Infraestrutura de Hardware

Instruindo um Computador – Ponteiros, Execução de Programas em C e Java, Características do Intel x86

Perguntas que Devem ser Respondidas ao Final do Curso

- Como um programa escrito em uma linguagem de alto nível é entendido e executado pelo HW?
- **Qual é a interface entre SW e HW e como o SW instrui o HW a executar o que foi planejado?**
- O que determina o desempenho de um programa e como ele pode ser melhorado?
- Que técnicas um projetista de HW pode utilizar para melhorar o desempenho?

Ponteiros

- Toda variável tem um endereço ou uma posição associados na memória
- Este endereço é visto como um ponteiro (ou apontador), uma referência para a posição de memória de uma variável
- Ponteiros fornecem um modo de acesso à variável sem referenciá-la diretamente
- Um endereço pode ser armazenado em uma variável do tipo ponteiro (ponteiro variável)

```
int* p;
```

Declara uma variável de nome `p` que pode armazenar um endereço de memória para um inteiro

Ponteiros e Arrays em C

- **Em C existe um relacionamento muito forte entre ponteiros e arrays**
  - Qualquer operação que possa ser feita com índices de um array pode ser feita com ponteiros
  - O identificador de um array representa um endereço, ou seja, um ponteiro

Ponteiros e Arrays

- Se tivermos a declaração

```
int   v [10] ;
```

- Podemos acessar elementos do array através de aritmética de ponteiros

`v + 0` ⟶ Aponta para (igual ao endereço do) primeiro elemento do array

`v + 1` ⟶ Aponta para o segundo elemento do array

⋮

`v + 9` ⟶ Aponta para o último elemento do array

- Portanto: `&v[i]↔(v + i)` `v[i]↔ *(v + i)`

Mapeando Arrays e Ponteiros no MIPS

```
int clear1(int array[], int size)
{
  int i;
  for (i = 0; i < size; i = i + 1)
    array[i] = 0;
}
```

```
int clear2(int *array, int size) {
  int *p;
  for(p= &array[0];p< &array[size];
p = p + 1)
    *p = 0;
}
```

```
   addi $t0,$zero,0 # i = 0
L1:sll $t1,$t0,2    # $t1 = i * 4
   add $t2,$a0,$t1 #$t2= &array[i]
   sw $zero,0($t2) # array[i] = 0
   addi $t0,$t0,1   # i = i + 1
   slt $t3,$t0,$a1 # $t3=(i< size)
   bne $t3,$zero,L1
```

```
   addi $t0,$a0,0 # p = & array[0]
   sll $t1,$a1,2  # $t1 = size * 4
   add $t2,$a0,$t1 #$t2=&array[size]
L2:sw $zero,0($t0) # *p = 0
   addi $t0,$t0,4   # p = p + 1 *4
   slt $t3,$t0,$t2 #$t3=
                    #(p<&array[size])
   bne $t3,$zero,L2
```

- Versão com array requer que shift seja dentro do loop
  - Para calcular endereço do elemento do índice i

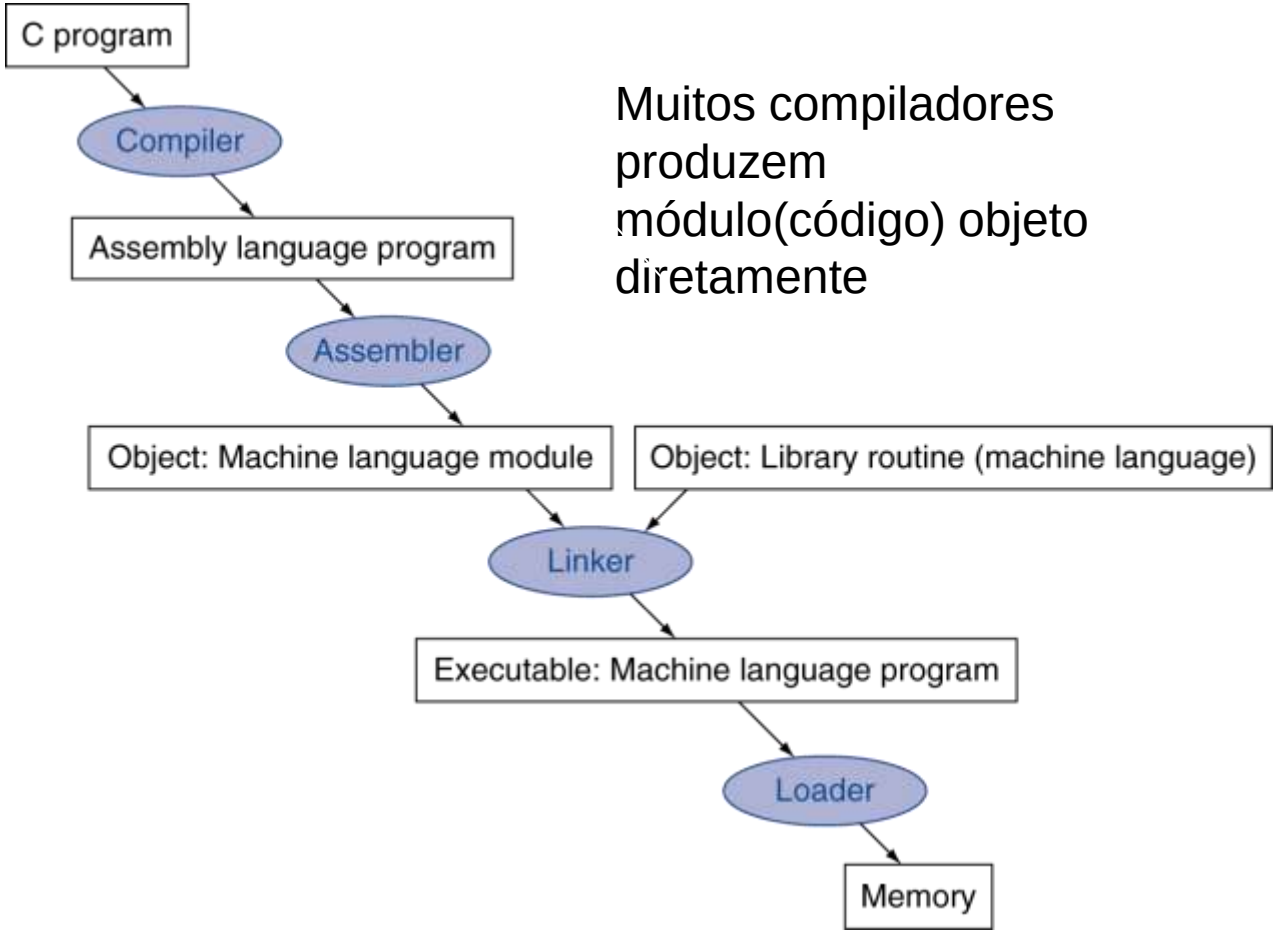
Arrays x Ponteiros

- Embora operações envolvendo arrays possam ser feitas com ponteiros, a escolha de uma forma de trabalhar ou outra pode ter impacto no desempenho

- Indexação de um array envolve:
  - Multiplicar índice pelo tamanho do tipo
  - Adicionar este valor ao endereço base do array para descobrir endereço do elemento indexado

- Ponteiros correspodem diretamente aos endereços de memória
  - Evita a complexidade extra da indexação

Executando um Programa em C



Muitos compiladores produzem módulo(código) objeto diretamente

Ligação estática

Produzindo um Módulo Objeto

- Assembler traduz o programa assembly em instruções de máquina
- Assembler transforma pseudo-instruções assembly (se houver) em instruções realmente aceitas pela arquitetura
  - Ex: instrução `move r1,r2` é aceita pelo assembler do MIPS, embora instrução não exista no MIPS
    - Transforma em `add r1, zero, r2`
- Assembler transforma um programa na linguagem assembly em um módulo objeto

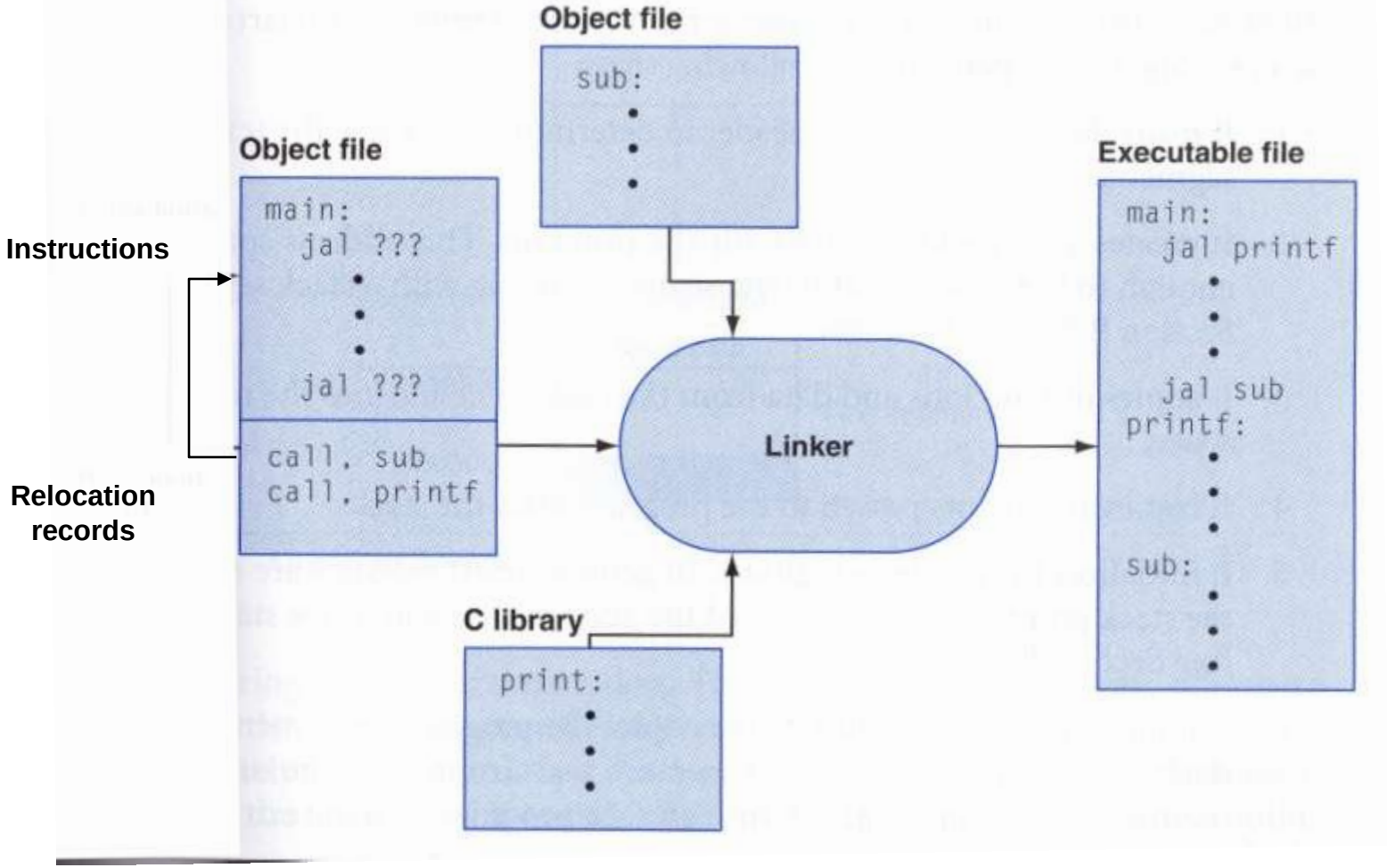
Informações de um Módulo Objeto

- Header
  - descreve conteúdo do módulo
- Segmento de texto
  - instruções traduzidas
- Segmento de dados estáticos
- Informações de Relocação
  - Identifica instruções e dados que dependem de endereços absolutos quando o programa for carregado na memória
- Tabela de Símbolos
  - Associa labels com endereços
  - Podem existir ainda labels indefinidos (não associados a nenhum endereço)
    - Ex: referências para rotinas externas

Ligando Módulos de Objetos

- Um programa geralmente é composto por diferentes módulos
  - Podem ser compilados separadamente
  - Mudança em uma rotina implica em compilar apenas um módulo do sistema
- Linker é o programa que combina módulos de objetos compilados independentemente
- Produz um executável
  - Estabelece como os diferentes módulos devem ser organizados na memória
  - Determina os endereços dos labels indefinidos
    - Através da tabela de símbolos e informações de relocação

Linker


Object file
sub:
Object file
main:
jal ???
jal ???
Instructions
call, sub
call, printf
Relocation records
Linker
Executable file
main:
jal printf
jal sub
printf:
sub:
C library
print:

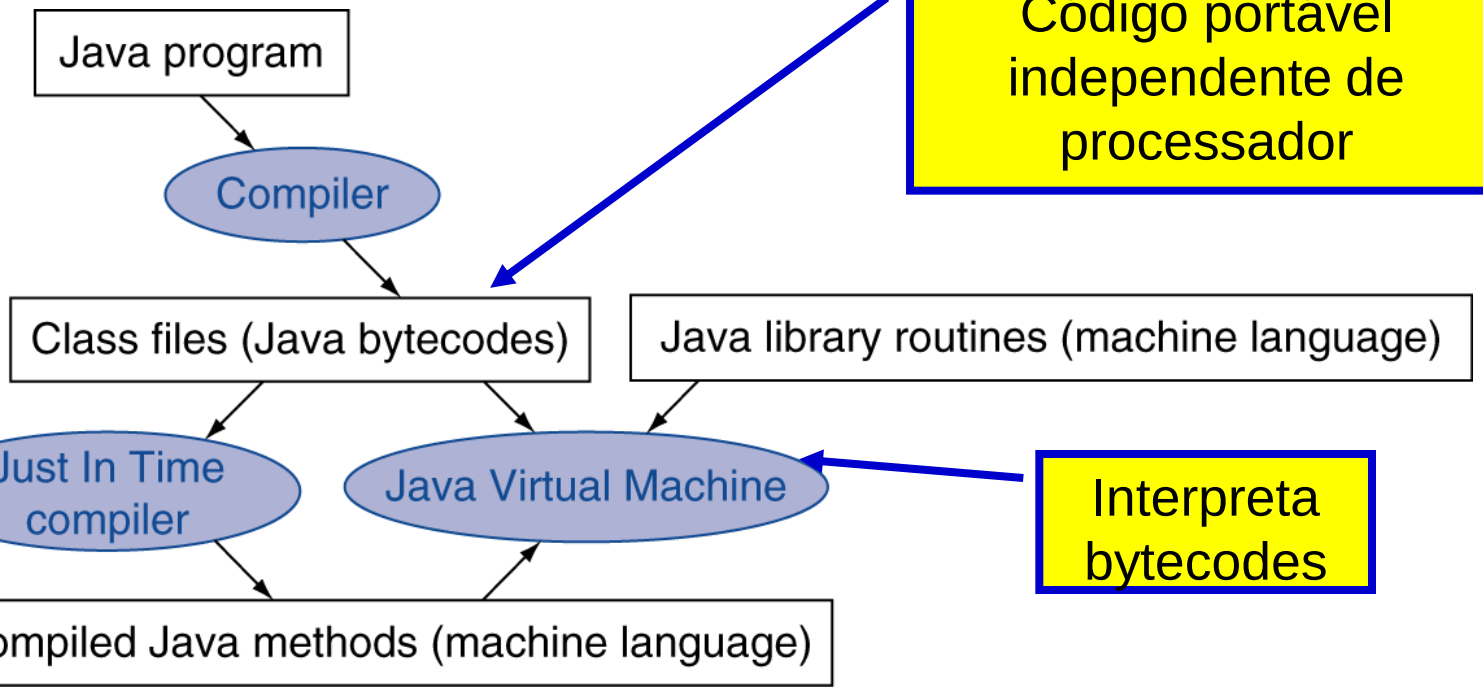
Ligando Dinâmica de Módulos de Objetos

- Abordagem tradicional de ligação é estática
  - Códigos das rotinas dos módulos externos chamados no código do módulo principal são incorporados ao executável
    - Mesmo se não forem executados
  - Se os módulos externos forem modificados, o executável continua com o código antigo
- Solução: Ligação Dinâmica
  - **DLL**s (**D**ynamic **L**inked **L**ibraries)
  - Fazer ligação apenas quando a rotina precisar ser executada
  - Requer que programa e rotinas armazenem informações extras
    - Nome das rotinas
    - Local das rotinas

Carregando um Programa

- Loader é um programa responsável por carregar o executável na memória
  - Faz parte do sistema operacional
- Passos para carregar executável na memória
  1. Lê header para determinar tamanhos dos segmentos
  2. Cria espaço na memória
  3. Copia texto e inicializa dados na memória
  4. Coloca argumentos da main na pilha
  5. Inicializa registradores (incluindo $sp, $gp…)
  6. Pula para rotina de início
     - Copia argumentos para $a0, … e chama main
     - Quando main retorna, sai do programa

Executando Aplicações Java



Código portável independente de processador

Interpreta bytecodes

Compila bytecodes de metódos mais custosos para linguagem de máquina

JVM (Java Virtual Machine)

- JVM é uma máquina virtual que possui seu próprio repertório de instruções
  - Bytecodes são as instruções entendidas pelo JVM
  - Arquivos com bytecodes também possuem informações sobre os tipos dos objetos
- JVM faz a ligação (link) do programa principal com os métodos das bibliotecas (pacotes) padrão
  - Funciona também como um loader
- Pode chamar o JIT para melhorar desempenho
- JIT identifica os métodos mais lentos e usados, compila-os para linguagem de máquina e salva o código compilado
  - Métodos tem melhor desempenho nas futuras execuções

Características de Bytecodes Java

- Bytecodes (instruções) variam de tamanho
  - 1 a 5 bytes
- Operandos não ficam em registradores, e sim numa pilha
- JVM garante execução segura dos bytecodes
  - Ex: instruções envolvendo arrays não depassam limites do array
- Instruções diferentes para operar com endereços e inteiros
- Instruções complexas
  - Alocação de arrays, invocação de um método, etc

Bytecodes (Instruções) Aritméticos

Operation	Java bytecode	Size (bits)	MIPS Instr.
add	iadd	8	add
subtract	isub	8	sub
increment	iinc I8a I8b	8	addi

Prefixo das instruções: i – trabalha com inteiros de 32 bits
a – trabalha com endereços
s – trabalha com inteiros de 16 bits (short)
b - trabalha com inteiros de 8 bits (byte)
I8 – constante de 8 bits

Bytecodes (Instruções) de Transferência de Dados

Operation	Java bytecode	Size (bits)	MIPS Instr.
load local integer/address	iload I8/aload I8	16	lw
load local integer/address	iload_/aload_{0,1,2,3}	8	lw
store local integer/address	istore I8/astore I8	16	sw
load integer/address from array	iaload/aaload	8	lw
store integer/address into array	iastore/aastore	8	sw
load half from array	saload	8	lh
store half into array	sastore	8	sh
load byte from array	baload	8	lb
store byte into array	bastore	8	sb
load immediate	bipush I8, sipush I16	16, 24	addi
load immediate	iconst_{–1,0,1,2,3,4,5}	8	addi

Bytecodes (Instruções) Lógicos

Operation	Java bytecode	Size (bits)	MIPS Instr.
and	iand	8	and
or	ior	8	or
shift left	ishl	8	sll
shift right	iushr	8	srl

Bytecodes (Instruções) de Desvios Condicionais

Operation	Java bytecode	Size (bits)	MIPS instr.
branch on equal	if_icompeq l16	24	beq
branch on not equal	if_icompne l16	24	bne
compare	if_icomp{lt,le,gt,ge} l16	24	slt

Bytecodes (Instruções) de Desvios Incondicionais

Operation	Java bytecode	Size (bits)	MIPS instr.
jump	goto l16	24	j
return	ret, ireturn	8	jr
jump to subroutine	jsr l16	24	jal

Bytecodes (Instruções) Complexas

Operation	Java bytecode	Size (bits)	MIPS instr.
check for null reference	ifnull I16, ifnotnull I16	24	
get length of array	arraylength	8	
check if object a type	instanceof I16	24	
invoke method	invokevirtual I16	24	
create new class instance	new I16	24	
create new array	newarray I16	24	

Implementando Loops com Bytecodes Java

Código Java

```
while (save[i] == h)
  i += 1;
```

`i`, `h` e `save` são as primeiras variáveis locais de um método

Bytecodes Java

Endereço do byte

Instrução de 3 bytes

```
0 aload_3 # Empilhe 3ª variável(save)
1 iload_1 #Empilhe 1ª variável(i)
2 iaload  #Empilhe elemento do array(save[i])
3 iload_2 #Empilhe 2ª variável (h)
4 if_icompne, 13 #Compara e sai se !=
7 iinc,1,1 # Incrementa variável 1(i)em 1
10 go to 0 #Vá para o topo do loop (endereço 0)
13 ... #Sai do loop
```

História do ISA Intel x86

- Evolução garantindo compatibilidade com versões anteriores
  - 8080 (1974): microprocessador de 8 bits
  - 8086 (1978): extensão do 8080 para 16 bits
    - Registradores dedicados de 16 bits
  - 8087 (1980): coprocessador para ponto flutuante
    - Adiciona instruções de ponto flutuante e pilha
  - 80286 (1982): endereços de 24 bits e MMU
    - Modelo de proteção da memória
  - 80386 (1985): extensão para 32 bits (chamado de IA-32)
    - Modos de endereçamento e operações adicionais
    - Registradores e endereçamento de 32 bits
  - i486 (1989): utilização de pipeline, caches on-chip
    - Competidores compatíveis: AMD, Cyrix, …

História do ISA Intel x86

- Evoluindo sempre…
  - Pentium (1993): superescalar
    - Versões posteriores incluiram instruções MMX (Multi-Media eXtension)
    - O famoso bug da instrução FDIV
  - Pentium III (1999)
    - Adicionou instruções SSE (Streaming SIMD Extensions) e registradores associados
    - 70 instruções adicionadas, 4 operações com ponto flutuante de precisão simples podiam ser feitas paralelamente
  - Pentium 4 (2001)
    - Adicionou 144 instruções (SSE2)
    - Permitia que operações com ponto flutuante com precisão dupla fossem realizadas paralelamente

História do ISA Intel x86

- E mais ainda…
  - AMD64 (2003): estendeu a arquitetura para 64 bits
    - Aumentou registradores para 64 bits
    - Aumentou o número de registradores
  - EM64T – Extended Memory 64 Technology (2004)
    - AMD64 adotada pela Intel (com refinamentos)
    - Adicionou 13 instruções SSE3
  - Intel Core (2006)
    - Adicionou 54 instruções SSE4
  - AMD64 (2007): mais 170 instruções SSE5
    - Introduziu instruções com 3 operandos como o MIPS

Registradores do x86 (Arquitetura de 32 bits)

Apenas 8 registradores de propósito geral

Registradores de 16 bits

Modos de Endereçamento Básicos do x86

- Dois operandos
  - Em instruções aritméticas/lógicas primeiro operando é origem e destino
  - Toda instrução pode ter um operando na memória

Operando origem/dest	Segundo operando origem
Registrador	Registrador
Registrador	Imediato
Registrador	Memória
Memória	Registrador
Memória	Imediato

Modos de Endereçamento Básicos do x86

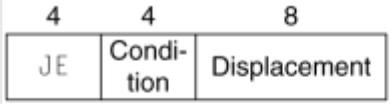
- Modos de endereçamento comuns
  - Registrador, imediato, endereçamento base
- Modos de endereçamento mais complexos
  - Registrador indireto
    - endereço no registrador
  - Endereçamento base mais índice escalar
    - $R_{base} + 2^{scale} \times R_{index}$ (scale = 0, 1, 2, or 3)
  - Endereçamento base mais índice escalar com deslocamento
    - $R_{base} + 2^{scale} \times R_{index}$ + deslocamento
- Dependendo do modo de endereçamento, certos registradores não podem ser utilizados
  - Ex: registrador indireto não pode utilizar o ESP ou EBP

Algumas Instruções do x86

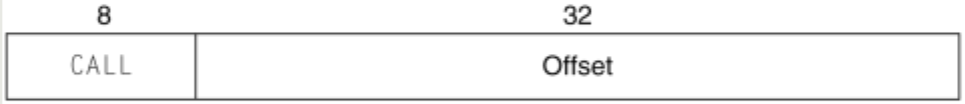
Instruction	Meaning
Control	**Conditional and unconditional branches**
`jnz, jz`	Jump if condition to EIP + 8-bit offset; `JNE` (for `JNZ`), `JE` (for `JZ`) are alternative names
`jmp`	Unconditional jump—8-bit or 16-bit offset
`call`	Subroutine call—16-bit offset; return address pushed onto stack
`ret`	Pops return address from stack and jumps to it
`loop`	Loop branch—decrement ECX; jump to EIP + 8-bit displacement if ECX ≠ 0
Data transfer	**Move data between registers or between register and memory**
`move`	Move between two registers or between register and memory
`push, pop`	Push source operand on stack; pop operand from stack top to a register
`les`	Load ES and one of the GPRs from memory
Arithmetic, logical	**Arithmetic and logical operations using the data registers and memory**
`add, sub`	Add source to destination; subtract source from destination; register-memory format
`cmp`	Compare source and destination; register-memory format
`shl, shr, rcr`	Shift left; shift logical right; rotate right with carry condition code as fill
`cbw`	Convert byte in eight rightmost bits of EAX to 16-bit word in right of EAX
`test`	Logical AND of source and destination sets condition codes
`inc, dec`	Increment destination, decrement destination
`or, xor`	Logical OR; exclusive OR; register-memory format
String	**Move between string operands; length given by a repeat prefix**
`movs`	Copies from string source to destination by incrementing ESI and EDI; may be repeated
`lods`	Loads a byte, word, or doubleword of a string into the EAX register

Formato de Instrução do x86

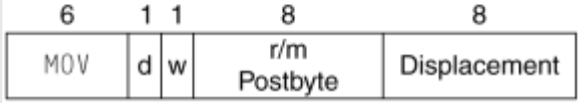
a. JE EIP + displacement



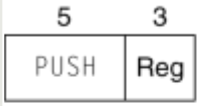
b. CALL



c. MOV EBX, [EDI + 45]



d. PUSH ESI



e. ADD EAX, #6765

f. TEST EDX, #42

- Formatos com tamanhos variados
  - Variam de 1 a 17 bytes
  - Bytes pós-fixados especificam modo de endereçamento
  - Bytes pré-fixados modificam operação
    - Tamanho do operando, modo de endereçamento

Algumas Conclusões sobre x86

- Instruções complexas tornam implementação difícil
  - Hardware traduz instruções para micro-operações mais simples
    - Instruções simples: 1–1
    - Instruções complexas : 1–muitos
  - Microengine similar ao RISC
  - Mercado torna o processador economicamente viável
- Desempenho comparável ao RISC
  - Compiladores evitam instruções complexas
- Compatibilidade amarra projeto do processador
  - Elegância técnica ≠ sucesso de mercado