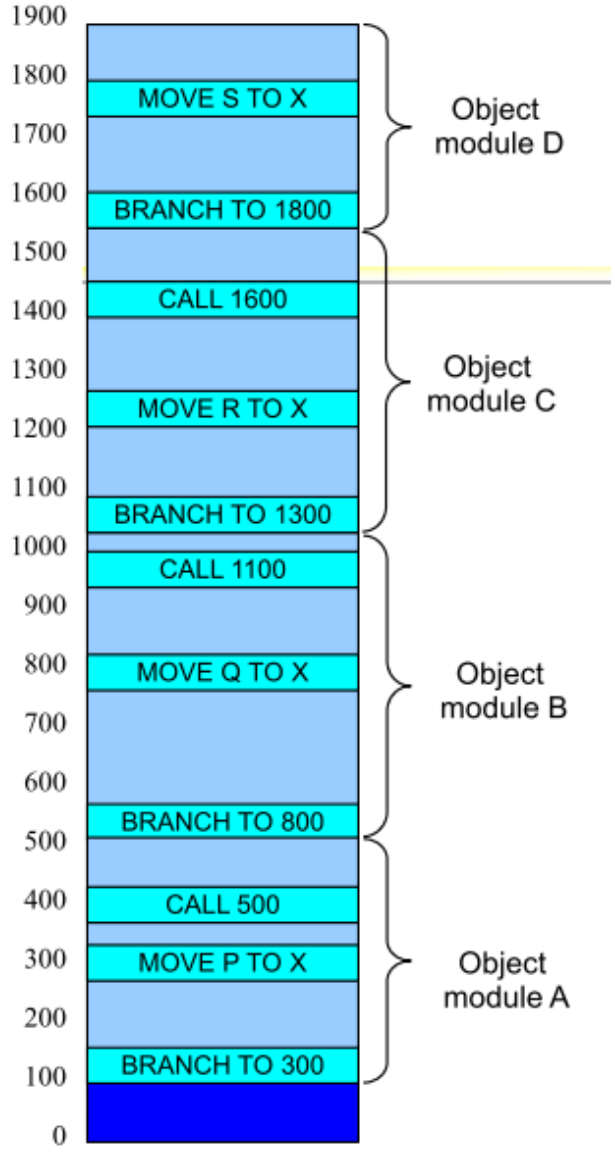
- Módulos isolados e módulos ligados:



Exemplo de Ligação

- Módulos ligados e módulos relocados:





Características do Carregador

- Com o arquivo executável em disco, o SO deve preparar a máquina para executar o programa
- Transfere o arquivo para a memória, no Unix:
  1. Leitura do cabeçalho do arquivo executável: tamanho do texto e dados
  2. Criação de espaço para armazenar o código e dados
  3. Copia os dados e as instruções para a memória
  4. Copia os parâmetros para a pilha do programa principal
  5. Inicializa os registradores da máquina e posicionar o SP=primeiro endereço livre da memória.
  6. Desvia para uma rotina de inicialização que copia os parâmetros nos registradores de argumento e chama a rotina principal do programa. Quando a rotina termina, a rotina de inicialização termina o programa executando uma chamada ao sistema do Unix → exit

Interpretação

- A execução de um programa tem fases distintas:

Compilação → Ligação → Execução

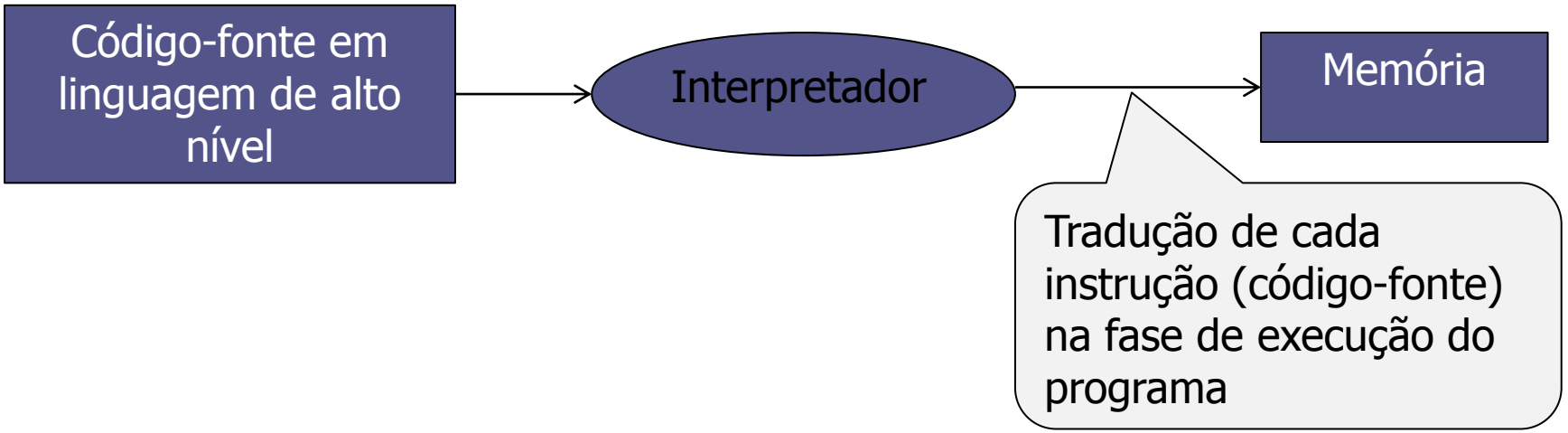
Assembler → Código Objeto → Ligador → Carregador

- Este processo não é a única forma de execução de um arquivo → Interpretação
- Interpretação também possui três fases distintas:

Compilação → Ligação → Execução, mas comando por comando em tempo de execução

Interpretação

- Não são produzidos códigos intermediários (.asm/.o)
- Cada comando é lido, verificado, convertido em código executável e imediatamente executado, antes que o comando seguinte seja sequer lido



Interpretação

- Exemplos:
  - Linguagens como HTML, BASIC, Bash, Perl, PHP, Python, Euphoria, Forth, JavaScript, Logo, Lisp, Haskell ...
  - Linguagens de programação de usuário: tais como das planilhas Excel, o Word Basic (linguagem de construção de Macros do Word), o Access, etc...

Comparação: Compilação x Interpretação

- Tempo de execução:
  - Interpretação: execução comando por comando
  - Compilação: o tempo de execução do programa é reduzido, compilação e ligação foram previamente cumpridos
- Consumo de memória:
  - O interpretador é um programa grande e permanece na memória durante todo o tempo que durar a execução
  - O compilador é carregado e fica na memória apenas durante o tempo de compilação, depois é descarregado

Comparação: Compilação x Interpretação

- Repetição de interpretação:
  - Na interpretação cada programa terá que ser interpretado toda vez que for ser executado
  - Na compilação o programa é compilado e ligado apenas uma vez, e na hora da execução é carregado apenas o módulo de carga

Comparação: Compilação x Interpretação

- Desenvolvimento e depuração:
  - Na interpretação a relação entre código fonte e executável é mais direta e o efeito da execução (certa ou errada) é direta e imediatamente sentido
  - Na compilação a identificação de erros durante a fase de execução fica sempre mais difícil, pois não há mais relação entre comandos do código fonte e instruções do executável

Emuladores

- *Um programa desenvolvido PCs rodando Windows não funciona em PCs com UNIX ou em Macintosh!!!???*
- Como uma página na Internet, com textos, imagens e programas que podem ser visualizados e processados por quase qualquer computador???

- O segredo é a utilização de linguagens padronizadas (HTML, PERL, CGI, *Java*, *Java Script*, etc.) que são suportadas por diversas plataformas → ***browsers* ou sistemas operacionais**

Emuladores

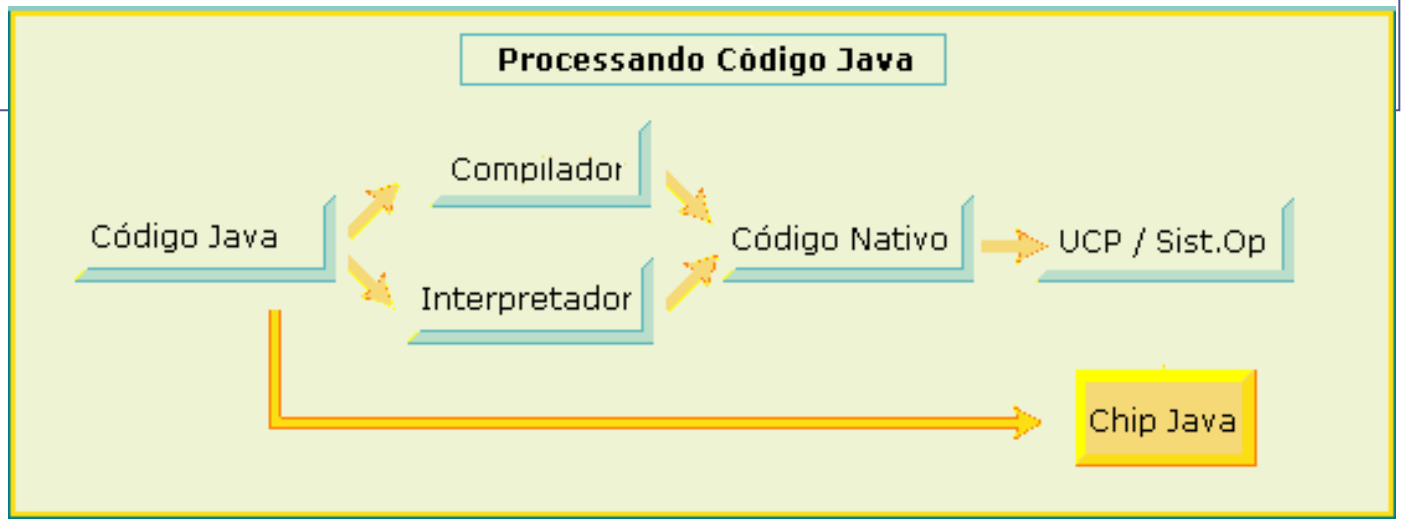
- Emuladores convertem um programa desenvolvido em uma plataforma para outra.
- Um programa cria uma camada de emulação em que uma máquina se comporta como uma outra máquina
- Exemplo:
  - Atari → PC
  - Apple → Windows → Unix
  - PC "virtual" emulado em um *Macintosh*

Máquinas Virtuais

- Uma máquina que se comporta como uma outra máquina diferente não compatível
- Camada de emulação: código executável traduzido em tempo de execução para instruções de um outro computador
- Sun → desenvolveu a linguagem Java
- Plataforma Java → *JVM - Java Virtual Machine*
- *JVM* suporta uma representação em *software* de uma UCP completa, com sua arquitetura perfeitamente definida incluindo seu próprio conjunto de instruções (ISA)

Máquinas Virtuais

- Programadores *Java* escrevem código na linguagem *Java*
- Código *Java* roda em máquinas virtuais que emulam o ambiente *Java*
- Código é compilado gerando código para a JVM
- JVM converte o código nativo e o interpreta para o ISA de cada arquitetura



Arquitetura do Conjunto de Instruções

- CISC – *Complex Instruction Set Computing*, ou computação por conjunto complexo de instruções
- RISC - *Reduced Instructions Set Computing*, ou computação por conjunto reduzido de instruções

RISC	CISC
Instruções simples e rápidas (poucos ciclos de clock)	Instruções complexas e demoradas (muitos ciclos de clock)
Instruções executadas pelo hardware	Instruções executadas em microprograma
Poucas instruções	Muitas instruções
Compiladores complexos	Compiladores simples

Referências

- Patterson, David A.; Hennessy, John; Organização e projeto de computadores: a interface hardware/software; 3ª ed.; Elsevier, 2005.

- Tanembaum, A. S. Organização Estruturada de Computadores. 5 ed – Editora Pearson, 2007.

- Prof. Luis Henrique Andrade Correia; Notas de aula.