AULA Nº 08
ORGANIZAÇÃO DE COMPUTADORES

Tradução, compilação e desempenho

Ligando objetos

Produz imagem executável
1. **Junta segmentos**
2. **Resolve rótulos (determina endereços)**
Ex.: instruções de desvio ou salto
3. **Corrige referências dependentes de local e externas**

Tradução

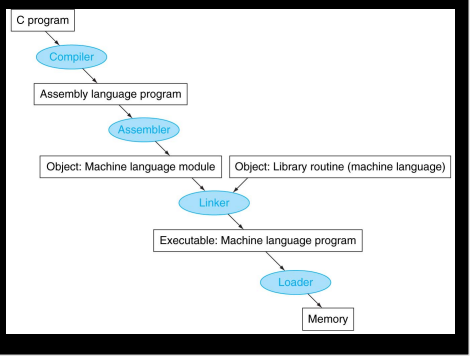

FIGURE 2.21 A translation hierarchy for C. A high-level language program is first compiled into an assembly language program and then assembled into an object module in machine language. The linker combines multiple modules with library routines to resolve all references. The loader then places the machine code into the proper memory locations for execution by the processor. To speed up the translation process, some steps are skipped or combined. Some compilers produce object modules directly, and some systems use linking loaders that perform the last two steps. To identify the type of file, UNIX follows a suffix convention for files: C source files are named x.c, assembly files are x.s, object files are named x.o, statically linked library routines are x.a, dynamically linked library routes are x.so, and executable files by default are called a.out. MS-DOS uses the suffixes .C, .ASM, .OBJ, .LIB, .DLL, and .EXE to the same effect.

Carregando um programa

Carrega imagem do disco para memória
1. **Lê header para obter tamanho de segmentos**
2. **Cria espaço de endereçamento virtual**
3. **Copia segmento de texto e inicializa dados**
4. **Coloca argumentos na pilha**
5. **Inicializa registradores (inclusive $sp, $fp, $gp)**
6. **Salta para rotina inicial**
   copia argumentos para $a0, … e inicia
   quando sair, chama syscall

Produzindo um objeto

***Assembler* traduz e provê informações.**
Header: conteúdo do módulo objeto
Segmento de texto: instruções traduzidas
Segmento de dados estático: toda duração
Info de relocação: para conteúdo que depende de referências absolutas do programa carregado
Tabela de símbolos: definições globais e referências externas
Info de debug: para associar a código fonte

Memória

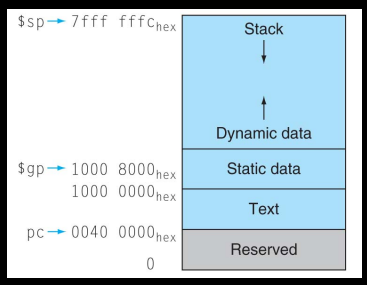

FIGURE 2.13 The MIPS memory allocation for program and data. These addresses are only a software convention, and not part of the MIPS architecture. The stack pointer is initialized to $7fff\ fffc_{hex}$ and grows down toward the data segment. At the other end, the program code ("text") starts at $0040\ 0000_{hex}$. The static data starts at 1000 0000hex. Dynamic data, allocated by malloc in C and by new in Java, is next. It grows up toward the stack in an area called the heap. The global pointer, $gp, is set to an address to make it easy to access data. It is initialized to $1000\ 8000_{hex}$ so that it can access from $1000\ 0000_{hex}$ to $1000\ ffff_{hex}$ using the positive and negative 16-bit offsets from $gp. This information is also found in Column 4 of the MIPS Reference Data Card at the front of this book.

Iniciando aplicações em Java

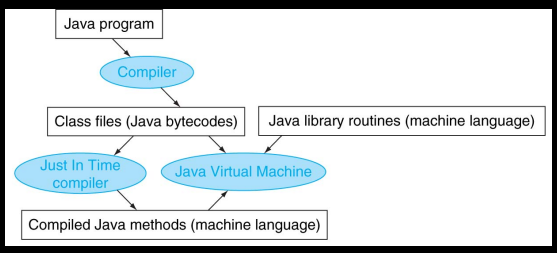

FIGURE 2.23 A translation hierarchy for Java. A Java program is first compiled into a binary version of Java bytecodes, with all addresses defined by the compiler. The Java program is now ready to run on the interpreter, called the Java Virtual Machine (JVM). The JVM links to desired methods in the Java library while the program is running. To achieve greater performance, the JVM can invoke the JIT compiler, which selectively compiles methods into the native machine language of the machine on which it is running.

Compilador e desempenho

gcc optimization	Relative performance	Clock cycles (millions)	Instruction count (millions)	CPI
None	1.00	158,615	114,938	1.38
01 (medium)	2.37	66,990	37,470	1.79
02 (full)	2.38	66,521	39,993	1.66
03 (procedure integration)	2.41	65,747	44,993	1.46

FIGURE 2.28 Comparing performance, instruction count, and CPI using compiler optimization for Bubble Sort. The programs sorted 100,000 words with the array initialized to random values. These programs were run on a Pentium 4 with a clock rate of 3.06 GHz and a 533 MHz system bus with 2 GB of PC2100 DDR SDRAM. It used Linux version 2.4.20.

Explicações sobre otimizações: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

Exemplo – Ordenação

```
void swap(int v[], int k)
{
 int temp;
 temp = v[k];
 v[k] = v[k+1];
 v[k+1] = temp;
}
```

FIGURE 2.24 A C procedure that swaps two locations in memory. This subsection uses this procedure in a sorting example.

Procedure body

```
swap: sll   $t1, $a1, 2      # reg $t1 = k * 4
      add   $t1, $a0, $t1    # reg $t1 = v + (k * 4)
                             # reg $t1 has the address of v[k]
      lw    $t0, 0($t1)      # reg $t0 (temp) = v[k]
      lw    $t2, 4($t1)      # reg $t2 = v[k + 1]
                             # refers to next element of v
      sw    $t2, 0($t1)      # v[k] = reg $t2
      sw    $t0, 4($t1)      # v[k+1] = reg $t0 (temp)
```

Procedure return

```
      jr    $ra              # return to calling routine
```

FIGURE 2.25 MIPS assembly code of the procedure swap in Figure 2.24.

Linguagens e algoritmos

Language	Execution method	Optimization	Bubble Sort relative performance	Quicksort relative performance	Speedup Quicksort vs. Bubble Sort
C	Compiler	None	1.00	1.00	2468
	Compiler	01	2.37	1.50	1562
	Compiler	02	2.38	1.50	1555
	Compiler	03	2.41	1.91	1955
Java	Interpreter	–	0.12	0.05	1050
	JIT compiler	–	2.13	0.29	338

FIGURE 2.29 Performance of two sort algorithms in C and Java using interpretation and optimizing compilers relative to unoptimized C version. The last column shows the advantage in performance of Quicksort over Bubble Sort for each language and execution option. These programs were run on the same system as Figure 2.28. The JVM is Sun version 1.3.1, and the JIT is Sun Hotspot version 1.3.1.

IC e CPI não são bons indicadores de desempenho isoladamente!

Otimizações do compilador são sensíveis ao algoritmo.

Código compilado é mais rápido do que o interpretado.

Exemplo – Ordenação

```
void sort (int v[], int n)
{
  int i, j;
  for (i = 0; i < n; i += 1) {
    for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j -= 1) {
      swap(v,j);
    }
  }
}
```

FIGURE 2.26 A C procedure that performs a sort on the array v.

Saving registers

```
sort:   addi  $sp,$sp, -20      # make room on stack for 5 registers
        sw    $ra, 16($sp)      # save $ra on stack
        sw    $s3,12($sp)       # save $s3 on stack
        sw    $s2, 8($sp)       # save $s2 on stack
        sw    $s1, 4($sp)       # save $s1 on stack
        sw    $s0, 0($sp)       # save $s0 on stack
```

Procedure body

```
Move parameters
        move  $s2, $a0          # copy parameter $a0 into $s2 (save $a0)
        move  $s3, $a1          # copy parameter $a1 into $s3 (save $a1)
Outer loop
        move  $s0, $zero        # i = 0
for1tst: slt  $t0, $s0, $s3     # reg $t0 = 0 if $s0 ≥ $s3 (i ≥ n)
        beq   $t0, $zero, exit1 # go to exit1 if $s0 ≥ $s3 (i ≥ n)
Inner loop
        addi  $s1, $s0, -1      # j = i - 1
for2tst: slti $t0, $s1, 0       # reg $t0 = -1 if $s1 < 0 (j < 0)
        bne   $t0, $zero, exit2 # go to exit2 if $s1 < 0 (j < 0)
        sll   $t1, $s1, 2       # reg $t1 = j * 4
        add   $t2, $s2, $t1     # reg $t2 = v + (j * 4)
        lw    $t3, 0($t2)       # reg $t3 = v[j]
        lw    $t4, 4($t2)       # reg $t4 = v[j + 1]
        slt   $t0, $t4, $t3     # reg $t0 = 0 if $t4 ≥ $t3
        beq   $t0, $zero, exit2 # go to exit2 if $t4 ≥ $t3
Pass parameters and call
        move  $a0, $s2          # 1st parameter of swap is v (old $a0)
        move  $a1, $s1          # 2nd parameter of swap is j
        jal   swap              # swap code shown in Figure 2.25
Inner loop
        addi  $s1, $s1, -1      # j -= 1
        j     for2tst           # jump to test of inner loop
Outer loop
exit2:  addi  $s0, $s0, 1       # i += 1
        j     for1tst           # jump to test of outer loop
```

Restoring registers

```
exit1:  lw    $s0, 0($sp)       # restore $s0 from stack
        lw    $s1, 4($sp)       # restore $s1 from stack
        lw    $s2, 8($sp)       # restore $s2 from stack
        lw    $s3,12($sp)       # restore $s3 from stack
        lw    $ra,16($sp)       # restore $ra from stack
        addi  $sp,$sp,20        # restore stack pointer
```

Procedure return

```
        jr    $ra               # return to calling routine
```

FIGURE 2.27 MIPS assembly version of procedure sort in Figure 2.26.

Distribuição de instruções

Instruction class	MIPS examples	HLL correspondence	Frequency	
			Integer	Ft. pt.
Arithmetic	`add, sub, addi`	Operations in assignment statements	16%	48%
Data transfer	`lw, sw, lb, lbu, lh, lhu, sb, lui`	References to data structures, such as arrays	35%	36%
Logical	`and, or, nor, andi, ori, sll, srl`	Operations in assignment statements	12%	4%
Conditional branch	`beq, bne, slt, slti, sltiu`	*If* statements and loops	34%	8%
Jump	`j, jr, jal`	Procedure calls, returns, and *case/switch* statements	2%	0%

FIGURE 2.45 MIPS instruction classes, examples, correspondence to high-level program language constructs, and percent age of MIPS instructions executed by category for the average SPEC2006 benchmarks.

ARM e MIPS

	ARM	MIPS
Date announced	1985	1985
Instruction size (bits)	32	32
Address space (size, model)	32 bits, flat	32 bits, flat
Data alignment	Aligned	Aligned
Data addressing modes	9	3
Integer registers (number, model, size)	15 GPR × 32 bits	31 GPR × 32 bits
I/O	Memory mapped	Memory mapped

FIGURE 2.31 Similarities in ARM and MIPS instruction sets.

Referências

Seções 2.12 a 2.19 - “Organização e Projeto de Computadores – A Interface Hardware/Software”, David A. Patterson & John L. Hennessy, Campus, 4 edição, 2013.

Instruções x86

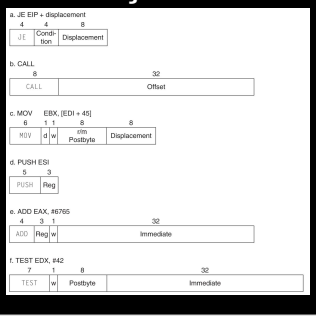

FIGURE 2.41 Typical x86 instruction formats. Figure 2.42 shows the encoding of the postbyte. Many instructions contain the 1-bit field w, which says whether the operation is a byte or a double word. The d field in MOV is used in instructions that may move to or from memory and shows the direction of the move. The ADD instruction requires 32 bits for the immediate field, because in 32-bit mode, the immediates are either 8 bits or 32 bits. The immediate field in the TEST is 32 bits long because there is no 8-bit immediate for test in 32-bit mode. Overall, instructions may vary from 1 to 17 bytes in length. The long length comes from extra 1-byte prefixes, having both a 4-byte immediate and a 4-byte displacement address, using an opcode of 2 bytes, and using the scaled index mode specifier, which adds another byte.

Número de instruções x86

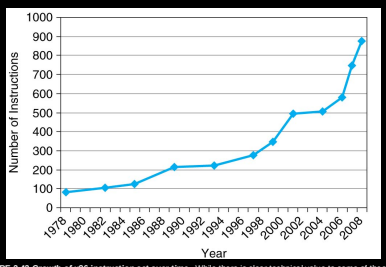

FIGURE 2.43 Growth of x86 instruction set over time. While there is clear technical value to some of these extensions, this rapid change also increases the difficulty for other companies to try to build compatible processors.