

Elementos de Sistemas

Aula 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



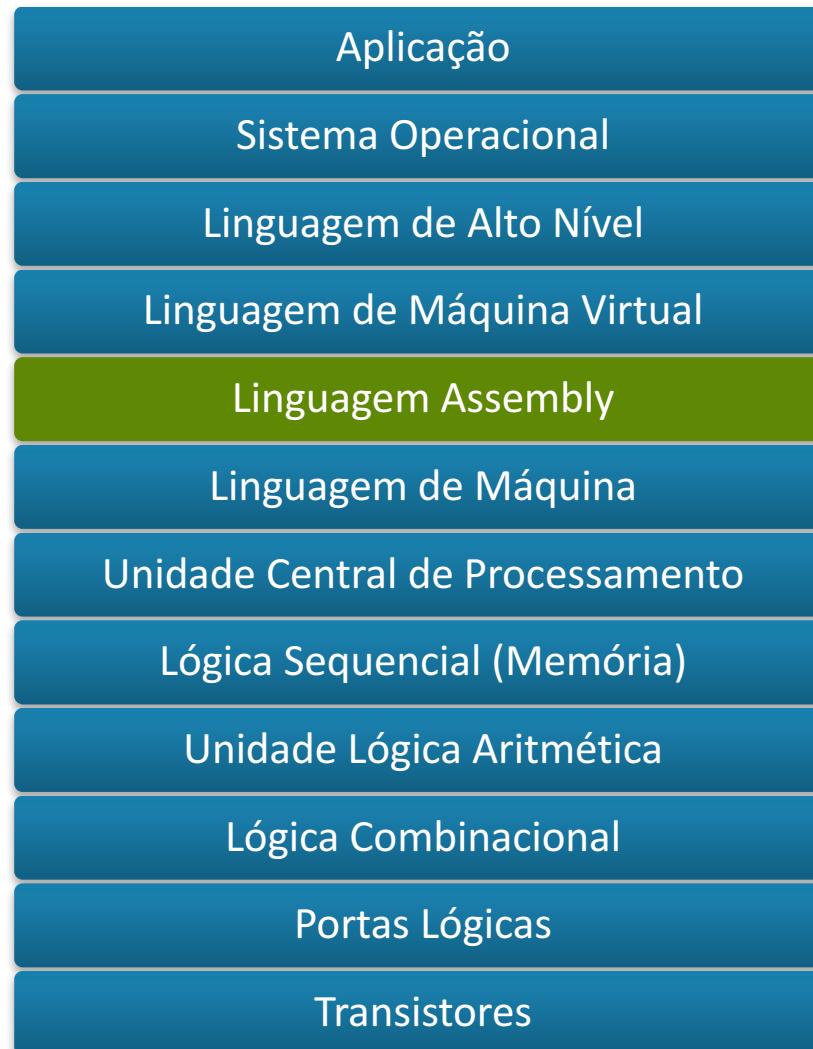
Objetivos de Aprendizado da Aula

- Desenvolver um Assembler;
- Fazer o *parsing* de arquivos;
- Gerenciar uma tabela de símbolos;
- Mapear instruções Assembly em Linguagem de Máquina.

Conteúdo(s): Geração e Otimização de Código.



Níveis de Abstração



Assembler on line

Códigos abaixo no formato Intel.

The screenshot shows the CodingGround online assembly compiler interface. The main window displays the assembly code for a 'Hello, world!' program. The code is as follows:

```
1 section .text
2     global _start          ;must be declared for using gcc
3 _start:
4     mov edx, len           ;message length
5     mov ecx, msg            ;message to write
6     mov ebx, 1               ;file descriptor (stdout)
7     mov eax, 4               ;system call number (sys_write)
8     int 0x80                ;call kernel
9     mov eax, 1               ;system call number (sys_exit)
10    int 0x80                ;call kernel
11
12 section .data
13
14 msg db 'Hello, world!',0xa ;our dear string
15 len equ $ - msg           ;length of our dear string
16
```

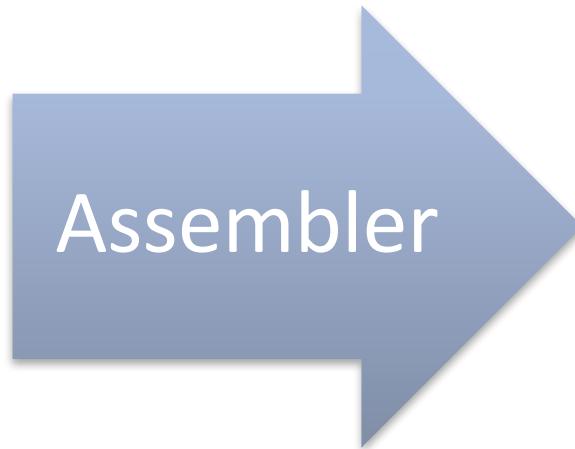
The interface includes a file tree on the left showing a project named 'root' with a file 'main.asm'. On the right, there's a sidebar with links to C++, JAVA Programming, Python, Ruby, and Swift. At the bottom, a terminal window shows the command-line output of the assembly compilation and execution:

```
sh-4.3$ nasm -f elf *.asm; ld -m elf_i386 -s -o demo *.o
sh-4.3$ demo
Hello, world!
sh-4.3$
```

Assembler



```
.NASM  
leaw $i,%A  
movw (%A),%D  
leaw $R0,%A  
subw %D,(%A),%D
```



```
.MIF  
000000000010000  
111111000010000  
0000000000000000  
1111010011010000
```

Arquivo de texto.

Cada linha podendo ser:

- Comentário
- Símbolo de endereçamento
- Instrução

Arquivo de texto.

Cada linha podendo ser:

- Cabeçalho com definições dos dados
- Instrução em binário (sequencia de 16 dígitos, de caracteres ASCII 0 e 1)



Formato de arquivos NASM (para o Z0)

Constantes:

São números que podem ser passados para o computador usando a instrução leaw. Esses número devem ser sempre positivos, informados em decimal.

Símbolos:

São uma sequencia de caracteres que definem endereços de memória de forma automática. Os símbolos não podem começar com um número.

Comentários:

Qualquer texto entre - (dois traços) e o fim da linha é considerado um comentário e deve ser ignorado pelo Assembler.

Espaço em Branco:

Espaços em branco, ou linhas em branco são ignoradas.

Maiúsculas e Minúsculas:

As instruções são sempre em minúsculas, já os símbolos podem ser dos dois tipos, porém o Assembler diferencia, assim "SOMA" é diferente de "soma".



Tabela de Símbolos

Símbolos de variáveis:

Qualquer símbolo definido pelo usuário em um programa é tratado como uma variável, e é automaticamente atribuído um endereço RAM único, começando no endereço RAM 16 (0x0010)

Registradores virtuais:

Os símbolos R0, ..., R15 são automaticamente predefinidos para se referir aos endereços de RAM 0, ..., 15

Ponteiros de I/O :

Os símbolos SCREEN e KBD são automaticamente predefinidos para se referir aos endereços de RAM 16384 e 24576, respectivamente

Ponteiros de controle da VM:

os símbolos SP, LCL, ARG, THIS, e THAT são automaticamente predefinidos para se referir ao endereços de RAM 0-4, respectivamente

Assembler lendo Assembly (Parser)

```
; Calculos 1+...+RAM[0]
; Armazene a soma na RAM[1].
    leaw $i,%A
    movw $1,(%A)    // i = 1
    leaw $sum,%A
    movw $0,(%A)    // sum = 0
LOOP:
    leaw $i,%A //if i>RAM[0] goto WRITE
    movw (%A),%D
    leaw $0,%A
    subw %D,(%A),%D
    leaw $WRITE,%A
    jg
    nop
    leaw $i,%A    // sum += i
    movw (%A),%D
    leaw $sum,%A
    addw %D,(%A),%D
    movw %D,(%A)
    leaw $i,%A    // i++
    movw (%A),%D
    incw %D
    movw %D,(%A)
    leaw $LOOP,%A // goto LOOP
    jmp
    nop
WRITE:
    leaw $sum,%A
    movw (%A),%D
    leaw $1,%A
    movw %D,(%A)  // RAM[1] = the sum
END:
    leaw $END,%A
    jmp
    nop
```

Cada linha do texto pode ser :

- Instrução tipo A
- Instrução tipo C
- Declaração de um Símbolo (Label)
- Comentário ; ou linha em branco

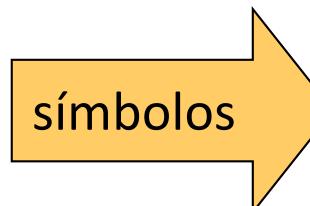
Para cada comando:

- Fazer o *parse* do comando
- Identificar se existem símbolos
- Identificar comandos e suas partes
- Substituir símbolos
- Gerar o código binário
- Juntar os códigos binários
- Escrever no arquivo de saída.

Tabela de Símbolos (Symbol Table)

```
; Calculos 1+...+RAM[0]
; Armazene a soma na RAM[1].
    leaw $i,%A
    movw $1,(%A)    // i = 1
    leaw $sum,%A
    movw $0,(%A)    // sum = 0
LOOP:
    leaw $i,%A //if i>RAM[0] goto WRITE
    movw (%A),%D
    leaw $0,%A
    subw %D,(%A),%D
    leaw $WRITE,%A
    jg
    nop
    leaw $i,%A    // sum += i
    movw (%A),%D
    leaw $sum,%A
    addw %D,(%A),%D
    movw %D,(%A)
    leaw $i,%A    // i++
    movw (%A),%D
    incw %D
    movw %D,(%A)
    leaw $LOOP,%A // goto LOOP
    jmp
    nop
WRITE:
    leaw $sum,%A
    movw (%A),%D
    leaw $1,%A
    movw %D,(%A) // RAM[1] = the sum
END:
    leaw $END,%A
    jmp
    nop
```

A tabela de símbolos já possuem alguns símbolos pré-definidos e outros são criado dinamicamente.



R0	0
R1	1
R2	2
...	...
R15	15
SCREEN	16384
KBD	24576
SP	0
LCL	1
ARG	2
THIS	3
THAT	4
LOOP	5
WRITE	22
END	28
i	16
sum	17

Geração de Código (Code)

```
; Calculos 1+...+RAM[0]
; Armazene a soma na RAM[1].
    leaw $i,%A
    movw $1,(%A)    // i = 1
    leaw $sum,%A
    movw $0,(%A)    // sum = 0
LOOP:
    leaw $i,%A //if i>RAM[0] goto WRITE
    movw (%A),%D
    leaw $0,%A
    subw %D,(%A),%D
    leaw $WRITE,%A
    jg
    nop
    leaw $i,%A    // sum += i
    movw (%A),%D
    leaw $sum,%A
    addw %D,(%A),%D
    movw %D,(%A)
    leaw $i,%A    // i++
    movw (%A),%D
    incw %D
    movw %D,(%A)
    leaw $LOOP,%A // goto LOOP
    jmp
    nop
WRITE:
    leaw $sum,%A
    movw (%A),%D
    leaw $1,%A
    movw %D,(%A)    // RAM[1] = the sum
END:
    leaw $END,%A
    jmp
    nop
```

OPTAB

mne	code
jmp	111
je	010
jne	101
...	...

código

Código de Máquina

```
WIDTH=16;
DEPTH=30;
ADDRESS_RADIX=UNS;
DATA_RADIX=BIN;
CONTENT BEGIN
    0 : 0000000000010000;
    1 : 1110111111001000;
    2 : 0000000000010001;
    3 : 1110101010001000;
    4 : 0000000000010000;
    5 : 1111110000010000;
    6 : 0000000000000000;
    7 : 1111010011010000;
    8 : 0000000000010111;
    9 : 1110001100000001;
   10 : 1110101010000000;
   11 : 0000000000010000;
   12 : 1111110000010000;
   13 : 0000000000010001;
   14 : 1111000010010000;
   15 : 1110001100001000;
   16 : 0000000000010000;
   17 : 1111110000010000;
   18 : 1110011111010000;
   19 : 1110001100001000;
   20 : 0000000000000100;
   21 : 1110101010000111;
   22 : 1110101010000000;
   23 : 0000000000010001;
   24 : 1111110000010000;
   25 : 0000000000000001;
   26 : 1110001100001000;
   27 : 0000000000011011;
   28 : 1110101010000111;
   29 : 1110101010000000;
END;
```

MIF



O arquivo MIF pode ser carregado diretamente no Z0 pelo gravador de FPGA do Quartus. O formato dele:

Definição:

```
WIDTH=<largura da instrução>;
DEPTH=<quantidade de instruções>;

ADDRESS_RADIX=<tipo>;
DATA_RADIX=<tipo>;

CONTENT BEGIN
    <índice> : <dado>;
    <índice> : <dado>;
    <índice> : <dado>;
    ...
    <índice> : <dado>;
END;
```

Exemplo:

```
WIDTH=16;
DEPTH=5;

ADDRESS_RADIX=UNS;
DATA_RADIX=BIN;

CONTENT BEGIN
    0 : 0000000000000010;
    1 : 1110101010001000;
    2 : 0000000000000001;
    3 : 1111110000010000;
    4 : 0000000000000000;
END;
```



Mais Dúvidas ?

Dúvidas da Vídeo Aula?

Lembrem-se:

Tragam sua dúvidas anotadas;

Verifiquem sites como:

- Google
- Stack Overflow
- Etc...

Usem o Slack, para perguntar para seus colegas, ninjas e o professor antes da aula

Formar Duplas

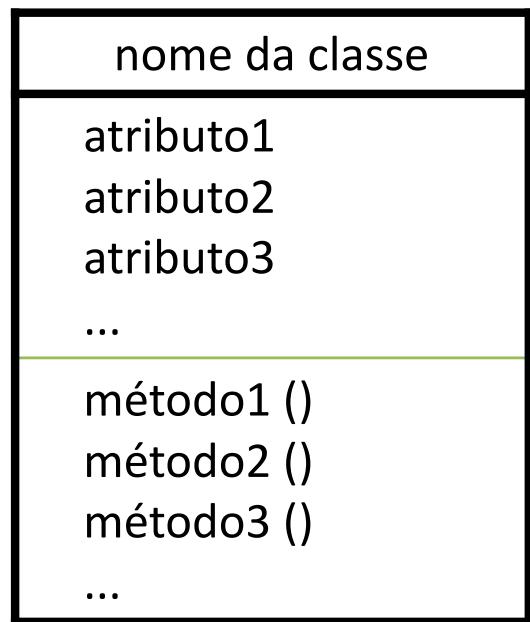
Almirante Grace Hopper está tentando provar que criar compiladores é viável. Vamos ajudar ela a provar isso começando pelo desenvolvimento de um Assembler para o Z0?

Formem duplas para fazer o projeto da arquitetura de software para esse Assembler.

Mantenha um ambiente em que todos participem de tudo.



Diagramas UML



relacionamento de
associação





Parser

Parser: Encapsula o código de leitura. Carrega as instruções na linguagem assembly, analisa, e oferece acesso as partes da instrução (campos e símbolos). Além disso, remove todos os espaços em branco e comentários.

Rotina	Argumentos	Retorno	Função
"construtor"	String	---	Abre o arquivo de entrada NASM e se prepara para analisá-lo.
advance	---	Boolean	Carrega uma instrução e avança seu apontador interno para o próximo linha do arquivo de entrada. Caso não haja mais linhas no arquivo de entrada o método retorna "Falso", senão retorna "Verdadeiro".
command		String	Retorna o comando "instrução" atual (sem o avanço)
commandType	String	A_COMMAND, C_COMMAND, L_COMMAND,	Retorna o tipo da instrução passada no argumento: A_COMMAND para leaw, por exemplo leaw \$1,%A L_COMMAND para labels, por ex. Xyz: , onde Xyz é um símbolo. C_COMMAND para todos os outros comandos
symbol	String	String	Retorna o símbolo ou valor numérico da instrução passada no argumento. Deve ser chamado somente quando commandType() é A_COMMAND.
label	String	String	Retorna o símbolo da instrução passada no argumento. Deve ser chamado somente quando commandType() é L_COMMAND.
instruction	String	String[]	Separa os mnemônicos da instrução fornecida em tokens em um vetor de Strings. Deve ser chamado somente quando CommandType () é C_COMMAND.



Code

Code: Traduz mnemônicos da linguagem assembly para códigos binários da arquitetura Z0.

Rotina	Argumentos	Retorno	Função
dest	String[]	3 bits	Retorna o código binário do(s) registrador(es) que vão receber o valor da instrução.
comp	String[]	7 bits	Retorna o código binário do mnemônico para realizar uma operação de cálculo.
jump	String[]	3 bits	Retorna o código binário do mnemônico para realizar uma operação de jump (salto).
toBinary	String	15 bits	Retorna o código binário de um valor decimal armazenado numa String.

SymbolTable

SymbolTable: Mantém uma tabela com a correspondência entre os rótulos simbólicos e endereços numéricos de memória.

Argumentos	Retorno	Função
---	---	Cria uma tabela de símbolos.
String , int	---	Insere uma entrada de um símbolo com seu endereço numérico na tabela de símbolos.
String	Boolean	Confere se o símbolo informado já foi inserido na tabela de símbolos.
String	int	Retorna o valor numérico associado a um símbolo já inserido na tabela de símbolos.

Assembler

AssemblerZ0: Classe principal que orquestra execução do Assembler.

Parametro	Argumentos	Função
arquivo	<arquivo nasm>	primeiro parâmetro é o nome do arquivo nasm a ser aberto
-f	<arquivo mif>	parâmetro -f <arquivo> indica onde será salvo o arquivo gerado .mif



Proposta de Implementação

Módulos de software:

Parser: Desmembra cada comando em seus campos;

Code: Traduz cada campo em seu valor binário correspondente e junta os valores resultantes;

SymbolTable: Gerencia a tabela de símbolos;

AssemblerZ0: Inicia leitura e escrita dos arquivos e controla aplicativo.

Sugestão de etapas de implementação:

Fase I: Criar um Assembler para programas com nenhum símbolo;

Fase II: Estender o Assembler básico com recursos de manipulação de símbolos.

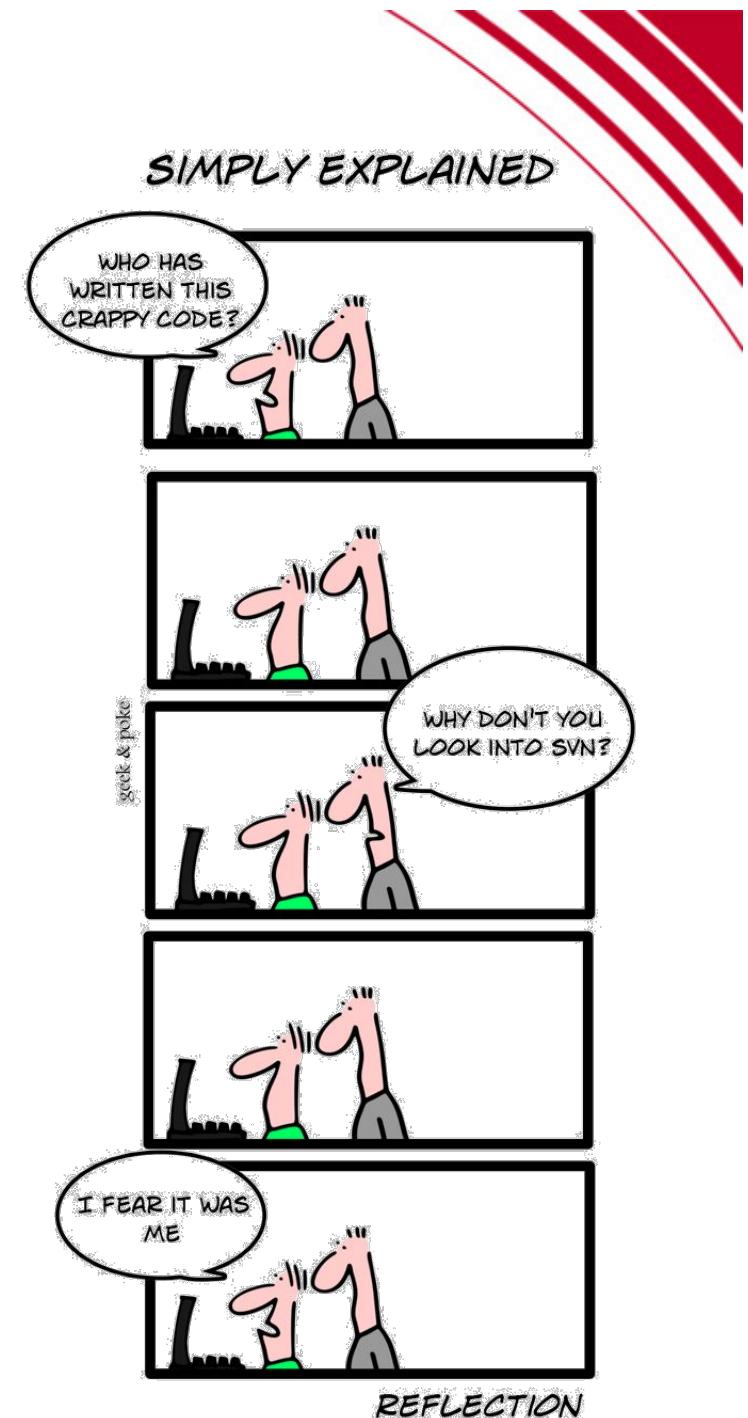
Integração Contínua (Travis-CI)

```
3107 Downloading: http://repo.maven.apache.org/maven2/org/apache/maven/surefire/surefire-junit4/2.12.4/surefire-junit4-2.12.4.jar
3108 Downloaded: http://repo.maven.apache.org/maven2/org/apache/maven/surefire/surefire-junit4/2.12.4/surefire-junit4-2.12.4.jar (37 KB at
655.6 KB/sec)
3109
3110 -----
3111 T E S T S
3112 -----
3113 Running assembler.AssemblerTest
3114 Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.003 sec
3115 Running assembler.ParserTest
3116 Tests run: 6, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.001 sec
3117 Running assembler.CodeTest
3118 Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.009 sec
3119 Running assembler.SymbolTableTest
3120 Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.085 sec
3121
3122 Results :
3123
3124 Tests run: 14, Failures: 0, Errors: 0, Skipped: 0
3125
3126 [INFO]
3127 [INFO] --- maven-jar-plugin:2.5:jar (default-jar) @ AssemblerZ0 ---
3128 Downloading: http://repo.maven.apache.org/maven2/org/apache/maven/maven-archiver/2.5/maven-archiver-2.5.pom
3129
3130 Downloaded: http://repo.maven.apache.org/maven2/org/apache/commons/commons-compress/1.5/commons-compress-1.5.jar (251 KB at 3626.6
KB/sec)
3131
3132 [INFO] Building jar: /home/travis/build/lpsoares/Z0_privado/Codigos/AssemblerZ0/target/AssemblerZ0-1.0.jar
3133 [INFO] -----
3134 [INFO] BUILD SUCCESS
3135 [INFO] -----
3136 [INFO] Total time: 13.387s
3137 [INFO] Finished at: Tue Apr 18 03:26:13 UTC 2017
3138 [INFO] Final Memory: 18M/177M
3139 [INFO] -----
3140
3141 ===== test session starts =====
3142 platform linux -- Python 3.5.2, pytest-3.0.5, py-1.4.31, pluggy-0.4.0 -- /home/travis/virtualenv/python3.5.2/bin/python
3143 cachedir: .cache
3144 rootdir: /home/travis/build/lpsoares/Z0_privado, inifile:
3145 collected 4 items
3146
3147 TestesSW/testeAssembler.py::test_Assembler[nop] PASSED
3148 TestesSW/testeAssembler.py::test_Assembler[loop] PASSED
3149 TestesSW/testeAssembler.py::test_Assembler[screen] PASSED
3150 TestesSW/testeAssembler.py::test_Assembler[keyboard] PASSED
3151
3152 ===== 4 passed in 0.85 seconds =====
3153
3154
3155
3156 The command "sh tests.sh" exited with 0.
3157
3158
3159
3160 Done. Your build exited with 0.
```

Top ▲

Recomendações

Esse é o primeiro projeto numa série. Tomem os diversos cuidados para ter um código legível e que vocês consigam reaproveitar (reuso).

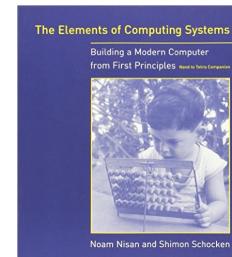


Próxima Aula

- Ver estudo para aula 16 sobre **Análise Sintática**
- Estudar Lista de Exercícios Aula 15 (opcional):
- Ler (opcional)

Capítulo 6

The Elements of Computing Systems
Building a Modern Computer from First Principles
Noam Nisan e Shimon Schocken



Insper

www.insper.edu.br