Insper

Elementos de Sistemas

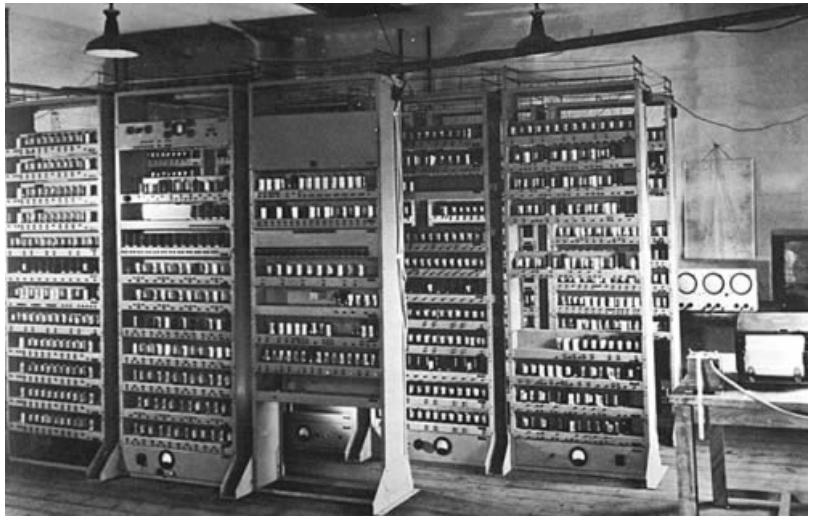
Conteúdo 15 – Assembler

"O que há num nome? O que chamamos de rosa com qualquer outro nome teria o mesmo aroma doce"
"What's in a name? That which we call a rose by any other name would smell as sweet."
William Shakespeare (1564–1616), Poeta Inglês
apud Nisan, N. & Schocken, S. 2005. Elements of Computing Systems

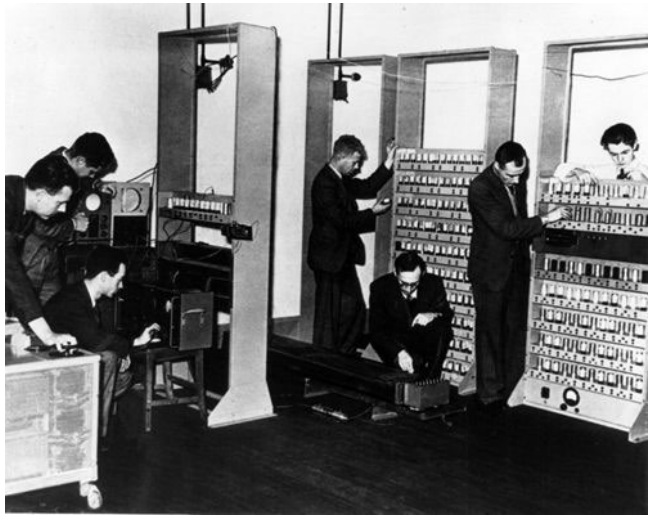
Engenharia - Luciano Soares <lpsoares@insper.edu.br>

EDSAC

O EDSAC (Electronic Delay Storage Automatic Calculator) é creditado como um dos primeiros computadores a utilizar um Assembler.



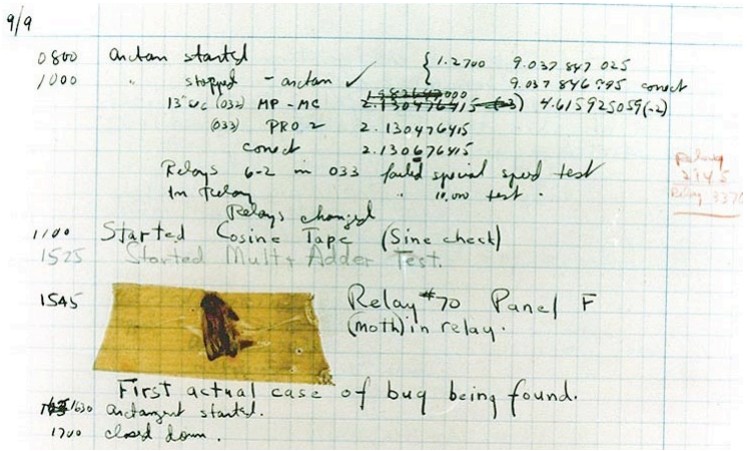
Painel do EDSAC.



Equipe trabalhando no EDSAC.

Grace Hopper

Grace Hopper foi uma pioneira na Informática, desenvolvendo linguagens como o COBOL (COmmon Business-Oriented Language).



Primeiro caso de um bug (mariposa) sendo encontrado no Mark II por Grace Murray Hopper

Grace Murray Hopper
(1906 – 1992)

Insper

Assembly

Linguagens Assembly em notação AT&A.

Assembly Arquitetura MIPS

```
    add     $t0, $gp, $zero    # &A[0] - 28
    lw      $t1, 4($gp)        # fetch N
    sll     $t1, $t1, 2        # N as byte offset
    add     $t1, $t1, $gp      # &A[N] - 28
    ori     $t2, $zero, 256    # MAX_SIZE
top:
    sltu    $t3, $t0, $t1      # have we reached the final address?
    beq     $t3, $zero, done   # yes, we're done
    sw      $t2, 28($t0)       # A[i] = 0
    addi    $t0, $t0, 4        # update $t0 to point to next element
    j       top                # go to top of loop
done:
```

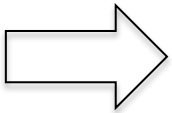
código C

```
for (i=0; i<N; i++) {A[i] = MAX_SIZE;}
```

O que um Assembler faz?

Assemblers são o primeiro nível da hierarquia de software. Eles convertem um arquivo com operações de mnemônicos (Assembly) para Código de Máquina.

```
mov     edx,2
mov     esi,4
add     eax,ebx
sub     eax,ecx
imul    edx,eax
mov     eax,edx
mov     edx,0
cmp     esi,0
```



```
ce fa ed fe 07 00 00 00 03 00 00 00 01 00 00 00
04 00 00 00 38 01 00 00 00 00 00 00 01 00 00 00
c0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 28 00 00 00 54 01 00 00
28 00 00 00 07 00 00 00 07 00 00 00 02 00 00 00
00 00 00 00 5f 5f 74 65 78 74 00 00 00 00 00 00
00 00 00 00 5f 5f 54 45 58 54 00 00 00 00 00 00
00 00 00 00 00 00 00 00 1b 00 00 00 54 01 00 00
00 00 00 00 7c 01 00 00 02 00 00 00 00 04 00 80
00 00 00 00 00 00 00 00 5f 5f 64 61 74 61 00 00
00 00 00 00 00 00 00 00 5f 5f 44 41 54 41 00 00
00 00 00 00 00 00 00 00 1b 00 00 00 0d 00 00 00
6f 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 24 00 00 00
10 00 00 00 00 0a 0a 00 00 00 00 00 02 00 00 00
18 00 00 00 8c 01 00 00 04 00 00 00 bc 01 00 00
18 00 00 00 0b 00 00 00 50 00 00 00 00 00 00 00
02 00 00 00 02 00 00 00 01 00 00 00 03 00 00 00
01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Programa **Executável**

Tipos de Assemblers

Conjuntos de sintaxe mais populares.

NASM (Sintaxe Intel)

```
001 ; Text segment begins
002  section .text
003
004      global _start
005
006 ; Program entry point
007     _start:
008
009 ; Put the code number for system call
010         mov   eax, 1
011
012 ; Return value
013         mov   ebx, 2
014
015 ; Call the OS
016         int   80h
```

GAS (Sintaxe AT&T)

```
# Text segment begins
.section .text

      .globl _start

# Program entry point
    _start:

# Put the code number for system call
        movl  $1, %eax

/* Return value */
        movl  $2, %ebx

# Call the OS
        int   $0x80
```

Análise Sintática (Parsing)

Separar cada termo para a análise:

```
     leaw $i,%A
     movw $1,(%A)
     leaw $sum,%A
     movw $0,(%A)
LOOP:
     leaw $i,%A
     movw (%A),%D
     leaw $R0,%A
     subw %D,(%A),%D
     leaw $LOOP,%A
     jg
```


```
{"leaw","$i","%A",
"movw","$1","(%A)",
"leaw","$sum","%A",
"movw","$0","(%A)"
"LOOP:",
"leaw","$i","%A",
"movw","(%A)","%D",
"leaw","$R0","%A",
"subw","%D","(%A)","%D",
"leaw","$LOOP","%A",
"jg"}
```

Funcionamento Básico

Mnemônico da instrução é a principal referência para se detectar o opcode (código de instrução em linguagem de máquina).

`leaw $i,%A`	`0000000000010000`
`movw $1,(%A)`	`1110111111001000`
`leaw $sum,%A`	`0000000000010001`
`movw $0,(%A)`	`1110101010001000`
`LOOP:`	
`leaw $i,%A`	`0000000000010000`
`movw (%A),%D`	`1111110000010000`
`leaw $R0,%A`	`0000000000000000`
`subw %D,(%A),%D`	`1111010011010000`
`leaw $LOOP,%A`	`0000000000000100`
`jg`	`1110001100000001`

Assembler (Símbolos)

Uma tabela de símbolos (Symbol Table) é usada para esta gestão de memória.

```
     leaw $i,%A
     movw $1,(%A)
     leaw $sum,%A
     movw $0,(%A)
LOOP:
     leaw $i,%A
     movw (%A),%D
     leaw $R0,%A
     subw %D,(%A),%D
     leaw $LOOP,%A
     jg
```

Símbolo	Endereço
R0	0
R1	1
R2	2
R3	3
...	...
i	16
sum	17
LOOP	4

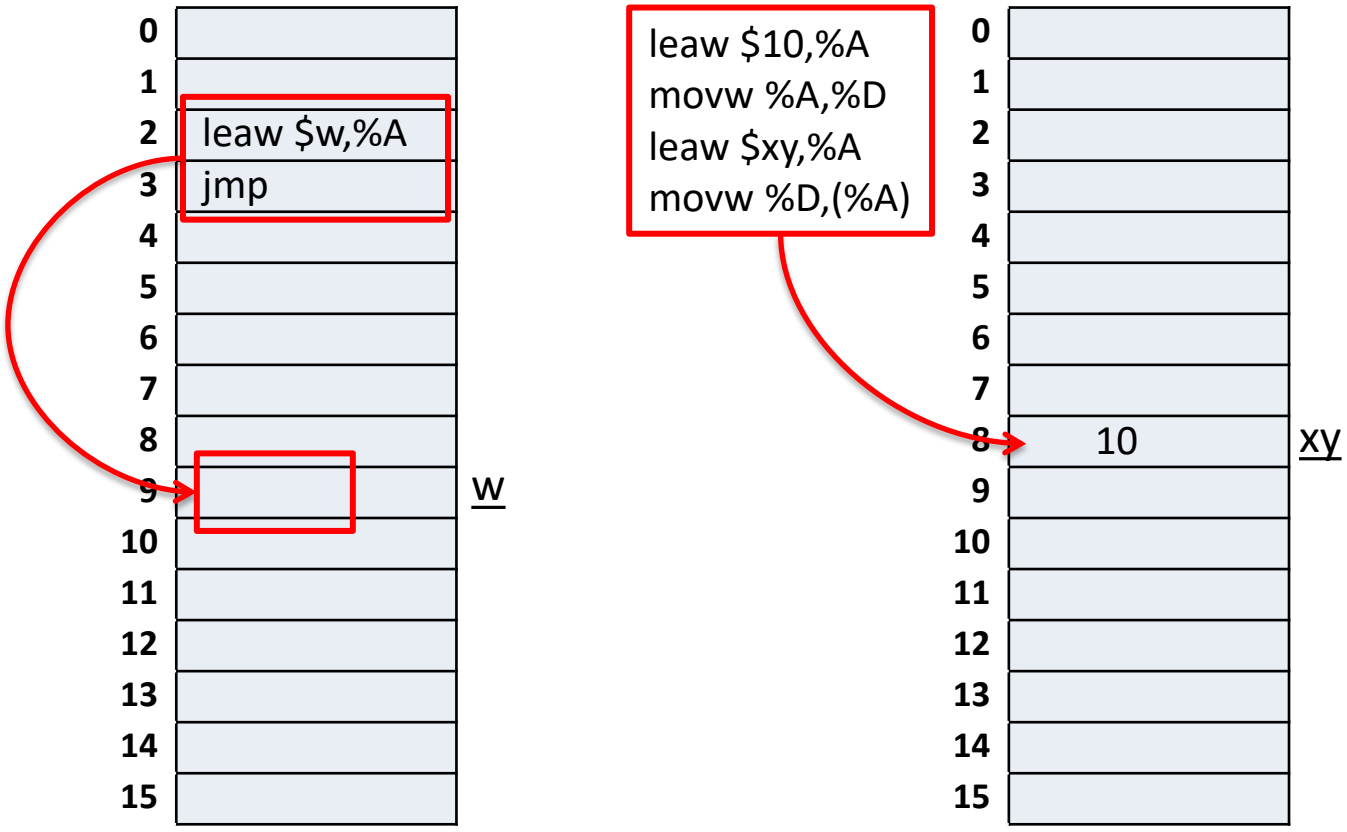
Símbolos

No desenvolvimento em linguagem de máquina, endereços de dados ou de instruções na memória são descritos diretamente. Por exemplo:

Insper

Símbolos

Para simplificar o desenvolvimento, os Assemblers podem gerenciar posições de memórias usando símbolos para isso. Por exemplo:



Insper

Símbolos

Variáveis : valores para dados.

Java

```
int dez = 10;
char a = 'a';
```

***Labels* :** valores para saltos (jump)

Basic

```
10 PRINT "Hello World!"
20 GOTO 10
```


* Instruções de goto não são recomendadas em geral em linguagens de alto nível.

Passos do Assembler

- Assembler de **um passo:** percorrer o código-fonte uma vez.

```
    leaw $i,%A
    movw $1,(%A)
    leaw $sum,%A
    movw $0,(%A)
LOOP:
    leaw $i,%A
    movw (%A),%D
    leaw $R0,%A
    subw %D,(%A),%D
    leaw $LOOP,%A
    jg
```

- Assembler de **múltiplos passos:** em uma primeira passagem criar a tabela de símbolo e depois usa.

1º

```
    leaw $i,%A
    movw $1,(%A)
    leaw $sum,%A
    movw $0,(%A)
LOOP:
    leaw $i,%A
    movw (%A),%D
    leaw $R0,%A
    subw %D,(%A),%D
    leaw $LOOP,%A
    jg
```

2º

```
    leaw $i,%A
    movw $1,(%A)
    leaw $sum,%A
    movw $0,(%A)
LOOP:
    leaw $i,%A
    movw (%A),%D
    leaw $R0,%A
    subw %D,(%A),%D
    leaw $LOOP,%A
    jg
```

Tabelas Auxiliares

- **Operation Code Table (OPTAB)**
  - Com os opcodes e formato de instruções
  - Eficiente para recuperar dados
  - Estática
- **Symbol Table (SYMTAB)**
  - Com os símbolos (variáveis e labels)
  - Eficiente para inserir e recuperar dados
  - Dinâmica

Por que aprender Assembly ?

Por exemplo para melhor controle de instruções otimizadas:

```
#ifdef CONFIG_SMP
#define LOCK_PREFIX "lock ; "
#else
#define LOCK_PREFIX ""
#endif

#define __xg(x) ((volatile long *)(x))

static inline unsigned long
   __cmpxchg(volatile void *ptr, unsigned long old, unsigned long newv) {
    unsigned long prev;
    __asm__ __volatile__(LOCK_PREFIX "cmpxchgb %b1,%2”
                         : "=a"(prev)
                         : "q"(newv), "m"(*__xg(ptr)), "0"(old)
                         : "memory");
    return prev;
```

Resumo do Assembler

Repete até o fim do arquivo assembly

- Lê a próxima instrução Assembly;
- Separa a instrução nos seus campos;
- Busca o código binário para cada campo da instrução;
- Combina os códigos binários numa instrução de máquina;
- Grava a instrução de máquina em um arquivo executável;

Boas práticas para programação

Códigos devem ser:

- curtos
- eficientes
- elegantes
- auto-descritivos

Insper

www.insper.edu.br